ANGULAR UNIVERSITY

PREMIUM QUALITY
ANGULAR TUTORIALS

angular-university.io

Angular Security - Authentication With JSON Web Tokens (JWT): The Complete Guide

This post is a step-by-step guide for both designing and implementing JWT-based Authentication in an Angular Application.

The goal here is to discuss **JWT-based Authentication Design and Implementation** in general, by going over the multiple design options and design compromises involved, and then apply those concepts in the specific context of an Angular Application.

We will follow the complete journey of a JWT from creation on the Authentication server and back to the client, and then back to the Application server and talk about all the design options and decisions involved.

Because Authentication also requires some server code, we will show that too so that we have the whole context and can see how all the multiple parts work together.

The server code will be in Node / Typescript, as it's very familiar to Angular developers, but the concepts covered are not Node-specific.

If you use another server platform, it's just a matter of choosing a JWT library for your platform at jwt.io, and with it, all the concepts will still apply.

Table of Contents

In this post we will cover the following topics:

So without further ado, let's get started learning JWT-based Angular Authentication!

JWT-based User Sessions

Let's start by introducing how JSON Web Tokens can be used to establish a user session: in a nutshell, JWTs are digitally signed JSON payloads, encoded in a URL-friendly string format.

A JWT can contain any payload in general, but the most common use case is to use the payload to define a user session.

The key thing about JWTs is that in order to confirm if they are valid, we only need to inspect the token itself and validate the signature, without having to contact a separate server for that, or keeping the tokens in memory or in the database between requests.

If JWTs are used for Authentication, they will contain at least a user ID and an expiration timestamp.

If you would like to know all the details about the JWT format in-depth including how the most common signature types work, have a look at this post JWT: The Complete Guide to JSON Web Tokens.

If you are curious to know what a JWT looks like, here is an example:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIzNTM0NTQzN
TQzNTQzNTM0NTMiLCJleHAiOjE1MDQ2OTkyNTZ9.zG-
2FvGegujxoLWwIQfNB5IT46D-xC4e8dEDYwi6aRM
```

You might be thinking: this does not look like JSON! Where is the JSON then?

To see it, let's head over to jwt.io and paste the complete JWT string into the validation tool, we will then see the JSON Payload:

```
{
  "sub": "3534543543543534 53",
  "exp": 1504699256
}
```

The `sub` property contains the user identifier, and the `exp` property contains the expiration timestamp. This type of token is known as a Bearer Token, meaning that it identifies the user that owns it, and defines a user session.

> *A bearer token is a signed temporary replacement for the username/password combination!*

If you would like to learn further about JWTs, have a look here. For the remainder of this post, we will assume that a JWT is a string containing a verifiable JSON payload, which defines a user session.

The very first step for implementing JWT-based Authentication is to issue a bearer token and give it to the user, and that is the main purpose of a Login / Sign up page.

Step 1 - The Login Page

Authentication starts with a Login page, which can be hosted either in our domain or in a third-party domain. In an enterprise scenario, the login page is often hosted on a separate server, which is part of a company-wide Single Sign-On solution.

On the public Internet, the login page might also be:

- hosted by a third-party Authentication provider such as Auth0
- available directly in our single page application using a login screen route or a modal

A separately hosted login page is an improvement security-wise because this way the password is never directly handled by our application code in the first place.

The separately hosted login page can have minimal Javascript or even none at all, and it could be styled to make it look and feel as part of the whole application.

But still, logging in a user via a login screen inside our application is also a viable and commonly used solution, so let's cover that too.

Login page directly on the SPA application

If we would create a login page directly in our SPA, this is what it would look like:

```
1
2  @Component({
3    selector: 'login',
4    template: `
5  <form [formGroup]="form">
6      <fieldset>
7          <legend>Login</legend>
8          <div class="form-field">
9              <label>Email:</label>
10             <input name="email" formControlName="email">
11         </div>
12         <div class="form-field">
```

```
            <label>Password:</label>
            <input name="password" formControlName="password"
                   type="password">
        </div>
    </fieldset>
    <div class="form-buttons">
        <button class="button button-primary"
                (click)="login()">Login</button>
    </div>
</form>`})
export class LoginComponent {
    form:FormGroup;

    constructor(private fb:FormBuilder,
                private authService: AuthService,
                private router: Router) {

        this.form = this.fb.group({
            email: ['',Validators.required],
            password: ['',Validators.required]
        });
    }

    login() {
        const val = this.form.value;

        if (val.email && val.password) {
            this.authService.login(val.email, val.password)
                .subscribe(
                    () => {
                        console.log("User is logged in");
                        this.router.navigateByUrl('/');
                    }
                );
        }
    }
}

```

As we can see, this page would be a simple form with two fields: the email and the password. When the user clicks the Login button, the user

and password are then sent to a client-side Authentication service via a `login()` call.

Why create a separate Authentication service?

Putting all our client authentication logic in a centralized application-wide singleton `AuthService` will help us keep our code organized.

This way, if for example later we need to change security providers or refactor our security logic, we only have to change this class.

Inside this service, we will either use some Javascript API for calling a third-party service, or the Angular HTTP Client directly for doing an HTTP POST call.

In both cases, the goal is the same: to get the user and password combination across the network to the Authentication server via a POST request, so that the password can be validated and the session initiated.

Here is how we would build the HTTP POST ourselves using the Angular HTTP Client:

```
1
2  @Injectable()
3  export class AuthService {
4
5      constructor(private http: HttpClient) {
6      }
7
8      login(email:string, password:string ) {
9          return this.http.post<User>('/api/login', {email, password})
10             // this is just the HTTP call,
11             // we still need to handle the reception of the token
```

```
            .shareReplay();
    }
}
```

We are calling `shareReplay` to prevent the receiver of this Observable from accidentally triggering multiple POST requests due to multiple subscriptions.

Before processing the login response, let's first follow the flow of the request and see what happens on the server.

Step 2 - Creating a JWT Session Token

Whether we use a login page at the level of the application or a hosted login page, the server logic that handles the login POST request will be the same.

The goal is in both cases to validate the password and establish a session. If the password is correct, then the server will issue a bearer token saying:

> *The bearer of this token is the user with the technical ID 353454354354353453, and the session is valid for the next two hours*

The token should then be signed and sent back to the user browser! The key part is the JWT digital signature: that is the only thing that prevents an attacker from forging session tokens.

This is what the code looks like for creating a new JWT session token, using Express and the node package node-jsonwebtoken:

```
import {Request, Response} from "express";
import * as express from 'express';
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
import * as jwt from 'jsonwebtoken';
import * as fs from "fs";

const app: Application = express();

app.use(bodyParser.json());

app.route('/api/login')
    .post(loginRoute);

const RSA_PRIVATE_KEY = fs.readFileSync('./demos/private.key');

export function loginRoute(req: Request, res: Response) {

    const email = req.body.email,
          password = req.body.password;

    if (validateEmailAndPassword()) {
       const userId = findUserIdForEmail(email);

        const jwtBearerToken = jwt.sign({}, RSA_PRIVATE_KEY, {
                algorithm: 'RS256',
                expiresIn: 120,
                subject: userId
            }

          // send the JWT back to the user
          // TODO - multiple options available
    }
    else {
        // send status 401 Unauthorized
        res.sendStatus(401);
    }
}
```

There is a lot going on in this code, so we will break it down line by line:

- We start by creating an Express appplication
- next, we configure the `bodyParser.json()` middleware, to give Express the ability to read JSON payloads from the HTTP request body
- We then defined a route handler named `loginRoute`, that gets triggered if the server receives a POST request targetting the `/api/login` URL

Inside the `loginRoute` method we have some code that shows how the login route can be implemented:

- we can access the JSON request body payload using `req.body`, due to the presence of the `bodyParser.json()` middleware
- we start by retrieving the email and password from the request body
- then we are going to validate the password, and see if it's correct
- if the password is wrong then we send back the HTTP status code 401 Unauthorized
- if the password is correct, we start by retrieving the user technical identifier
- we then create a a plain Javascript object with the user ID and an expiration timestamp and we send it back to the client
- We sign the payload using the node-jsonwebtoken library and choose the RS256 signature type (more on this in a moment)
- The result of the `.sign()` call is the JWT string itself

To summarize, we have validated the password and created a JWT session token. Now that we have a good picture of how this code works,

let's focus on the key part which is the signing of the JWT containing the user session details, using an RS256 signature.

Why is the type of signature important? Because without understanding it we won't understand the Application server code that we will need to validate this token.

What are RS256 Signatures?

RS256 is a JWT signature type that is based on RSA, which is a widely used public key encryption technology.

> *One of the main advantages of using a RS256 signature is that we can separate the ability of creating tokens from the ability to verify them.*

You can read all about the advantages of using this type of signatures in the JWT Guide, if you would like to know how to manually reproduce them.

In a nutshell, RS256 signatures work in the following way:

- a private key (like `RSA_PRIVATE_KEY` in our code) is used for signing JWTs
- a public key is used to validate them
- the two keys are not interchangeable: they can either only sign tokens, or only validate them, but neither key can do both things

Why RS256?

Why use public key crypto to sign JWTs? Here are some examples of both security and operational advantages:

- we only have to deploy the private signing key in the Authentication Server, and not on the multiple Application servers that use the same Authentication Server
- We don't have to shut down the Authentication and the Application servers in a coordinated way, in order to change a shared key everywhere at the same time
- the public key can be published in a URL and automatically read by the Application server at startup time and periodically

This last part is a great feature: being able to publish the validating key gives us built-in key rotation and revocation, and we will implement that in this post!

This is because in order to enable a new key pair we simply publish a new public key, and we will see that in action.

RS256 vs HS256

Another commonly used signature is HS256, that does not have these advantages.

HS256 is still commonly used, but for example providers such as Auth0 are now using RS256 by default. If you would like to learn more about HS256, RS256 and JWT signatures in general, have a look at this post.

Independently of the signature type that we use, we need to send the freshly signed token back to the user browser.

Step 3 - Sending a JWT back to the client

We have several different ways of sending the token back to the user, for example:

- In a Cookie
- In the Request Body
- In a plain HTTP Header

JWTs and Cookies

Let's start with cookies, why not use them? JWTs are sometimes mentioned as an alternative to Cookies, but these are two very different concepts. Cookies are a browser data storage mechanism, a place where we can safely store a small amount of data.

That data could be anything such as for example the user preferred language, but it can also contain a user identification token such as for example a JWT.

So we can for example, store a JWT *in* a cookie! Let's then talk about the advantages and disadvantages of using cookies to store JWTs, when compared to other methods.

How the browser handles cookies

A unique aspect of cookies is that the browser will automatically with each request append the cookies for a particular domain or sub-domain to the headers of the HTTP request.

This means that if we store the JWT in a cookie, we will not need any further client logic for sending back the cookie to the application server with each request, assuming the login page and the application share the same root domain.

Let's then store our JWT in a cookie, and see what happens. Here is how we would finish the implementation of our login route, by sending the

JWT back to the browser in a cookie:

```
... continuing the implementation of the Express login route

// this is the session token we created above
const jwtBearerToken = jwt.sign(...);

// set it in an HTTP Only + Secure Cookie
res.cookie("SESSIONID", jwtBearerToken, {httpOnly:true, secure:true});

```

Besides setting a cookie with the JWT value, we also set a couple of security properties that we are going to cover next.

Unique security properties of Cookies - HttpOnly and Secure Flags

Another unique aspect of Cookies is that they have some security-related properties that help with ensuring secure data transfer.

A Cookie can be marked as Secure, meaning that the browser will only append the cookie to the request if it's being made over an HTTPS connection.

A Cookie can also be marked as Http Only, meaning that it's not accessible by the Javascript code *at all*! Note that the browser will still append the cookie to each request sent back to the server, just like with any other cookie.

This means for example that in order to delete a HTTP Only cookie, we need to send a request to the server, like for example to logout the user.

Advantages of HTTP Only cookies

One advantage of an HTTP Only cookie is that if the application suffers, for example, a script injection attack (or XSS), the Http Only flag would still, in this disastreous scenario, prevent the attacker from getting access to the cookie and use it to impersonate the user.

The two flags Secure and Http Only can and are often used together for maximum security, which might make us think that Cookies are the ideal place for storing a JWT.

But Cookies have some disadvantages too, so let's talk about those: this will help us decide if storing cookies in a JWT is a good approach for our application.

Disadvantages of Cookies - XSRF

Applications with Bearer tokens stored in a Cookie suffer from a vulnerability called Cross-Site Request Forgery, also known as XSRF or CSRF. Here is how it works:

- somebody sends you a link and you click on it
- The link ends up sending an HTTP request to the site under attack containing all the cookies linked to the site
- And if you were logged into the site this means the Cookie containing our JWT bearer token will be forwarded too, this is done automatically by the browser
- The server receives a valid JWT, so there is no way for the server to distinguish this attack from a valid request

This means that an attacker could trick a user to do certain actions on its behalf, just by sending an email, or posting a link in a public forum.

This attack is less powerful than it might look but the problem is that it's very easy to perform: all it takes is an email or a post on social media.

We will cover in detail this attack in a future post, right now it's important to realize that if we choose to store our JWT in a cookie then we need to also put in place some defenses against XSRF.

The good news is that all major frameworks come with defenses that can be easily put in place against XSRF, as it's such a well-known vulnerability.

Like it happens many times, there is a design tradeoff going on here with Cookies: using them means leveraging HTTP Only which is a great defense against script injection, but on the other hand, it introduces a new problem - XSRF.

Cookies and Third-Party Authentication providers

A potential problem with receiving the session JWT in a cookie is that we would not be able to receive it from a third-party web domain, that handles the authentication logic.

This is because an application running on `app.example.com` cannot access cookies from another domain like `security-provider.com`.

So in that scenario, we would not be able to access the cookie containing the JWT, and send it to our server for validation, making the use of cookies unfeasible.

Can we get the best of the two solutions?

Third-party authentication providers might allow us to run the externally hosted login page in a configurable subdomain of our website, such as for example `login.example.com`.

So it would be possible to get the best of all these solutions combined. Here is what the solution would look like:

- an externally hosted login page running on our own subdomain `login.example.com`, and an application running on `example.com`
- that page sets an HTTP Only and Secure Cookie containing the JWT, giving us good protection against many types of XSS attacks that rely on stealing user identity
- Plus we need to add some XSRF defenses, but there are well-understood solutions for that

This would give us maximum protection against both password and identity token theft scenarios:

- the Application never gets the password in the first place
- the Application code never accesses the session JWT, only the browser
- the application is not vulnerable to request forgery (XSRF)

This scenario is sometimes used in enterprise portals and gives great security features. However, this relies on the security provider or enterprise security proxy that we are using to support a custom domain for hosted login pages.

This feature (custom subdomain for hosted login page) is however not always available, and that would render the HTTP Only cookie approach undoable.

If your application falls into that case or if you are looking for alternatives that don't rely on cookies, let's go back to the drawing board and find what else we can do.

Sending the JWT back in the HTTP response body

Cookies with their unique HTTP Only property are a solid choice for storing JWTs, but there are other good choices available. For example, instead of cookies we are going to send the JWT back to the client in the HTTP Response body.

Not only do we want to send back the JWT itself, but it's better to send also the expiration timestamp as a separate property.

It's true that the expiration timestamp is also available inside the JWT, but we want to make it simple for the client to obtain the session duration without having to install a JWT library just for that.

Here is how we can send the JWT back to the client in the HTTP response body:

```
1
2  ... continuing the implementation of the Express login route
3
4  // this is the session token we created above
5  const jwtBearerToken = jwt.sign(...);
6
7  // set it in the HTTP Response body
```

```
8  res.status(200).json({
9    idToken: jwtBearerToken,
10   expiresIn: ...
11 });
12
13
```

And with this, the client will receive both the JWT and its expiration timestamp.

Design compromises of not using Cookies for JWT storage

Not using cookies has the advantage that our application is no longer vulnerable to XSRF, which is one advantage of this approach.

But this also means that we will have to add some client code to handle the token, because the browser will no longer forward it to the application server with each request.

This also means that the JWT token is now readable by an attacker in case of a successful script injection attack, while with the HTTP Only cookie that was not possible.

This is a good example of the design compromises that are often associated with choosing a security solution: there is usually a security vs convenience trade-off going on.

Let's then continue following the journey of our JWT Bearer Token. Since we are sending the JWT back to the client in the request body, we will need to read it and handle it.

Step 4 - Storing and using the JWT on the client side

Once we receive the JWT on the client, we need to store it somewhere, otherwise, it will be lost if we refresh the browser and would have to log in again.

There are many places where we could save the JWT (other than cookies). A practical place to store the JWT is on Local Storage, which is a key/value store for string values that is ideal for storing a small amount of data.

Note that Local Storage has a synchronous API. Let's have a look at an implementation of the login/logout logic using Local Storage:

```
1
2  import * as moment from "moment";
3
4  @Injectable()
5  export class AuthService {
6
7      constructor(private http: HttpClient) {
8
9      }
10
11     login(email:string, password:string ) {
12         return this.http.post<User>('/api/login', {email, password})
13             .do(res => this.setSession)
14             .shareReplay();
15     }
16
17     private setSession(authResult) {
18         const expiresAt = moment().add(authResult.expiresIn,'second');
19
20         localStorage.setItem('id_token', authResult.idToken);
21         localStorage.setItem("expires_at", JSON.stringify(expiresAt.valueOf
22     }
```

```
23
24      logout() {
25          localStorage.removeItem("id_token");
26          localStorage.removeItem("expires_at");
27      }
28
29      public isLoggedIn() {
30          return moment().isBefore(this.getExpiration());
31      }
32
33      isLoggedOut() {
34          return !this.isLoggedIn();
35      }
36
37      getExpiration() {
38          const expiration = localStorage.getItem("expires_at");
39          const expiresAt = JSON.parse(expiration);
40          return moment(expiresAt);
41      }
42  }
43
44
```

Let's break down what is going on in this implementation, starting with the login method:

- We are receiving the result of the login call, containing the JWT and the `expiresIn` property, and we are passing it directly to the `setSession` method
- inside `setSession`, we are storing the JWT directly in Local Storage in the `id_token` key entry
- We are taking the current instant and the `expiresIn` property, and using it to calculate the expiration timestamp
- Then we are saving also the expiration timestamp as a numeric value in the `expires_at` Local Storage entry

Using Session Information on the client side

Now that we have all session information on the client side, we can use this information in the rest of the client application.

For example, the client application needs to know if the user is logged in or logged out, in order to decide if certain UI elements such as the Login / Logout menu buttons should be displayed or not.

This information is now available via the methods `isLoggedIn()`, `isLoggedOut()` and `getExpiration()`.

Sending The JWT to the server on each request

Now that we have the JWT saved in the user browser, let's keep tracking its journey through the network.

Let's see how we are going to use it to tell the Application server that a given HTTP request belongs to a given user, which is the whole point of the Authentication solution.

Here is what we need to do: we need with each HTTP request sent to the Application server, to somehow also append the JWT!

The application server is then going to validate the request and link it to a user, simply by inspecting the JWT, checking its signature and reading the user identifier from the payload.

To ensure that every request includes a JWT, we are going to use an Angular HTTP Interceptor.

How to build an Authentication HTTP Interceptor

Here is the code for an Angular Interceptor, that includes the JWT with each request sent to the application server:

```
1
2  @Injectable()
3  export class AuthInterceptor implements HttpInterceptor {
4
5      intercept(req: HttpRequest<any>,
6                next: HttpHandler): Observable<HttpEvent<any>> {
7
8          const idToken = localStorage.getItem("id_token");
9
10         if (idToken) {
11             const cloned = req.clone({
12                 headers: req.headers.set("Authorization",
13                     "Bearer " + idToken)
14             });
15
16             return next.handle(cloned);
17         }
18         else {
19             return next.handle(req);
20         }
21     }
22 }
23
```

Let's then break down how this code works line by line:

- we first start by retrieving the JWT string from Local Storage directly
- notice that we did not inject here the `AuthService`, as that would lead to a circular dependency error
- then we are going to check if the JWT is present

- if the JWT is not present, then the request goes through to the server unmodified
- if the JWT is present, then we will clone the HTTP headers, and add an extra `Authorization` header, which will contain the JWT

And with this in place, the JWT that was initially created on the Authentication server, is now being sent with each request to the Application server.

Let's then see how will the Application server use the JWT to identify the user.

Validating a JWT on the server side

In order to authenticate the request, we are going to have to extract the JWT from the `Authorization` header, and check the timestamp and the user identifier.

We don't want to apply this logic to all our backend routes because certain routes are publicly accessible to all users. For example, if we built our own login and signup routes, then those routes should be accessible by any user.

Also, we don't want to repeat the Authentication logic on a per route basis, so the best solution is to create an Express Authentication middleware and only apply it to certain routes.

Let's say that we have defined an express middleware called `checkIfAuthenticated`, this is a reusable function that contains the Authentication logic in only one place.

Here is how we can apply it to only certain routes:

```
1
2  import * as express from 'express';
3
4  const app: Application = express();
5
6  //... define checkIfAuthenticated middleware
7
8  // check if user authenticated only in certain routes
9  app.route('/api/lessons')
10     .get(checkIfAuthenticated, readAllLessons);
11
```

In this example, `readAllLessons` is an Express route that serves a JSON list of lessons if a GET request hits the `/api/lessons` Url.

We have made this route accessible only to authenticated users, by applying the `checkIfAuthenticated` middleware before the REST endpoint, meaning that the order of middleware functions is important.

The `checkIfAuthenticated` middleware will either report an error if no valid JWT is present, or allow the request to continue through the middleware chain.

The middleware needs to throw an error also in the case that a JWT is present, correctly signed but expired. Note that all this logic is the same in any application that uses JWT-based Authentication.

We could write this middleware ourselves using node-jsonwebtoken, but this logic is easy to get wrong so let's instead use a third-party library.

Configuring a JWT validation middleware using express-jwt

In order to create the `checkIfAuthenticated` middleware, we are going to be using the express-jwt library.

This library allows us to quickly create middleware functions for commonly used JWT-based authentication setups, so let's see how we would use it to validate JWTs like the ones that we created in the login service (signed using RS256).

Let's start by assuming that we had first installed the public signature validation key in the file system of the server. Here is how we could use it to validate JWTs:

```
1
2  const expressJwt = require('express-jwt');
3
4  const RSA_PUBLIC_KEY = fs.readFileSync('./demos/public.key');
5
6  const checkIfAuthenticated = expressJwt({
7      secret: RSA_PUBLIC_KEY
8  });
9
10 app.route('/api/lessons')
11     .get(checkIfAuthenticated, readAllLessons);
12
```

Let's now break down this code line by line:

- we started by reading the public key from the file system, which will be used to validate JWTs
- this key can only be used to validate existing JWTs, and not to create and sign new ones

- we passed the public key to `express-jwt`, and we got back a ready to use middleware function!

This middleware will throw an error if a correctly signed JWT is not present in the `Authorization` header. The middleware will also throw an error if the JWT is correctly signed, but it has already expired.

If we would like to change the default error handling behavior, and instead of throwing an error, for example, return a status code 401 and a JSON payload with a message, that is also possible.

But one of the main advantages of using RS256 signatures is that we don't have to install the public key locally in the application server, like we did in this example.

Imagine that the server had several running instances: replacing the public key everywhere at the same time would be problematic.

Leveraging RS256 Signatures

Instead of installing the public key on the Application server, it's much better to have the Authentication server *publish* the JWT-validating public key in a publicly accessible Url.

This give us a lot of benefits, such as for example simplified key rotation and revocation. If we need a new key pair, we just have to publish a new public key.

Typically during periodic key rotation, we will have the two keys published and active for a period of time larger than the session duration, in order not to interrupt user experience, while a revocation might be effective much faster.

There is no danger that the attacker could leverage the public key. The only thing that an attacker can do with the public key is to validate signatures of existing JWTs, which is of no use for the attacker.

There is no way that the attacker could use the public key to forge newly create JWTs, or somehow use the public key to guess the value of the private signing key.

The question now is, how to publish the public key?

JWKS (JSON Web Key Set) endpoints and key rotation

JWKS or JSON Web Key Set is a JSON-based standard for publishing public keys in a REST endpoint.

The output of this type of endpoint is a bit scary, but the good news is that we won't have to consume directly this format, as this will be consumed transparently by a library:


```
{
  "keys": [
    {
      "alg": "RS256",
      "kty": "RSA",
      "use": "sig",
      "x5c": [
        "MIIDJTCCAg2gAwIBAgIJUP6A\/iwWqvedMA0GCSqGSIb3DQEBCwUAMDAxLjAsBgNVE
      ],
      "n": "wUvZ-4dkT2nTfCDIwyH9K0tH4qYMGcW_KDYeh-TjBdASUS9cd741C0XMvmVSYGR
      "e": "AQAB",
      "kid": "QzY0NjREMjkyQTI4RTU2RkE4MUJBRDExNzY1MUY1N0I4QjFCODlBOQ",
      "x5t": "QzY0NjREMjkyQTI4RTU2RkE4MUJBRDExNzY1MUY1N0I4QjFCODlBOQ"
    }
  ]
```

```
}

```

A couple of details about this format: `kid` stands for Key Identifier, and the `x5c` property is the public key itself (its the x509 certificate chain).

Again, we won't have to write code to consume this format, but we do need to have an overview of what is going on in this REST endpoint: its simply publishing a public key.

Implementing JWKS key rotation using the `node-jwks-rsa` library

Since the public key format is standardized, what we need is a way of reading the key, and pass it to `express-jwt` so that it can be used instead of the public key that was read from the file system.

And that is exactly what the node-jwks-rsa library will allow us to do! Let's have a look at this library in action:

```

const jwksRsa = require('jwks-rsa');
const expressJwt = require('express-jwt');

const checkIfAuthenticated = expressJwt({
    secret: jwksRsa.expressJwtSecret({
        cache: true,
        rateLimit: true,
        jwksUri: "https://angularuniv-security-course.auth0.com/.well-known
    }),
    algorithms: ['RS256']
});

app.route('/api/lessons')
    .get(checkIfAuthenticated, readAllLessons);

```

This library will read the public key via the URL specified in property `jwksUri`, and use it to validate JWT signatures. All we have to do is configure the URL and if needed a couple of extra parameters.

Configuration options for consuming the JWKS endpoint

The parameter `cache` set to true is recommended, in order to prevent having to retrieve the public key each time. By default, a key will be kept for 10 hours before checking back if its still valid, and a maximum of 5 keys are cached at the same time.

The `rateLimit` property is also enabled, to make sure the library will not make more then 10 requests per minute to the server containing the public key.

This is to avoid a denial of service scenario, were by some reason (including an attack, but maybe a bug), the public server is constantly rotating the public key.

This would bring the Application server to a halt very quickly so its great to have built-in defenses against that! If you would like to change these default parameters, have a look at the library docs for further details.

And with this, we have completed the JWT journey through the network!

- We have created and signed a JWT in the Application server
- We have shown how the client can use the JWT and send it back to the server with each HTTP request

- we have shown how the Application server can validate the JWT, and link each request to a given user

And we have discussed the multiple design decisions involved in this roundtrip. Let's summarize what we have learned.

Summary and Conclusions

Delegating security features like Authentication and Authorization to a third-party JWT-based provider or product is now more feasible than ever, but this does not mean that security can be added transparently to an application.

Even if we choose a third party authentication provider or an enterprise single sign-on solution, we will still have to know how JWTs work at least to some detail, if nothing else to understand the documentation of the products and libraries that we will need to choose from.

We will still have to take a lot of security design decisions ourselves, choose libraries and products, choose critical configuration options such as JWT signature types, setup hosted login pages if applicable and put in place some very critical security-related code that is easy to get wrong.