Secure APIs using NestJS

Exposing a secure API is paramount in today's digital landscape due to the interconnected nature of applications and systems. Security breaches can lead to catastrophic consequences, ranging from data theft and financial losses to reputational damage and legal liabilities. A secure API serves as the gateway to valuable resources and sensitive data, making it a prime target for malicious actors. Therefore, ensuring robust security measures is essential to mitigate risks and protect against various threats.

One of the primary reasons for exposing a secure API is to safeguard sensitive information. APIs often handle critical data, including user credentials, financial transactions, and personal details. Without adequate protection, this data is vulnerable to interception, tampering, or unauthorized access. By implementing security protocols such as encryption, authentication, and access controls, organizations can prevent data breaches and maintain the confidentiality, integrity, and availability of their information assets.

Furthermore, secure APIs play a pivotal role in maintaining trust and compliance. In an era of heightened privacy concerns and stringent regulations like GDPR and CCPA, users expect their data to be handled responsibly and ethically. Exposing a secure API demonstrates a commitment to protecting user privacy and adhering to regulatory requirements. It helps organizations build credibility, foster customer loyalty, and avoid costly penalties associated with non-compliance.

Moreover, securing an API is essential for defending against a myriad of cyber threats, including unauthorized access attempts, injection attacks, and denial-of-service (DoS) attacks. By implementing robust authentication mechanisms, rate limiting, and input validation, organizations can thwart malicious activities and ensure the reliability and availability of their services. Additionally, ongoing monitoring, threat detection, and incident response capabilities are crucial for identifying and mitigating security incidents promptly.

Exposing a secure API using guards in NestJS is a fundamental aspect of building robust and protected web applications. NestJS, a progressive Node.js framework, provides a comprehensive and flexible approach to implementing security measures through guards. Guards act as middleware components that intercept incoming

requests to routes or controllers and determine whether the request should be allowed to proceed based on specified criteria such as authentication, authorization, or custom business logic.

One of the primary advantages of using guards in NestJS is their versatility and extensibility. Developers can create custom guard classes tailored to their application's specific security requirements, allowing for fine-grained control over access to resources. Whether it's enforcing authentication through token validation, restricting access based on user roles or permissions, or implementing rate limiting to prevent abuse, guards provide a powerful mechanism for enforcing security policies at various levels of the application.

To expose a secure API using guards in NestJS, developers typically follow a structured approach. First, they define guard classes that implement the `CanActivate` interface, which requires the implementation of a `canActivate()` method. Within this method, developers write the logic to determine whether the request should be allowed to proceed or denied based on the defined criteria. This could involve verifying authentication tokens, checking user roles or permissions, or any other conditions relevant to the application's security posture.

```
async canActivate(context: ExecutionContext): Promise<boolean> {
  const request = context.switchToHttp().getRequest();
  const token : string  = this.extractTokenFromHeader(request);
  if (!token) {
    throw new UnauthorizedException();
  }

  try {
    const result : DecodedIdToken  = await admin.auth().verifyIdToken(token);

    const user : User  = await this.usersService.getUserDetailsForLogin(result.email);
    if (!user) {
      return false;
    }

    this.eventEmitter.emit(UserEventsEnum.updateLastActive, user);
    this.eventEmitter.emit(
      SplunkEventsEnum.createEvent,
      values: `User with email ${user.email} last active updated`,
      SplunkIndexes.updateLastActive,
      user,
    );

    const roles : string[]  = this.reflector.get<string[]>( metadataKey: 'roles', context.getHandler());

    // 💡 We're assigning the payload to the request object here
    // so that we can access it in our route handlers
    request['user'] = user;

    // If there are no roles, and we need just authentication then it's a GO
    if (!roles) {
      return true;
    }

    return this.matchRoles(roles, user.role);
  } catch (e) {
    console.log('e ------------------>> ', e);
    throw new UnauthorizedException();
  }
  return true;
}
```

NestJS Passport also provides a set of built-in guards. One of the most commonly used guards from `@nestjs/passport` is the `AuthGuard`. This guard is designed to protect routes by requiring authentication before allowing access to the associated resources. It is typically used in conjunction with authentication strategies provided by the `passport` module, such as `JwtStrategy` for JWT-based authentication or `OAuth2Strategy` for OAuth-based authentication.

```
2 usages
@Injectable()
export class JwtAuthGuard extends AuthGuard( type: 'jwt') {}
```

The `AuthGuard` works by intercepting incoming requests and verifying the presence and validity of authentication credentials, such as JWT tokens or OAuth tokens. If the credentials are valid, the guard allows the request to proceed to the route handler; otherwise, it denies access and returns an appropriate error response.

Once the guard classes are defined, developers apply them to the routes or controllers they wish to protect using the `@UseGuards()` decorator provided by NestJS. This declarative approach makes it straightforward to apply guards at both the controller and route handler levels, providing a high degree of flexibility in defining access control policies. When a request is made to a guarded route, NestJS automatically invokes the appropriate guards, executing their logic to determine whether the request should be allowed or rejected. If a guard denies access, NestJS handles the rejection gracefully, typically returning an appropriate HTTP status code and error message to the client.

```
no usages
@UseGuards(JwtAuthGuard)
@Get()
findAll(): Promise<TennisPlayer[]> {
  return this.tennisPlayerService.getAll();
}
```