Authentication & Session

Course Code: CSC 4182 Course Title: Advanced Programming In Web Technologies

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**5**	**Week No:**	**07**	**Semester:**	
Lecturer:	*Sazzad Hossain; sazzad@aiub.edu*				

Lecture Outline

- ✓ Bcrypt
- ✓ Session
- ✓ Gaurds
- ✓ Session Gaurds

Bcrypt

- bcrypt is a password-hashing function that is widely used for securely storing passwords.
- Bcrypt is a popular choice for password hashing because it incorporates a salt (a random value) into the hashing process.
- The salt adds additional randomness and makes each hashed password unique, even if two users have the same password.
- It is recommended to use bcrypt in nestjs for password hashing.

Using Bcrypt

install required packages:

```
$ npm i bcrypt
$ npm i -D @types/bcrypt
```

Import bcrypt library

```
import * as bcrypt from 'bcrypt';
```

To generate a salt, use the genSalt function:

```
const salt = await bcrypt.genSalt();
```

To perform hash

```
const hassedpassed = await bcrypt.hash(password, salt);
```

To compare/check a password:

```
const isMatch = await bcrypt.compare(password, dbpassword);
```

Session

HTTP sessions provide a way to store information about the user across multiple requests

Sessions provide a way to maintain statefulness in stateless HTTP-based protocols.

install the required package

```
$ npm i express-session
$ npm i -D @types/express-session
```

Session

Once the installation is complete, apply the express-session middleware as global middleware (for example, in your main.ts file).

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as session from 'express-session';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(
    session({
    secret: 'my-secret',
    resave: false,
    saveUninitialized: false,
   cookie:{
    maxAge: 300000
    }
   }),
   );
  await app.listen(3000);
}
bootstrap();
```

Session

- The **secret** is used to sign the session ID cookie.
- secrets is provided to verify the signature in requests.
- The secret would best be a random set of characters.
- Enabling the **resave** option forces the session to be saved back to the session store.
- Enabling the **saveUninitialized** option Forces a session that is "uninitialized" to be saved to the store.
- A session is uninitialized when it is **new** but **not modified.**
- **False** is set for implementing login sessions, reducing server storage usage

Session

- **@Session()** decorator is used to access and modify the session information
- Below is an example to access and modify session data

```
signIn(@Session() session){
session.email = "myemail@email.com";
console.log(session.email);
}
```

- To destroy session, destroy() method is used.

```
session.destroy()
```

Gaurds

- Guards are a powerful mechanism for handling authorization and access control in application.
- Guards determine whether a given **request** will be handled by the route handler or not, depending on present at run-time.
- This is often referred to as **authorization**. Authorization has typically been handled by **middleware** in traditional Express applications.
- Guards can be applied at different levels in the application's routing pipeline to protect routes, controllers, or individual handler methods.

Gaurds


Client Side
HTTP Request
Guard
Route Handler
@RequestMapping

Gaurds

Guard Class: Guards are implemented as classes that implement the **CanActivate** interface. The guard class contains the logic that determines whether the request should be allowed or denied.

Injectable: Guards can be annotated with the @Injectable() decorator to make them injectable and allow dependencies to be injected.

Guard Logic: The guard class defines a method called **canActivate**(). This method contains the authorization logic that decides whether to allow or deny access to the requested resource.

Returning a Value: The canActivate() method must return a boolean value or a promise/observable that resolves to a boolean. If it returns **true**, the request is allowed to **proceed** otherwise the request is **blocked**.

Session Guards

Following is an example of session guard located in session.gaurd.ts

```
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';

@Injectable()
export class SessionGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean {
    const request = context.switchToHttp().getRequest();
  return (request.session.email !== undefined);
  }
}
```

Session Guards

To use guards, @UseGaurds has been used. Following is an example where the session guards has been used to protect the update request.

```
@Put('/updateadmin/')
@UseGuards(SessionGuard)
@UsePipes(new ValidationPipe())
updateAdmin(@Session() session,@Body('name') name: string): any {
  console.log(session.email);
  return this.adminService.updateUser(name, session.email);
}
```

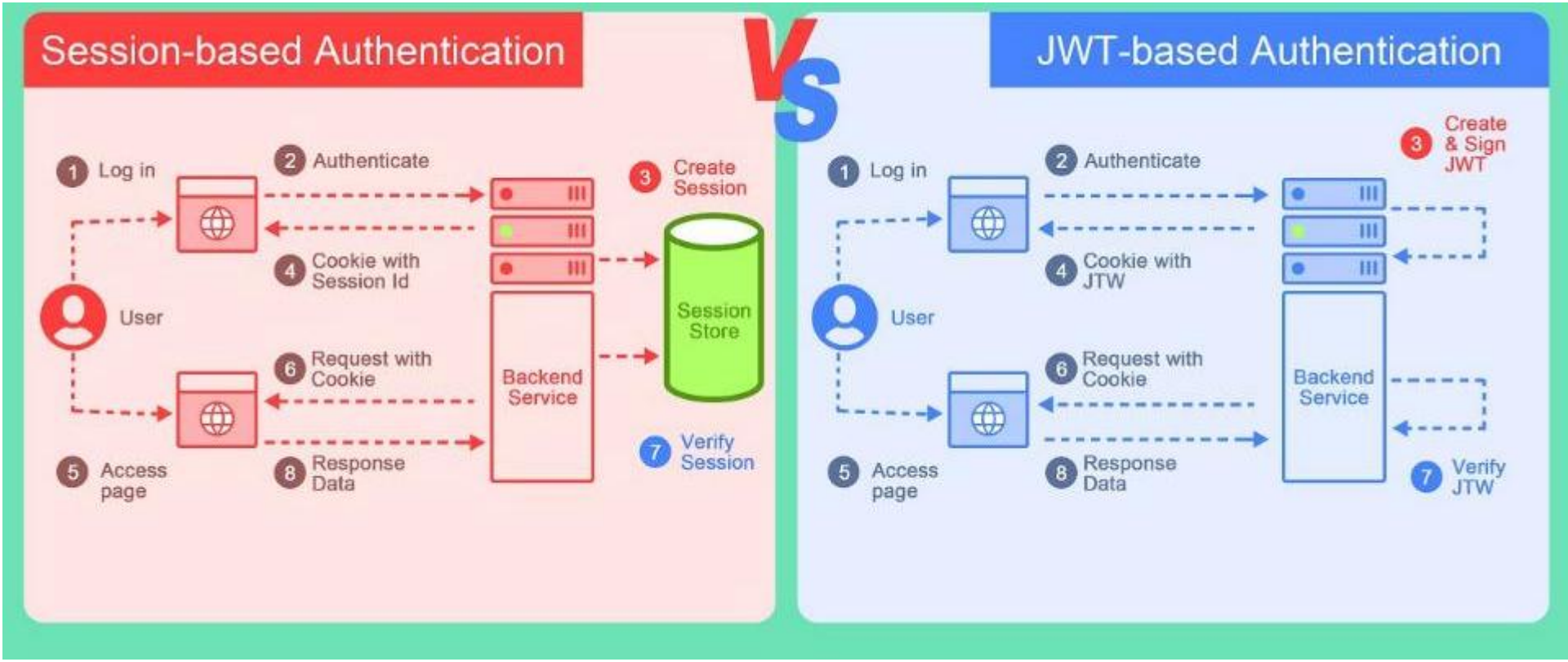
JWT

- JSON Web Token (JWT) is a compact, URL-safe means of representing claims between two parties.
- It is defined by the RFC 7519 standard.

RFC 7519 standard is

- JSON object that is used as the payload of a JSON Web Signature (JWS) structure or
- as the plaintext of a JSON Web Encryption (JWE) structure
- Digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

Session Vs JWT Authentication



JWT Example

Example has been given following;

https://docs.nestjs.com/security/authentication

JWT Authentication

- Upon successful authentication, the server generates a JWT containing relevant claims (e.g., user ID, issuance time, expiration time).

```
{
  "email": "abc@email.com",
  “id": “12",
  "iat": 1709667424,
  "exp": 1709669224
}
```

- The client includes the JWT in the Authorization header of subsequent requests.

  Authorization: Bearer **eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9…**

- The server validates the JWT's signature using the shared secret key.
- The server checks the token's claims, expiration, and other relevant information.

CORS

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that **allows a server to indicate any origins** (domain, scheme, or port) other than its own from which a browser should permit loading resources.
example : the front-end JavaScript code served **from https://domain-a.com** uses XMLHttpRequest to make a request for **https://domain-b.com/data.json.**
To enable CORS in Nestjs, go to **main.ts** and add following syntax in bootstrap function

```
app.enableCors();
```

References

1. **W3Schools Online Web Tutorials, URL: http://www.w3schools.com**
2. **Node.js, URL: https://nodejs.org/en/**
3. **Next.js, URL: https://nextjs.org/**
4. **TypeScript URL: https://www.typescriptlang.org/**
5. **MDN Web Docs URL: https://developer.mozilla.org/**


AMERICAN INTERNATIONAL UNIVERSITY
PRAESIDIUM
DISCIPLINA
CIVITATIS
1994
BANGLADESH

Thank You!