Security

1. Einleitung

2. Ausgangsprojekt aufbauen (REST mit DB-Zugriff)

1. Git-Projekt klonen und alle branches downloaden:
   https://github.com/aisge/securitydemo.git
   `git branch --all`
2. Datenbank starten
3. Security Policy definieren → Annotationen setzen
4. GET-Methoden: `@RolesAllowed("user")`
   UPDATE-Methoden: `@RolesAllowed("admin")`

3. Securing Quarkus services with Elytron

3.1 Einfachste Variante: Properties Files

1. `mvnw quarkus:add-extension -Dextension="io.quarkus:quarkus-elytron-security-properties-file"`
2. Application.properties erweitern:
   ```
   quarkus.security.users.file.enabled=true
   quarkus.security.users.file.users=test-users.properties
   quarkus.security.users.file.roles=test-roles.properties
   quarkus.security.users.file.realm-name=MyRealm
   quarkus.security.users.file.plain-text=true
   ```
3. test-users.properties: (nur Klartext oder MD5-Hashes möglich)
   ```
   max=passme
   susi=passme
   ```
4. test-roles.properties:
   ```
   max=admin,user
   susi=user
   ```
5. Für Hashes: MD5 (user ':' Realm ':' passwd)
   plain-text = false !!
   ```
   max=00AD7BAB4019042E5E96DB859E25F14C
   ```
6. Tests: .auth().preemtive().basic(„max“, „passme“)

3.2 Credentials in der Datenbank
Achtung: Funktioniert aktuell nicht im DEV-Modus !!

```
mvn -Dmaven.test.skip=true install
java -jar ./target/security-jdbc-quickstart-1.0-SNAPSHOT-runner.jar
```

1. `mvnw quarkus:add-extension -Dextension="io.quarkus:quarkus-elytron-security-jdbc"`
2. import.sql erweitern:
```
create table quarkus_user (  id int, username varchar2(50),
    password varchar2(255), role varchar2(255));
insert into quarkus_user values (1, 'max', 'passme', 'admin');
insert into quarkus_user values (2, 'susi', 'passme', 'user');
```
3. application.properties erweitern:
```
quarkus.security.jdbc.enabled=true
quarkus.security.jdbc.principal-query.sql=SELECT u.password,
     u.role FROM quarkus_user u where u.username=?
quarkus.security.jdbc.principal-query.clear-password-mapper.enabled=true
quarkus.security.jdbc.principal-query.clear-password-mapper.password-index=1
quarkus.security.jdbc.principal-query.attribute-mappings.0.index=2
quarkus.security.jdbc.principal-query.attribute-mappings.0.to=groups
```
4. Tests anpassen
5. Test mittels http-File in IntelliJ
(https://www.jetbrains.com/help/webstorm/exploring-http-syntax.html#)
```
GET http://localhost:8080/students
Accept: application/json
Authorization: Basic max passme

###

POST http://localhost:8080/students/
Content-Type: application/json
Authorization: Basic max passme

{ "userid": "it1000", "firstname": "Gerald", "lastname": "Aistleitner"}

###
```
6. Verschlüsselung
```
insert into quarkus_user(id, username, password, salt, iteration_count, role) values (1, 'max',
'sjl8xkG1Mc/yQF1Nengx3Ogg57Y5F0c=', 'ZE9EKfds3D7VT/0bTNCIgg==', 10, 'admin');

insert into quarkus_user(id, username, password, salt, iteration_count, role) values (2, 'susi',
'aPnJAMErXgXIR1RlsB1yegY2JmNeXps=','pc9CKhxWJmag8dbHzg7yKA==', 10, 'user');
```

```
quarkus.security.jdbc.enabled=true
quarkus.security.jdbc.principal-query.sql=SELECT u.password, u.salt, u.iteration_count, u.role FROM
quarkus_user u WHERE u.username=?
quarkus.security.jdbc.principal-query.bcrypt-password-mapper.enabled=true
quarkus.security.jdbc.principal-query.bcrypt-password-mapper.password-index=1
quarkus.security.jdbc.principal-query.bcrypt-password-mapper.salt-index=2
quarkus.security.jdbc.principal-query.bcrypt-password-mapper.iteration-count-index=3
quarkus.security.jdbc.principal-query.attribute-mappings.0.index=4
quarkus.security.jdbc.principal-query.attribute-mappings.0.to=groups
```

7. Optional: Password-Generator

```
@GET
@Path("/{password}")
public String getStudent(@PathParam("password") String password) throws Exception {
    PasswordFactory passwordFactory =
PasswordFactory.getInstance(BCryptPassword.ALGORITHM_BCRYPT, ELYTRON_PROVIDER);
    int iterationCount = 10;
    byte[] salt = new byte[BCryptPassword.BCRYPT_SALT_SIZE];
    SecureRandom random = new SecureRandom();
    random.nextBytes(salt);
    IteratedSaltedPasswordAlgorithmSpec iteratedAlgorithmSpec =
        new IteratedSaltedPasswordAlgorithmSpec(iterationCount, salt);
    EncryptablePasswordSpec encryptableSpec =
        new EncryptablePasswordSpec(password.toCharArray(),
                                    iteratedAlgorithmSpec);
    BCryptPassword original =
      (BCryptPassword) passwordFactory.generatePassword(encryptableSpec);
    byte[] hash = original.getHash();
    Base64.Encoder encoder = Base64.getEncoder();
    JsonObject result = Json.createObjectBuilder()
            .add("salt", encoder.encodeToString(salt))
            .add("hash", encoder.encodeToString(hash))
            .build();
    return result.toString();
}
```

4. JWT

1. Auschecken von SecurityDemo-Projekt (master-Branch)
   https://github.com/aisge/securitydemo.git
   `git branch -all`

2. jwt-Modul hinzufügen
   ./mvnw quarkus:add-extension -Dextensions="io.quarkus:quarkus-smallrye-jwt"

3. JWT in application.properties aktivieren/konfigurieren'
   ```
   mp.jwt.verify.publickey.location=
                                    META-INF/resources/publicKey.pem
   mp.jwt.verify.issuer=https://at.htl.4ahitm
   quarkus.smallrye-jwt.enabled=true
   ```

4. Key-Paar generieren und speichern als privateKey.pem / publicKey.pem:
   openssl req -newkey rsa:2048 -new -nodes -keyout privatekey.pem
   https://csfieldguide.org.nz/en/interactives/rsa-key-generator/
   (2048 bits, pkcs#8)

5. TokenUtils kopieren / implementieren
   private-Key-Pfad anpassen

6. Security-Endpoint anlegen und /login implementieren
   ```
   public class LoginData {
     String username;  String password;  // …constr, getter/setter
   }

   @Path("/security")
   public class SecurityEndpoint {
     @ConfigProperty(name = "mp.jwt.verify.issuer")
     String iss;

     @GET
     @Path("login")
     @Consumes(MediaType.APPLICATION_JSON)
     public Response login(LoginData data) {
       try {
         if ("aisge".equals(data.getUsername()) &&
                 "geheim".equals(data.getPassword())) {
           return Response.ok(generateToken()).build();
         }
       } catch (Exception e) {
         Logger.getAnonymousLogger().info(e.toString());
         e.printStackTrace();
     }
     return Response.status(401).build();
     }
   ```

```
private String generateToken() throws Exception {
Map<String, Long> timeClaims = new HashMap<>();
timeClaims.put(Claims.exp.name(),
 TokenUtils.currentTimeInSecs() + 120L);
Map<String, Object> claims = new HashMap<>();
claims.put(Claims.iss.name(), iss);
return TokenUtils.generateTokenString(claims, timeClaims);
 }
}
```

7. Testen von /login

```
GET http://localhost:8080/security/login

Content-Type: application/json
{
 "username": "aisge",
 "password": "geheim"
}

###
```

8. */info anlegen*

```
@Inject
JsonWebToken jwt;

@GET
@Path("info")
@PermitAll
public Response getInfo(@Context SecurityContext ctx) {
    Principal caller = ctx.getUserPrincipal();
    String name = caller == null ? "anonymous" :
                                   caller.getName();  // upn
   boolean hasJWT = jwt.getClaimNames() != null;
   String result = String.format(
        "user: %s, isSecure: %s, hasJWT: %s",
        name, ctx.isSecure(), hasJWT);
   return Response.ok(result).build();
}
```

9. getAll – Students mit @RolesAllowed absichern... → 401

```
@GET
@RolesAllowed("user")
public List<Student> getAll() {
   return studentRepository.findAll().list();
}
```

10. Groups im Token mit einbauen!

```
claims.put(Claims.groups.name(), Set.of(new String[] {"user"}));
```

5. KeyCloak

1. Installation von KeyCloak mittels Docker:

```
docker run -p 8180:8080 -e KEYCLOAK_USER=admin
   -e KEYCLOAK_PASSWORD=passme --name keycloak -d jboss/keycloak
```

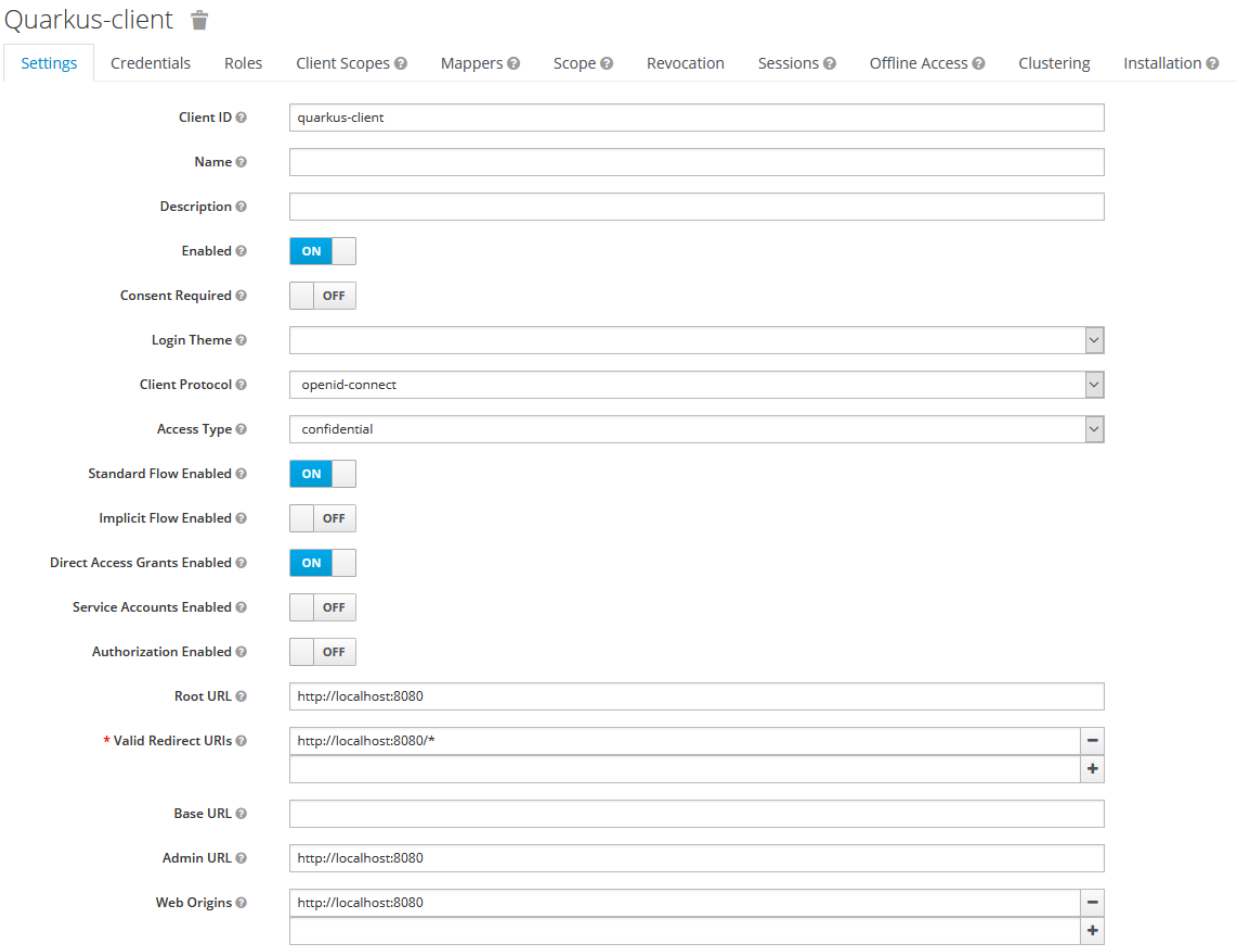
2. Anmelden unter http://localhost:8180

3. Realm anlegen (quarkus-realm)

4. Roles hinzufügen (admin, user = wie in unserer App)

5. User anlegen und Rollen zuweisen
   max: admin, user
   susi: user

6. Client konfigurieren:



7. Add KeyCloak-Modul in Quarkus-Projekt
   ./mvnw quarkus:add-extension -Dextensions="io.quarkus:quarkus-keycloak-authorization"

8. application.properties:
   ```
   keycloak.url=http://localhost:8180
   quarkus.oidc.enabled=true
   quarkus.oidc.auth-server-url=${keycloak.url}/auth/realms/quarkus-realm
   quarkus.oidc.client-id=quarkus-client
   quarkus.oidc.credentials.secret=###credential einsetzen###
   quarkus.keycloak.policy-enforcer.enable=true
   quarkus.http.cors=true
   ```

9. Ersten Test mit CURL oder HTTP-File durchführen
   ```
   curl -X POST http://192.168.99.101:8180/auth/realms/quarkus-realm/protocol/openid-connect/token
    --user quarkus-client:###credential einsetzen###
    -H "content-type: application/x-www-form-urlencoded"
    -d "username=susi&password=passme&grant_type=password"
   ```

   bzw:

   ```
   POST http://192.168.99.101:8180/auth/realms/quarkus-realm/protocol/openid-connect/token
   Authorization: Basic quarkus-client ###credential###
   Content-Type: application/x-www-form-urlencoded

   username=susi&password=passme&grant_type=password

   ###
   ```

10. Unit Tests vorbereiten indem Tokens geholt werden:
   ```
   @ConfigProperty(name="keycloak.url") String keycloakURL;
   @ConfigProperty(name="quarkus.oidc.credentials.secret")
   String credential;
   String userToken;
   String adminToken;

   @BeforeEach
   void setup() {
     // Only get tokens once
     // Workaround, because @BeforeAll needs static method where
        @ConfigProperty is not working...
     if (userToken != null) {
       return;
     }
     RestAssured.baseURI = keycloakURL;
     Response response =
   given().urlEncodingEnabled(true).auth().preemptive()
   .basic("quarkus-client", credential)
   ```

```
        .param("grant_type", "password")
        .param("client_id", "quarkus-client")
        .param("username", "susi")
        .param("password", "passme")
        .header("Accept", ContentType.JSON.getAcceptHeader())
        .post("/auth/realms/quarkus-realm/protocol/openid-
                connect/token")
        .then().statusCode(200).extract().response();
…
```

11. Unit-Tets mit oAuth2()-Aufrufen versehen

```
given()
    .auth().preemptive().oauth2(userToken)
    .when().get("/students")
```

6. Angular Client

1. Siehe https://www.linkedin.com/pulse/implicit-flow-authentication-using-angular-ghanshyam-shukla

2. Installation von Modul:
   npm i angular-oauth2-oidc –save

3. app.modules.ts
```
const appRoutes: Routes = [
 {path: 'secret', component: SecretComponent, canActivate:
[AuthGuard]},
 {path: '**', component: HelloComponent}
];

imports: [
 BrowserModule, HttpClientModule,
 OauthModule.forRoot(),
 RouterModule.forRoot(appRoutes)
],
```

4. app.component.ts
```
  constructor(private oauthService: OAuthService) {
  this.oauthService.configure(authCodeFlowConfig);
  this.oauthService.loadDiscoveryDocumentAndTryLogin();
  // optional
  this.oauthService.setupAutomaticSilentRefresh();
 }
}

export const authCodeFlowConfig: AuthConfig = {
 issuer: 'http://localhost:8180/auth/realms/quarkus',
 redirectUri: window.location.origin,
 clientId: 'my-backend-service',
 responseType: 'code',
 scope: 'openid profile email',
 showDebugInformation: true,
 requireHttps: false
};
```

5. home.component.html
```
<h1>Welcome {{givenName()}}</h1>
<button type="button" (click)="login()" *ngIf="!
    isLoggedIn()">Login</button>
<button type="button" (click)="logout()"
    *ngIf="isLoggedIn()">Logout</button>
```

6. home.component.ts

```
constructor(private oauthService: OAuthService) { }

login() {
 console.log('calling login...');
 this.oauthService.initLoginFlow();
}

logout() {
 this.oauthService.logOut();
}

isLoggedIn() {
 return this.oauthService.hasValidIdToken();
}

givenName() {
 if (!this.isLoggedIn()) {
  return '';
 }
 const claims: any = this.oauthService.getIdentityClaims();
 if (!claims) {
  return null;
 }
 return claims.given_name;
}
```

7. secret.component.html

```
<button type="button" (click)="getData()">getData</button>
<pre>
{{ info }}
</pre>
```

8. secret.component.ts

```
info;
constructor(
   private http: HttpClient,
   private oauthService:OAuthService) { }

getData() {
 const reqHeader = new HttpHeaders({
  'Content-Type': 'application/json',
  'Authorization': 'Bearer ' + this.oauthService.getAccessToken()
 });


 this.http.get('http://localhost:8080/users', {headers:
reqHeader}).subscribe(
  data => { this.info = JSON.stringify(data); console.log(data); },
  error => { console.log(error); this.info = error; }
 );
}
```

9. auth-guard.service.ts

```
@Injectable({
 providedIn: 'root'
```

```
})

export class AuthGuard implements CanActivate {
 constructor(private oauthService: OAuthService, private router:
Router) {
 }

 canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot): Observable<boolean
  | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
  if (this.oauthService.hasValidIdToken()) {
   return true;
  }
  this.router.navigate(['/']);
  return false;
 }
}
```

10. CORS-config

```
quarkus.http.cors=true
quarkus.http.cors.origins=http://localhost:4200,http://localhost:8180
/auth/realms/quarkus
quarkus.http.cors.methods=GET,POST
```

11.