

Ktor Authentication and Authorization

So to retain the client information after login we have two options:

1. Create a server-side session.
2. Store the session in a Json token on the Client side

With server-side sessions, you will either have to store the session identifier in a database, or else keep it in memory and make sure that the client always hits the same server. Both of these have drawbacks. In the case of the database (or other centralised storage), this becomes a bottleneck and a thing to maintain - essentially an extra query to be done with every request.

With an in-memory solution, you limit your horizontal scaling, and sessions will be affected by network issues (clients roaming between Wifi and mobile data, servers rebooting, etc).

Moving the session to the client means that you remove the dependency on a server-side session, but it imposes its own set of challenges.

- Storing the token securely.
- Transporting it securely.
- JWT sessions can sometimes be hard to invalidate.
- Trusting the client's claim.

And I'm not going to go into much detail on the types of encryption or cryptography. I'll be using RSA-256 encryption, which requires a public and private key. Ktor has a great example of JWT authentication with RSA-256 encryption. I've stolen some of their code below, but I decided to use `javax.security` rather than the `ktor JwkProvider`.

Basic Auth

Setting up basic authentication in Ktor is pretty straightforward. What gets more complicated is storing the user session in a token or in-memory session.

In the validation function we can choose how we want to verify the username and password. I like to create a database to store the username and a hashed version of the password, and then

validate the user in the repository. Something easier would be to create an in-memory hashtable of users and passwords, all in the docs.

```
install(Authentication) {  
  
    basic("auth-basic") {  
        validate { credentials ->  
            val login = LoginRepo(loginConfig)  
            val validation = login.validateUser(credentials.name, credentials.password)  
            login.close()  
            validation  
        }  
    }  
}
```

Now when ever I make a request to a route protected by "auth-basic" the application will automatically search the request for a basic authentication credentials.

What I do is have my login route return a RS256 signed JWT token after being signed in, so I just wrap the route in an authentication block.

```
authenticate("auth-basic") {  
    get("/LOGIN") {  
        log.info("Starting login sequence")  
        val publicKey = jwkProvider.get(jwtConfig.pubKeyId).publicKey  
        val keySpecPKCS8 = PKCS8EncodedKeySpec(Base64.getDecoder().decode(jwtConfig.privateKey))  
        val privateKey = KeyFactory.getInstance("RSA").generatePrivate(keySpecPKCS8)  
        val user = this.call.authentication.principal<LoginEntity>()  
        val token = JWT.create()  
            .withAudience(jwtConfig.audience)  
            .withIssuer(jwtConfig.issuer)  
            .withClaim("NAME", user?.name)  
            .withClaim("ROLE", user?.role)  
            .withClaim("EMAIL", user?.email)  
            .withExpiresAt(Date(System.currentTimeMillis() + 60000))  
            .sign(Algorithm.RSA256(publicKey as RSAPublicKey, privateKey as RSAPrivateKey))  
        call.respond(hashMapOf("token" to token))  
    }  
}
```

Note there is some the token is signed with an RSA256 algorithm above, more about that in another chapter. Also I'm returning the token in the response body. This is ok, but the production

standard is to store the token in an SSL-encrypted cookie. That way, we don't have to send the token back and forth or store anything in session storage. Also this reduces the risk of cross-site-scripting. More on that in a second.

JSON Web Tokens

"JWTs are an open standard that defines a way for securely transmitting information between parties as a JSON object."

JWTs are used for quick Authorization, not authentication. Store things like a user's role, email, or other nonsensitive information in the payload. NEVER use a token to store a password.

Ktor handles JWTs passed in the Authorization header using the Bearer scheme like so:

```
Authorization: Bearer {{auth_token}}
```

See my Angular page on JWTs for more info.

The following dependencies are required:

```
implementation "io.ktor:ktor-auth:$ktor_version"
implementation "io.ktor:ktor-auth-jwt:$ktor_version"
```

Authorization Flow:

1. Client makes a POST request with credentials:

```
POST http://localhost:8080/login
```

```
Content-Type: application/json
```

```
{
  "username": "jetbrains",
  "password": "foobar"
}
```

2. If the credentials are validate the server generates a JSON web token and signs it with the specified algorithm
 3. Server sends generated JWT to a client
- A client can now make a request to a protext resource with JSON scheme in the header

```
GET http://localhost:8080/hello
Authorization: Bearer {{auth_token}}
```

4. Server receives the request and performs the following validations:
 - verify the signature of a JSON object
 - perform additional validations on the JWT payload
5. After validation server responds with contents of protected resource

Installation and Configuration

Add the JWT function to the install Authentication module, you can define your private key, issuer, audience and realm in application.conf,

```
val privateKeyString = environment.config.property("jwt.privateKey").getString()
val issuer = environment.config.property("jwt.issuer").getString()
val audience = environment.config.property("jwt.audience").getString()
val myRealm = environment.config.property("jwt.realm").getString()
val jwkProvider = JwkProviderBuilder(issuer)
    .cached(10, 24, TimeUnit.HOURS)
    .rateLimited(10, 1, TimeUnit.MINUTES)
    .build()
install(Authentication) {
    jwt("auth-jwt") {
        realm = myRealm
        verifier(jwkProvider, issuer) {
            acceptLeeway(3)
        }
        validate { credential ->
            if (credential.payload.getClaim("username").asString() != "") {
                JWTPrincipal(credential.payload)
            } else {
                null
            }
        }
    }
}
```

Alternatively, there's no downside to generating a new private key every run. In the official Ktor example they create the JWT token directly in the routing function. I prefer to create a separate class for token generation and validation that looked something like this:

```

class JWTService(private val jwtConfig: JwtConfig, private val jwkProvider: JwkProvider) {

    private val privateKey: PrivateKey

    init {
        val keySpecPKCS8 = PKCS8EncodedKeySpec(Base64.getDecoder().decode(jwtConfig.privateKey))
        privateKey = KeyFactory.getInstance("RSA").generatePrivate(keySpecPKCS8)
    }

    fun generateToken(user: LoginEntity): String = JWT.create()
        .withAudience(jwtConfig.audience)
        .withIssuer(jwtConfig.issuer)
        .withClaim("NAME", user.name)
        .withClaim("ROLE", user.role)
        .withClaim("EMAIL", user.email)
        .withExpiresAt(Date(System.currentTimeMillis() + 60000))
        .sign(Algorithm.RSA256(jwkProvider.get(jwtConfig.pubKeyId).publicKey as RSAPublicKey, privateKey
as RSAPrivateKey))

    fun verifyToken(token: String?): Boolean {
        if (token == null) {
            println("No token found in memory")
            return false
        }
        val payloadJson = validatedToken(token) ?: return false
        return (payloadJson["ROLE"] == jwtConfig.realm)
    }

    fun getLoginEntity(token: String): LoginEntity? {
        val payloadJson = validatedToken(token) ?: return null
        val name = payloadJson["NAME"].toString()
        val email = payloadJson["EMAIL"].toString()
        val role = payloadJson["ROLE"].toString()
        return LoginEntity(name, email, role)
    }

    private fun validatedToken(validateToken: String): JSONObject? {
        try {
            val encodedPayload = JWT.require(
                Algorithm.RSA256(

```

```

        jwkProvider.get(jwtConfig.pubKeyId).publicKey as RSAPublicKey,
        privateKey as RSAPrivateKey
    )
)
    .build()
    .verify(validateToken)
    .payload
val payload = String(Base64.getDecoder().decode(encodedPayload))
val parser = JSONParser()
return parser.parse(payload) as JSONObject
} catch (jwtException: JWTVerificationException) {
    println("Failed to verify JWT: " + jwtException.message)
    return null
} catch (exception: Exception) {
    println("An error occurred: " + exception.message)
    return null
}
}
}
}

```

Now that we have that ready I change the login function to look like so:

```

authenticate("auth-basic") {
    get(CommonRoutes.LOGIN) {
        log.info("Starting login sequence")
        val user = this.call.authentication.principal<LoginEntity>()!!
        val token = jwtService.generateToken(user)
        // "secure=true" will only work when a valid HTTPS certificate is present!
        val cookie = Cookie("token", token, httpOnly = true, secure = true)
        call.response.cookies.append(cookie)
        call.respond(Response(status = "ok"))
    }
}

authenticate("auth-jwt"){
    get("getRole") {
        val entity = jwtService.getLoginEntity(call.request.cookies["token"]!!) ?: LoginEntity("", "", "")
        call.respond(entity)
    }
}

get("logout") {

```

```
        call.response.cookies.appendExpired("token")
        call.respond(Response(status = "ok"))
    }
}
```

As well as implemented cookies, I added a "getRole" route that checks for cookies and returns a user entity if the user is logged in. Using this we can prevent the user from having to login everytime the page is refreshed.

Then we can protect a route with the following syntax:

```
routing {
    authenticate("jwt-auth") {
        get("/") {
            call.respondText("Hello, ${call.principal<UserIdPrincipal>()?.name}!")
        }
    }
}
```

Generating a Self-Signed Certificate

Above I mentioned the best security practice is to SSL-encrypt our cookies over https. This requires a SSL-certificate. This may be a bit difficult to set up depending on how you are running Ktor. I'm using a docker container with an nginx reverse proxy and a cloudflare domain. I already have an SSL certfate for my domain, but the problem was the my docker was running on http which was causing issues when using `secure` cookies.

In many situtations it is bad practice to use self-signed certificates, but in the case of an LAN address that only we have access to I think it will be okay. TODO: More research on if a reverse proxy exposes the keys on the host.

Ktor has a great library for generating self-signed certificates within a embedded server, but it's labeled as only for testing purposes. Instead I'll generate the ssl certificate with *Let's Encrypt* and store that in a keystore generated manually with `keytool` and add the configuration to `application.conf`.

In a nutshell, steps are as follows:

1. Pulling the Let's Encrypt client (certbot).
2. Generating a certificate for your domain (e.g. example.com)

```
./certbot-auto certonly -a standalone -d example.com -d www.example.com
```

Things are generated in `/etc/letsencrypt/live/example.com`. Industry standard is PKCS12 formatted file. Convert the keys to a PKCS12 keystore using OpenSSL as follows:

Open `/etc/letsencrypt/live/example.com` directory.

```
openssl pkcs12 -export -in fullchain.pem -inkey privkey.pem -out keystore.p12 -name tomcat -CAfile chain.pem -caname root
```

The file `keystore.p12` with PKCS12 is now generated in `/etc/letsencrypt/live/example.com`.

It's time to configure your Spring Boot application. Open the `application.properties` file and put following properties there:

```
server.port=8443 security.require-ssl=true server.ssl.key-store=/etc/letsencrypt/live/example.com/keystore.p12 server.ssl.key-store-password=server.ssl.keyStoreType=PKCS12 server.ssl.keyAlias=tomcat
```

Read my blog post for further details and remarks.

Revision #2

Created 17 April 2022 00:23:54 by Elkip

Updated 17 April 2022 01:32:23 by Elkip