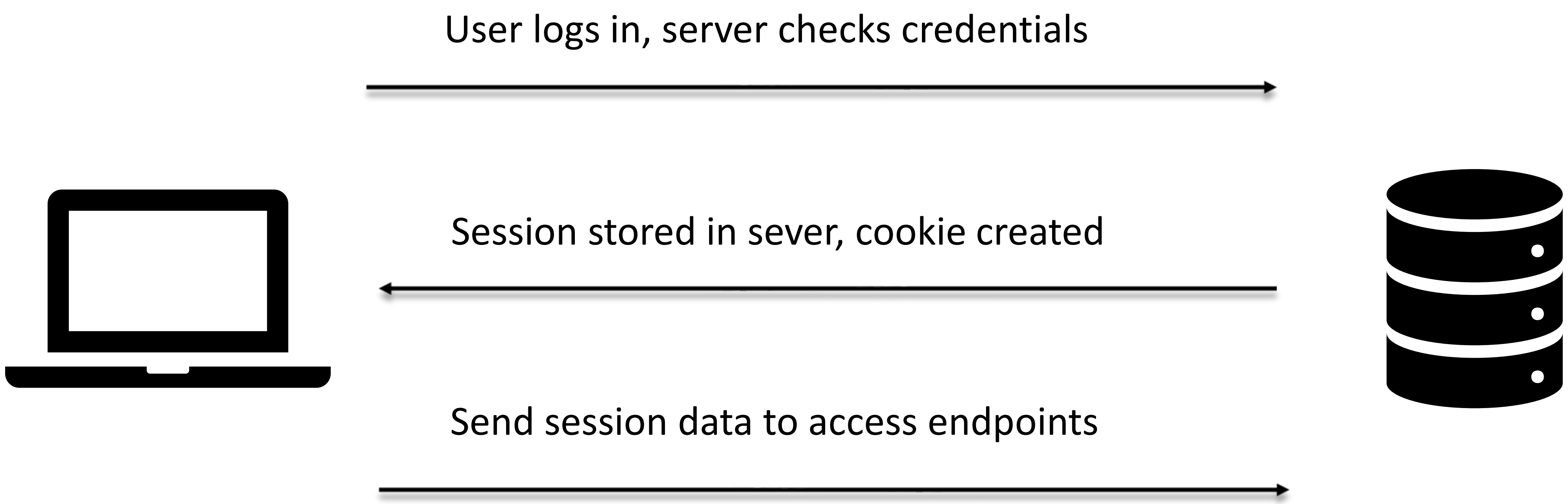
Authentication & Authorization

Traditional Authentication System



User logs in, server checks credentials

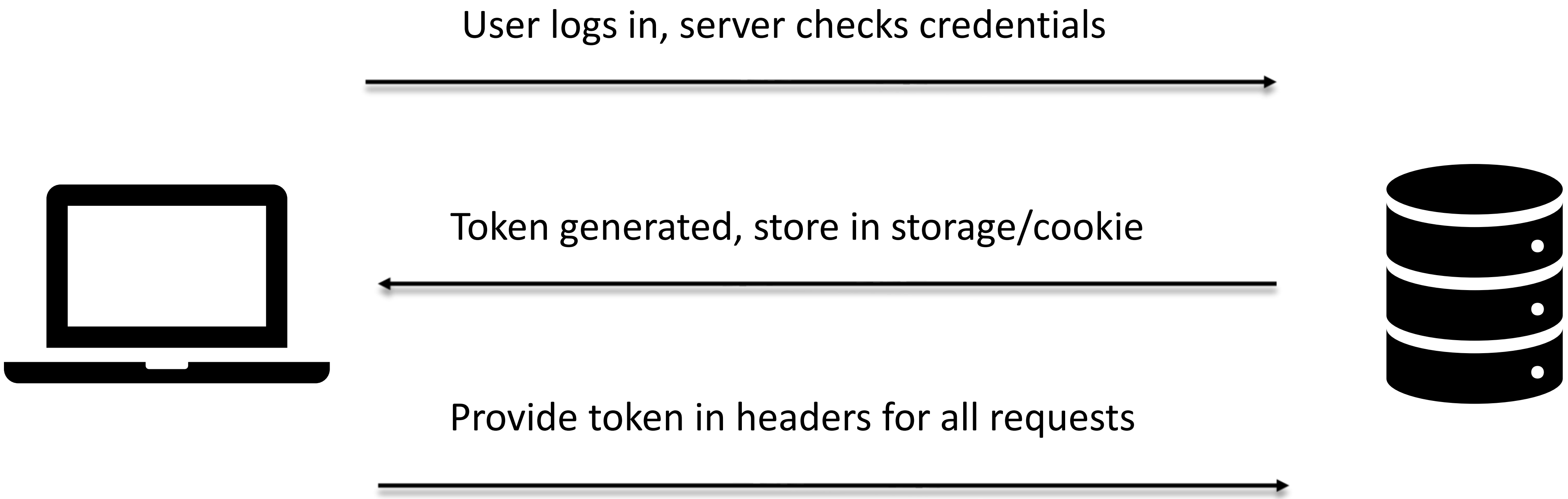
Session stored in sever, cookie created

Send session data to access endpoints

Issues with Traditional Systems

- Sessions: Record needs to be stored on server.
- Scalability: With sessions in memory, load increases drastically in a distributed system.
- CORS: When using multiple devices grabbing data via AJAX requests, may run into forbidden requests.
- CRSF: Riding session data to send requests to server from a browser that is trusted via session.

Token-Based Authentication Systems



User logs in, server checks credentials

Token generated, store in storage/cookie

Provide token in headers for all requests

Token-based Authentication System

- Stateless: self containded.
- Scalability: no need to store session in memory
- CSRF: no session being used
- Digitally-signed
- Mobile-ready
- Decoupled

What is JSON Web Token?

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.
- This information can be verified and trusted because it is digitally signed.
- **Compact**: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.
  - Simply a string in the format of **header.payload.signature**
- **Self-contained**: The payload contains all the required information about the user, avoiding the need to query the database more than once.

JSON Web Token Structure

- JSON Web Tokens consist of three parts separated by dots (.), which are:
  - header
  - payload
  - signature

- Therefore, a JWT typically looks like the following:
  - xxxxx.yyyyy.zzzzz
  - eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

JWT Header

- The header *typically* consists of two parts: the type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA256 or RSA.
- For example:

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

- Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

HMAC SHA256 vs RSA SHA256 hashing algorithms

- HMAC SHA256: Symmetric Key cryptography, single shared private key. Faster, good between trusted parties.
  - A combination of a hashing function and one (secret) key that is shared between the two parties used to generate the hash that will serve as the signature.

- RSA SHA256: Asymmetric Key cryptography, public/private keys. Slower, good between untrusted parties.
  - The identity provider has a private (secret) key used to generate the signature, and the consumer of the JWT gets a public key to validate the signature.

JWT Payload

- The second part of the token is the payload, which contains the claims.
- **Claims** are statements about an entity (typically, the user) and additional metadata. There are three types of claims:
  - *reserved*
    - The JWT specification defines seven reserved claims that are not required, but are recommended to allow interoperability with third-party applications.
  - *Public*
    - These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
  - *Private*
    - These are the custom claims created to share information between parties that agree on using them.

JWT Payload

- For example:

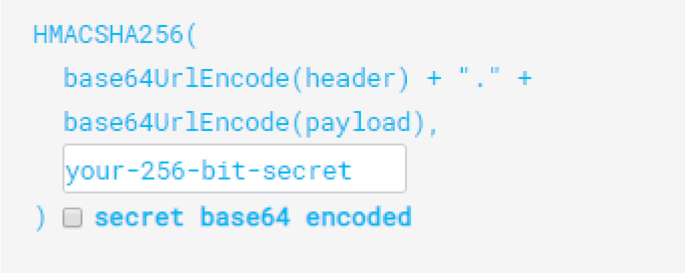
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

- The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I
kpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6
yJV_adQssw5c

JWT Signature

- To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
- The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.
- For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

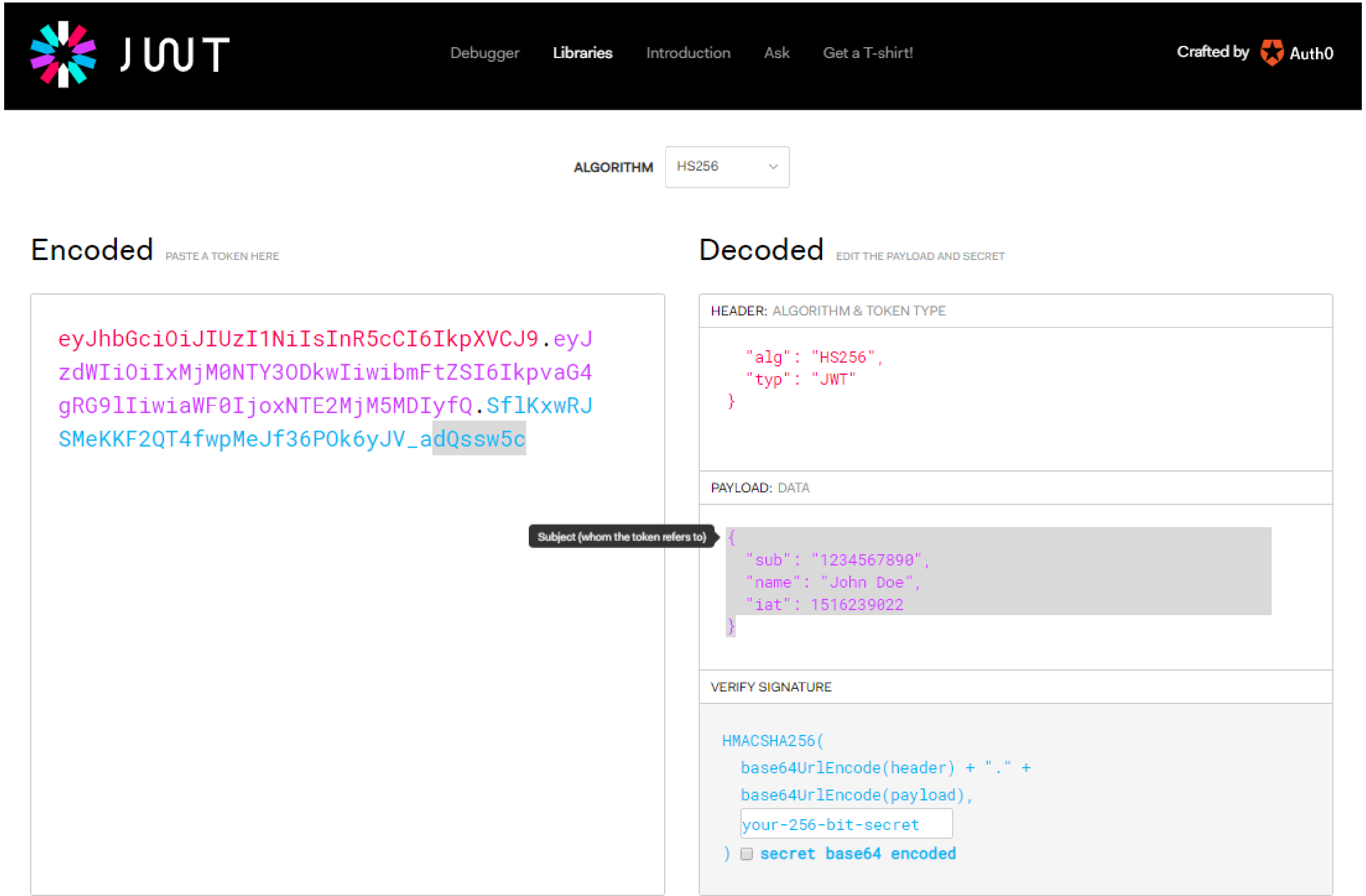


```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

jwt.io

- JWT.IO allows you to decode, verify and generate JWT.



ALGORITHM HS256

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ
SMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

Subject (whom the token refers to)

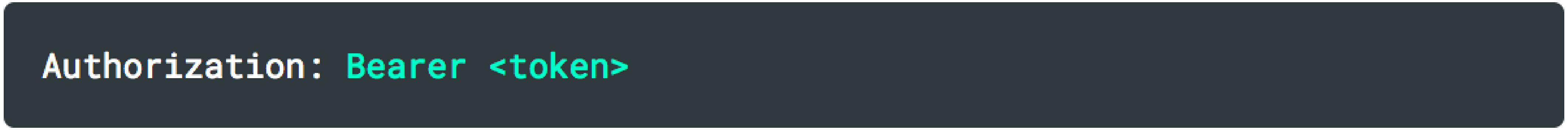
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

How does JWT work?

- In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used).
- Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema. The content of the header should look like the following:



```
Authorization: Bearer <token>
```

Verifying a JWT

- Use the secret(only application knows) to generate a signature with the header and payload from in the incoming JWT
- If the generated signature matches the incoming JWT signature, the JWT is considered valid.

- Now let's pretend that you're a hacker trying to issue a fake token. You can easily generate the header and payload, but without knowing the key, there is no way to generate a valid signature. If you try to tamper with the existing payload of a valid JWT, the signatures will no longer match.

Implement JWT on Server-Side

- Add `/signin` feature to generate JWT token

```
exports.signin = async(req, res, next) => {
    try {
        const user = await User.findOne({ username: req.body.username });
        if (user) {
            const isValid = await bcrypt.compare(req.body.password, user.password);
            if (isValid) {
                const token = jwt.sign({ data: req.body.username }, config.jwtKey, {
                    expiresIn: config.jwtExpirySeconds
                });
                res.status(200).send(new ApiResponse(200, 'success', { token: token, expiresIn: config.jwtExpirySeconds, user: user }));
            } else {
                res.status(401).send(new ApiResponse(401, 'error', { err: 'username or password not exist' }));
            }

        } else {
            res.status(401).send(new ApiResponse(401, 'error', { err: 'username or password not exist' }));
        }
    } catch (err) {
        res.status(500).send(new ApiResponse(500, 'error', err));
    }
}
```

Protect Routes on Server Side

```
exports.verifyToken = (req, res, next) => {
    const authHeader = req.headers['authorization'];
    if (!authHeader) {
        return res.status(403).send(new ApiResponse(403, 'error', { err: 'No Token Provided!' }));
    }
    const token = authHeader.split(' ')[1];

    jwt.verify(token, config.jwtKey, (err, decoded) => {
        if (err) {
            return res.status(401).send(new ApiResponse(401, 'error', { err: 'Unauthorized!' }));
        }
        next();
    });
}
```

app.js

```
app.use(authRoutes);
app.use(authMiddleware.verifyToken);
app.use(userRoutes);
```

Implement Login on Front-Side

```
login() {
    const val = this.form.value;
    if (val.username && val.password) {
      this.authService.login(val)
        .subscribe(
          (response) => {
            this.setSession(response.result);
            this.router.navigateByUrl('/list-user');
          }
        );
}}
```

```
@Injectable({
  providedIn:'root'
})
export class AuthService {
  constructor(private http: HttpClient) {
  }
  baseUrl: string = SERVER_URL;

  login(val: { username: string, password: string }): Observable<ApiResponse> {
    return this.http.post<ApiResponse>(this.baseUrl + 'signin', val);
  }
```

Angular Interceptor

- **intercept and modify the application’s http requests globally** before they are sent to the server.
- Can be used for:
  - configure *authentication tokens*
  - add *logs* of the requests
  - add *custom headers*

- Generate interceptor:
  - ng generate interceptor <name> [options]
  - ng g interceptor <name> [options]

Implementing an Interceptor

- The *intercept* method transforms each request into *Observables*, which later are going to be resolved by calling *next.handle()*.

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

  constructor() { }

  intercept(req: HttpRequest<any>,
    next: HttpHandler): Observable<HttpEvent<any>> {


      return next.handle(req);

  }
}
```

Providing the Interceptor

- Interceptors are dependencies of the `HttpClient`, you must add them to providers in the same injector (or parent) that provides the `HttpClient`.
- *multi: true* option provided tells Angular that you are providing **multiple interceptors**

```
@NgModule({
  …
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: AuthInterceptor,
    multi: true
  }],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Handling Authrization

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const idToken = localStorage.getItem("id_token");
    if (idToken)
      req = req.clone({headers: req.headers.set("Authorization","Bearer " + idToken)});

    return next.handle(req).pipe(
      catchError(error => {
        // Checking if it is an Authentication Error (401)
        if (error.status === 401) {
          alert('Access Denied');
          // <Log the user out of your application code>
          this.router.navigate(['login']);
          return throwError(error);
        }
        // If it is not an authentication error, just throw it
        return throwError(error);
      })
    );
  }
```

Reference

- https://blog.angular-university.io/angular-jwt-authentication/
- https://www.sohamkamani.com/blog/javascript/2019-03-29-node-jwt-authentication/
- https://itnext.io/understanding-angular-interceptors-405b84d7ad69