JWT - JSON Web Token

JSON Web Token (JWT) ist ein offener Standard (RFC 7519), der eine kompakte, in sich geschlossene Möglichkeit zur Informationsübertragung zwischen Teilnehmern bietet.

JWTs werden zur Authentifizierung oder zum Informationsaustausch genutzt.

Ein JWT besteht aus drei Teilen:

- Header
- Payload
- Signature

JWT - Header

Der Header besteht aus zwei Teilen:

- Tokentyp
- Verschlüsselungsalgorithmus

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

Dieser JSON-String wird Base64Url-codiert
und das ergibt den ersten Teil des JWT.

JWT - Payload

Die Nutzlast eines JWT enthält sogenannte Claims.
Ein Claim ist eine Aussage über ein bestimmtes Objekt (üblicherweise der Benutzer).

Man unterscheidet:

- Reservierte Claims
  - **`iss`** (issuer)
  - **`exp`** (expiration time)
  - **`sub`** (subject)
  - **`aud`** (audience)
- Öffentliche Claims
  - scheinen in einem IANA JWT Register auf
- Private Claims
  - können selbst erstellt werden

```
{"sub": "1234567890", "name": "John Doe", "admin": true}
```

Der JSON-Payload-String wird ebenfalls Base64Url-codiert
und das ergibt den zweiten Teil des JWT.

JWT - Signature

Der dritte Teil des JWT bildet die Signatur.

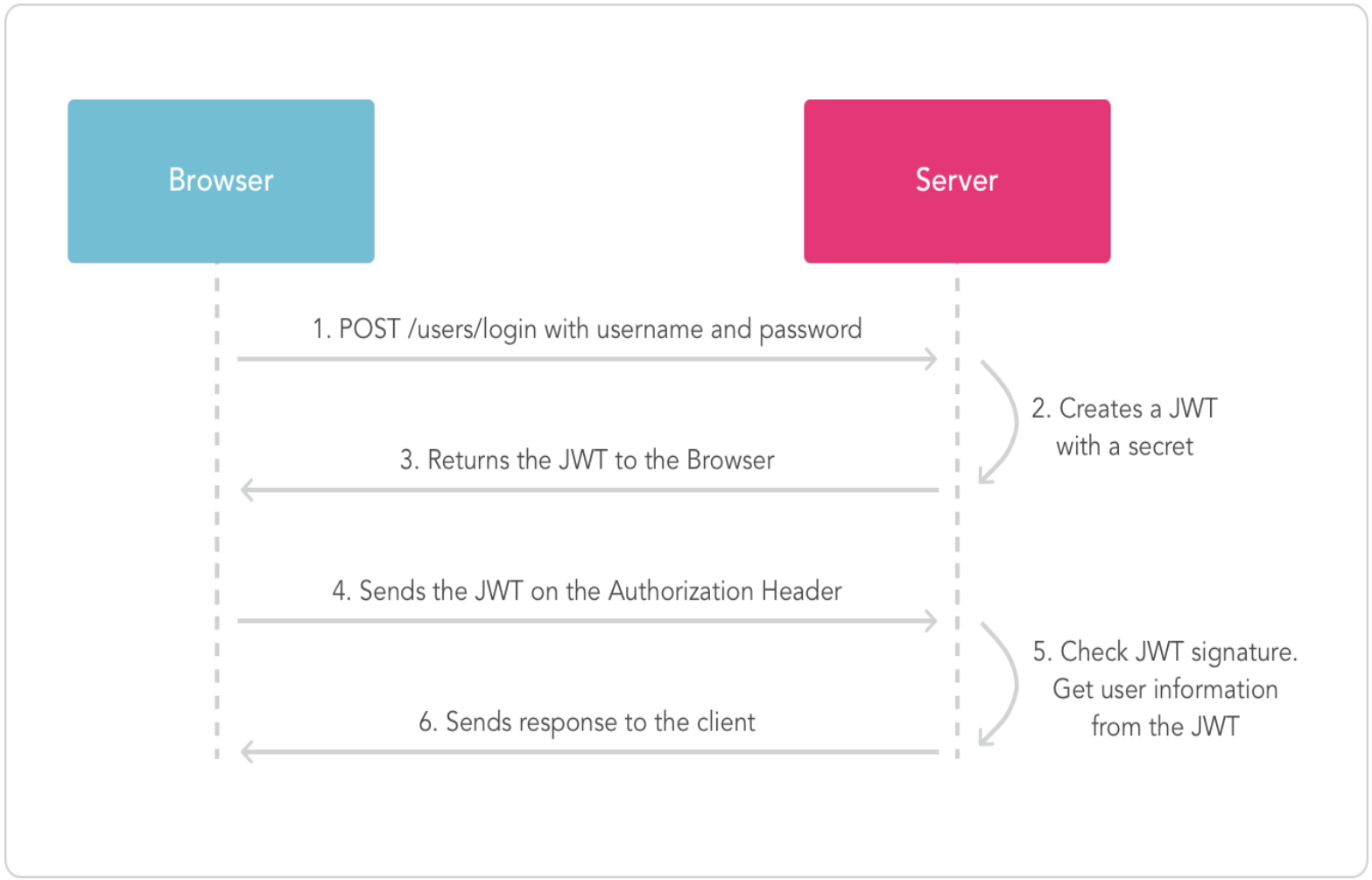
Diese wird erstellt, indem der erste und zweite durch einen Punkt getrennte codierte Teil mit dem angegeben Verschlüsselungsalgorithmus mit einem privaten Schlüssel signiert wird.
Daraus entsteht der dritte Teil.

```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret)
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzd
WIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9
lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RM
HrHDcEfxjoYZgeFONFh7HgQ
```

JSON Web Token

Ablauf



Server: Web-Token erzeugen

```
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt</artifactId>

<groupId>org.glassfish.jersey.media</groupId>
<artifactId>jersey-media-json-jackson</artifactId>
```

```
public class JwtBuilder {
    private String key = "secret";

    public String create(String subject) {
        return Jwts.builder()
            .signWith(SignatureAlgorithm.HS256, key)
            .setSubject(subject)
            .compact();
    }
     . . .
}
```

Server: Web-Token überprüfen

Abrufen des Subject-Claims:

```
public class JwtBuilder {
      . . .

    public String checkSubject(String token) {
        try {
            return Jwts.parser()
                .setSigningKey(key)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
        } catch (SignatureException e) {
            return null;
        }
    }
}
```

Server: Web-Token über REST verfügbar machen

```
@Path("jwt")
public class JwtResource {

    @POST
    @Consumes({MediaType.APPLICATION_JSON})
    public String post(User user) {
        if ( checkUser(user) )
            return new JwtBuilder().create(user.getUsername());
        return null;
    }

     . . .
}
```

Server: Filter für die Web-Token Überprüfung einrichten

```
@Provider
public class JwtFilter implements ContainerRequestFilter {
   @Override
   public void filter(ContainerRequestContext rc) {
      if ( rc.getUriInfo().getPath().contains("jwt") ||
           rc.getMethod().equals("OPTIONS") )
         return;
      JwtBuilder jwtbuilder = new JwtBuilder();
      try {
         String [] auth = rc.getHeaderString("Authorization")
                            .split("\\s");
         String subject = jwtbuilder.checkSubject(auth[1]);
      } catch ( Exception ex ) {
         throw new WebApplicationException(Status.UNAUTHORIZED);
      }
   }
}
```

Client: Web-Token anfordern

```
const body = { username: this.username, password: this.password };
this.http.post(`${this.baseUrl}jwt`, body, {responseType: 'text'})
    .subscribe(result => {
        if ( result ) {
            localStorage.setItem('token', result);
             . . .
        }
});
```

Client: REST-Service mit Web-Token aufrufen

Zur Analyse des Web-Token installiert man das Modul **`angular2-jwt`**

```
import { JwtHelper } from 'angular2-jwt';
```

Damit kann man z. Bsp. den Subject-Claim auslesen

```
let token = localStorage.getItem('token');
if ( token ) {
   let user = new JwtHelper().decodeToken(token).sub;
   this.message = `${user}, `;
   let headers = new HttpHeaders()
      .set('Authorization', `Bearer ${token}`)
   this.http.get(`${this.baseUrl}test/message`,
                 {headers: headers, responseType: 'text'})
      .subscribe(
         result => this.message += result,
         err => this.message = `Fehlercode ${err.status}`
   );
}
```

Client: Zugriff auf Komponenten einrichten

Der Zugriff auf die einzelnen Router-Pfade kann mit einem ***Authentication-Guard*** geschützt werden. Dieser muss das Interface **`CanActivate`** implementieren, welches die Methode **`canActivate()`** besitzt.
Diese Methode entscheidet über einen **`boolean`**-Rückgabewert, ob der Zugriff auf einen Router-Pfad erlaubt ist:

```
@Injectable()
export class AuthGuardService implements CanActivate {

   constructor(public router: Router) { }

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

Client: Zugriff auf Komponenten einrichten

Der ***Authentication-Guard*** wird bei der Einrichtung der Routen beim `canActivate` Attribut angegeben:

```
const appRoutes: Routes = [
    { path: 'login', component: LoginComponent },
    { path: 'message', component: MessageComponent,
           canActivate: [AuthGuard] },
    { path: '**', redirectTo: 'login' }
]
```

Client: Zugriff auf Komponenten einrichten

Der Zugriff auf eine Komponente soll nur mit einem gültigen Web-Token möglich sein:

```
@Injectable()
export class AuthGuardService implements CanActivate {

    . . .

   public isAuthenticated(): boolean {
      const token = localStorage.getItem('token');
      if ( token && token != 'null')
         return !new JwtHelper().isTokenExpired(token);
      return false;
   }
    . . .
}
```