# 인증

인증은 대부분의 애플리케이션에서 필수적인 부분입니다. 인증을 처리하기 위한 다양한 접근 방식과 전략이 있습니다. 프로젝트마다 특정 애플리케이션 요구 사항에 따라 접근 방식이 달라집니다. 이 장에서는 다양한 요구 사항에 맞게 조정할 수 있는 인증에 대한 몇 가지 접근 방식을 소개합니다.

요구 사항을 구체화해 봅시다. 이 사용 사례에서 클라이언트는 사용자 이름과 비밀번호로 인증하는 것으로 시작합니다. 인증이 완료되면 서버는 인증을 증명하기 위해 후속 요청 시 인증 헤더에 베어러 토큰으로 전송할 수 있는 JWT를 발급합니다. 또한 유효한 JWT가 포함된 요청만 액세스할 수 있는 보호된 경로를 생성합니다.

첫 번째 요구 사항인 사용자 인증부터 시작하겠습니다. 그런 다음 JWT를 발행하여 이를 확장합니다. 마지막으로 요청에 대해 유효한 JWT를 검사하는 보호된 경로를 생성합니다.

## 인증 모듈 만들기

먼저 AuthModule을 생성하고 그 안에 AuthService와 AuthController를 생성하겠습니다. AuthService를 사용하여 인증 로직을 구현하고 AuthController를 사용하여 인증 엔드포인트를 노출할 것입니다.

```
nest g 모듈 인증
nest g 컨트롤러 인증
nest g 서비스 인증
```

AuthService를 구현하면서 사용자 작업을 UsersService에 캡슐화하는 것이 유용하다는 것을 알게 될 것이므로 이제 해당 모듈과 서비스를 생성해 보겠습니다:

```
nest g 모듈 사용자
nest g 서비스 사용자
```

생성된 파일의 기본 내용을 아래와 같이 바꿉니다. 샘플 앱의 경우, 사용자 서비스는 단순히 하드코딩된 인메모리 사용자 목록과 사용자 이름으로 사용자를 검색하는 찾기 메서드를 유지 관리합니다. 실제 앱에서는 선택한 라이브러리(예: TypeORM, Sequelize, 몽구스 등)를 사용하여 사용자 모델과 지속성 레이어를 빌드할 수 있습니다.

```
@@파일명(사용자/사용자.서비스)
'@nestjs/common'에서 { Injectable }을 임포트합니다;

// 사용자 엔티티 내보내기 유형 User = any를 나타내는 실제 클래스/인터페이스여야 합니
다;

@Injectable()
export 클래스 UsersService {
  private 읽기 전용 사용자 = [
```

```
      {
        userId: 1, 사용자명:
        'john',
        비밀번호: 'changeme',
      },
      {
        userId: 2, 사용자 이
        름: 'maria', 비밀번호
        : 'guess',
      },
    ];

    async findOne(username: 문자열): Promise<사용자 | 정의되지 않음> {
      return this.users.find(사용자 => 사용자.사용자이름 === 사용자이름);
    }
  }
  @@switch
  '@nestjs/common'에서 { Injectable }을 임포트합니다;

  @Injectable()
  내보내기 클래스 UsersService {
    constructor() {
      this.users = [
        {
          userId: 1, 사용자명:
          'john',
          비밀번호: 'changeme',
        },
        {
          userId: 2, 사용자 이
          름: 'maria', 비밀번호
          : 'guess',
        },
      ];
    }

    async findOne(username) {
      반환 이.사용자.찾기(사용자 => 사용자.사용자 이름 === 사용자 이름);
    }
  }
```

`UsersModule에서` 필요한 유일한 변경 사항은 UsersService를

`모듈` 데코레이터를 추가하여 이 모듈 외부에서 볼 수 있도록 합니다(곧 `AuthService에서` 사용하게 될 것입니다).

```
@@파일명(사용자/사용자.모듈)
'@nestjs/common'에서 { Module }을 가져오고,
'./users.service'에서 { UsersService }를 가져옵니
다;

모듈({
  제공자: [UsersService], 내보내기:
  [UsersService],
```

```
})
사용자 모듈 클래스 {} @@스위치 내보내기
'@nestjs/common'에서 { Module }을 가져오고,
'./users.service'에서 { UsersService }를 가져옵니
다;

모듈({
  제공자: [UsersService], 내보내기:
  [UsersService],
})
사용자 모듈 클래스 {} 내보내기
```

"로그인" 엔드포인트 구현하기

AuthService는 사용자를 검색하고 비밀번호를 확인하는 작업을 수행합니다. 이를 위해 `signIn()` 메서드를 생성합니다. 아래 코드에서는 편리한 ES6 스프레드 연산자를 사용하여 사용자 객체에서 비밀번호 속성을 제거한 후 반환합니다. 이는 사용자 객체를 반환할 때 비밀번호나 기타 보안 키와 같은 민감한 필드를 노출하지 않으려는 일반적인 관행입니다.

```
@@파일명(auth/auth.service)
'@nestjs/common'에서 { Injectable, UnauthorizedException }을 임포트하고,
'../users/users.service'에서 { UsersService }를 임포트합니다;

@Injectable()
내보내기 클래스 AuthService {
  constructor(private usersService: UsersService) {}

  async signIn(사용자 이름: 문자열, 패스: 문자열): Promise<any> {
    const user = await this.usersService.findOne(username); if
    (user?.password !== pass) {
      새로운 UnauthorizedException()을 던집니다;
    }
    const { password, ...result } = 사용자;
    // TODO: JWT를 생성하여 여기에 반환합니다.
    // 대신 사용자 객체 반환 결과를 반
    환합니다;
  }
}
@@switch
'@nestjs/common'에서 { Injectable, Dependencies, UnauthorizedException }을 임포
트합니다;
'../users/users.service'에서 { UsersService }를 가져옵니다;

주입 가능()
@Dependencies(UsersService) 내
보내기 클래스 AuthService {
  constructor(usersService) {
    this.usersService = usersService;
  }

  async signIn(username: 문자열, pass: 문자열) {
```

```
      const user = await this.usersService.findOne(username);
      if (user?.password !== pass) {
        새로운 UnauthorizedException()을 던집니다;
      }
      const { password, ...result } = 사용자;
      // TODO: JWT를 생성하여 여기에 반환합니다.
      // 대신 사용자 객체 반환 결과를 반
      환합니다;
    }
  }
```

> 경고 경고 물론 실제 애플리케이션에서는 비밀번호를 일반 텍스트로 저장하지 않습니다. 대신 솔트 처리된 단방향 해시 알고리즘이 포함된 bcrypt와 같은 라이브러리를 사용할 것입니다. 이 접근 방식을 사용하면 해시된 비밀번호만 저장한 다음 저장된 비밀번호를 들어오는 비밀번호의 해시된 버전과 비교하므로 사용자 비밀번호를 일반 텍스트로 저장하거나 노출하지 않습니다. 샘플 앱을 단순하게 유지하기 위해 이러한 절대적인 의무를 위반하고 일반 텍스트를 사용했습니다. 실제 앱에서는 이렇게 하지 마세요!

이제 `AuthModule을` 업데이트하여 `UsersModule을` 가져옵니다.

```
@@파일명(auth/auth.module)
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니
다;
'./auth.controller'에서 { AuthController }를 임포트하고,
'../users/users.module'에서 { UsersModule }을 임포트합니다;

모듈({
  임포트: [UsersModule], 공급자:
  [AuthService], 컨트롤러:
  [AuthController],
})
내보내기 클래스 AuthModule {}
@@switch
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니
다;
'./auth.controller'에서 { AuthController }를 임포트하고,
'../users/users.module'에서 { UsersModule }을 임포트합니다;

모듈({
  임포트: [UsersModule], 공급자:
  [AuthService], 컨트롤러:
  [AuthController],
})
내보내기 클래스 AuthModule {}
```

이제 `AuthController`를 열고 `signIn()` 메서드를 추가해 보겠습니다. 이 메서드는 클라이언트에서 사용자를 인증하기 위해 호출됩니다. 이 메서드는 요청 본문에서 사용자 이름과 비밀번호를 받고, 사용자가 인증되면 JWT 토큰을 반환합니다.

```
@@파일명(auth/auth.controller)
'@nestjs/common'에서 { Body, Controller, Post, HttpCode, HttpStatus }를
임포트합니다;
'./auth.service'에서 { AuthService }를 가져옵니다;

컨트롤러('auth')
내보내기 클래스 AuthController {
  constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @Post('login')
  signIn(@Body() signInDto: Record<string, any>) {
    return this.authService.signIn(signInDto.username,
signInDto.password);
  }
}
```

> 정보 힌트 이상적으로는 `Record<string, any>` 유형을 사용하는 대신 DTO 클래스를 사용하여 요청 본문의 모양을 정의하는 것이 좋습니다. 자세한 내용은 유효성 검사 챕터를 참조하세요.

### JWT 토큰

이제 인증 시스템의 JWT 부분으로 넘어갈 준비가 되었습니다. 요구 사항을 검토하고 구체화해 보겠습니다:

- 사용자가 사용자 이름/비밀번호로 인증할 수 있도록 허용하여 보호된 API 엔드포인트에 대한 후속 호출에서 사용할 수 있도록 JWT를 반환합니다. 이 요구 사항을 충족하기 위한 작업은 순조롭게 진행 중입니다. 이를 완료하려면 JWT를 발급하는 코드를 작성해야 합니다.
- 무기명 토큰으로 유효한 JWT의 존재를 기반으로 보호되는 API 경로 만들기 JWT 요구 사항을 지원하

```
npm install --save @nestjs/jwt
```

> 정보 힌트 `@nestjs/jwt` 패키지(자세한 내용은 여기를 참조하세요)는 JWT 조작에 도움이 되는 유틸리티 패키지입니다. 여기에는 JWT 토큰 생성 및 확인이 포함됩니다.

서비스를 깔끔하게 모듈화하기 위해 `authService에서` JWT 생성을 처리합니다. `auth` 폴더에서 `auth.service.ts` 파일을 열고 `JwtService를` 삽입한 다음 `로그인` 메서드를 업데이트하여 아래와 같이

JWT 토큰을 생성합니다:

```
@@파일명(auth/auth.service)
'@nestjs/common'에서 { Injectable, UnauthorizedException }을 임포트하고,
'../users/users.service'에서 { UsersService }를 임포트합니다;
'@nestjs/jwt'에서 { JwtService }를 가져옵니다;

@Injectable()
내보내기 클래스 AuthService {
```

```
    생성자(
      비공개 usersService: UsersService, 비
      공개 jwtService: JwtService
    ) {}

    async signIn(username, pass) {
      const user = await this.usersService.findOne(username);
      if (user?.password !== pass) {
        새로운 UnauthorizedException()을 던집니다;
      }
      const payload = { sub: user.userId, username: user.username };
      return {
        access_token: await this.jwtService.signAsync(payload),
      };
    }
  }
  @@switch
  '@nestjs/common'에서 { Injectable, Dependencies, UnauthorizedException }을 임포트
  합니다;
  '../users/users.service'에서 { UsersService }를 임포트하고,
  '@nestjs/jwt'에서 { JwtService }를 임포트합니다;

  @Dependencies(UsersService, JwtService)
  @Injectable()
  export class AuthService {
    constructor(usersService, jwtService) {
      this.usersService = usersService;
      this.jwtService = jwtService;
    }

    async signIn(username, pass) {
      const user = await this.usersService.findOne(username);
      if (user?.password !== pass) {
        새로운 UnauthorizedException()을 던집니다;
      }
      const payload = { username: user.username, sub: user.userId };
      return {
        access_token: await this.jwtService.signAsync(payload),
      };
    }
  }
```

사용자 객체 프로퍼티의 하위 집합에서 JWT를 생성하기 위해 `signAsync()` 함수를 제공하는 `@nestjs/jwt` 라이브러리를 사용하고 있으며, 이 함수는 단일 `access_token` 프로퍼티가 있는 간단한 객체로 반환합니다. 참고: JWT 표준과 일관성을 유지하기 위해 `sub`라는 속성 이름을 선택하여 `userId` 값을 보유합니다. 인증 서비스에 JwtService 공급자를 삽입하는 것을 잊지 마세요.

이제 새 종속성을 가져오고 `JwtModule`을 구성하기 위해 `AuthModule`을 업데이트해야 합니다. 먼저 `auth` 폴더에 `constants.ts`를 생성하고 다음 코드를 추가합니다:

```
@@파일명(auth/constants) export
const jwtConstants = {
  비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게
보관하세요.',
};
@@switch
export const jwtConstants = {
  비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게
보관하세요.',
};
```

이 키를 사용하여 JWT 서명 및 확인 단계 간에 키를 공유합니다.

> 경고 경고 이 키를 공개적으로 노출하지 마세요. 여기서는 코드가 수행하는 작업을 명확히 하기 위해 공개했지만, 프로덕션 시스템에서는 시크릿 볼트, 환경 변수 또는 구성 서비스 등의 적절한 조치를 사용하여 이 키를 보호해야 합니다.

이제 인증 폴더에서 `auth.module.`ts를 열고 다음과 같이 업데이트합니다:

```
@@파일명(auth/auth.module)
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니
다;
'../users/users.module'에서 { UsersModule }을 임포트하고,
'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { AuthController } from './auth.controller';
import { jwtConstants } './constants'에서 임포트합니다;

모듈({ import:
  [
    UsersModule,
    JwtModule.register({
      global: true,
      비밀: jwtConstants.secret, signOptions:
      { expiresIn: '60s' },
    }),
  ],
  제공자: [AuthService], 컨트롤러:
  [AuthController], 내보내기:
  [AuthService],
})
내보내기 클래스 AuthModule {}
@@switch
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니
다;
'../users/users.module'에서 { UsersModule }을 임포트하고,
'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { AuthController } from './auth.controller';
import { jwtConstants } './constants'에서 임포트합니다;

모듈({ import:
  [
```

```
    UsersModule,
    JwtModule.register({
      global: true,
      비밀: jwtConstants.secret, signOptions:
      { expiresIn: '60s' },
    }),
  ],
  공급자: [AuthService], 컨트롤러:
  [AuthController], 내보내기:
  [AuthService],
})
내보내기 클래스 AuthModule {}
```

> 힌트 힌트 작업을 더 쉽게 하기 위해 JwtModule을 전역으로 등록하고 있습니다. 즉, 애플리케이션의 다른 곳에서는 JwtModule을 임포트할 필요가 없습니다.

구성 객체를 전달하여 register()를 사용하여 JwtModule을 구성합니다. Nest JwtModule에 대한 자세한 내용은 여기를, 사용 가능한 구성 옵션에 대한 자세한 내용은 여기를 참조하세요.

계속해서 cURL을 사용하여 경로를 다시 테스트해 보겠습니다. UsersService에 하드코딩된 모든 사용자 객체로 테스트할 수 있습니다.

```
인증/로그인에 $ # POST 보내기
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
참고: 위 JWT는 잘렸습니다.
```

### 인증 가드 구현

이제 마지막 요구 사항인 요청에 유효한 JWT가 있어야 엔드포인트를 보호하는 문제를 해결할 수 있습니다. 경로를 보호하는 데 사용할 수 있는 AuthGuard를 생성하여 이를 수행합니다.

```
@@파일명(auth/auth.guard)
import {
  CanActivate,
  ExecutionContext,
  Injectable,
  UnauthorizedException,
}를 '@nestjs/common'에서 가져옵니다;
import { JwtService } from '@nestjs/jwt';
import { jwtConstants } './constants'에서 임포
트; 'express'에서 { Request }를 임포트합니다;

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
```

```
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      새로운 UnauthorizedException()을 던집니다;
    }
    try {
      const payload = await this.jwtService.verifyAsync(
        token,
        {
          비밀: jwtConstants.secret
        }
      );
      // 💡 여기서 페이로드를 요청 객체에 할당합니다.
      //를 추가하여 라우트 핸들러 요청['user'] = 페이로드에서 액세스할 수
      있도록 합니다;
    } catch {
      새로운 UnauthorizedException()을 던집니다;
    }
    참을 반환합니다;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    반환 유형 === '무기명' ? 토큰 : undefined;
  }
}
```

이제 보호 경로를 구현하고 이를 보호하기 위해 `AuthGuard`를 등록할 수 있습니다.

`auth.controller.ts` 파일을 열고 아래와 같이 업데이트합니다:

```
@@파일명(auth.controller) 가져오
기 {
  본문, 컨트롤러,
  가져오기,
  HttpCode,
  HttpStatus,
  게시, 요청, 사
  용가드
}를 '@nestjs/common'에서 가져옵니다;
'./auth.guard'에서 { AuthGuard }를 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니
다;

컨트롤러('auth')
내보내기 클래스 AuthController {
  constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @Post('login')
  signIn(@Body() signInDto: Record<string, any>) {
```

```
    this.authService.signIn(signInDto.사용자이름, signInDto.비밀번호)을 반환
합니다;
  }

  UseGuards(AuthGuard)
  @Get('profile')
  getProfile(@Request() req) {
    req.user를 반환합니다;
  }
}
```

방금 생성한 AuthGuard를 `GET /profile` 경로에 적용하여 보호되도록 합니다. 앱이 실행 중인지 확인하고

`cURL을` 사용하여 경로를 테스트합니다.

```
$ # GET /profile
curl http://localhost:3000/auth/profile
{"statusCode":401,"message":"승인되지 않음"}

$ # POST /auth/login
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."}

이전 단계에서 무기명 코드로 반환된 access_token을 사용하여 /profile을 GET합니다.
curl http://localhost:3000/auth/profile -H "인증: 무기명
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."
{"sub":1,"username":"john","iat":...,"exp":...}
```

`인증 모듈에서` JWT의 만료 시간을 `60초로` 구성했습니다. 이는 너무 짧은 만료 시간이며 토큰 만료 및 새로 고침에 대한 세부 사항을 다루는 것은 이 문서의 범위를 벗어납니다. 하지만 JWT의 중요한 특성을 보여주기 위해 이 방법을 선택했습니다. 인증 후 60초를 기다렸다가 `GET /auth/프로필` 요청을 시도하면 `401 권한 없음` 응답을 받게 됩니다. 이는 `@nestjs/jwt가` 자동으로 JWT의 만료 시간을 확인하므로 애플리케이션에서 직접 확인해야 하는 수고를 덜어주기 때문입니다.

이제 JWT 인증 구현이 완료되었습니다. 이제 자바스크립트 클라이언트(예: Angular/React/Vue) 및 기타 자바스크립트 앱이 트위터의 API 서버와 안전하게 인증하고 통신할 수 있습니다.

### 전 세계적으로 인증 사용

대부분의 엔드포인트를 기본적으로 보호해야 하는 경우, 인증 가드를 전역 가드로 등록하고 각 컨트롤러 위에

`@UseGuards()` 데코레이터를 사용하는 대신 어떤 경로를 공개해야 하는지 플래그를 지정하면 됩니다.

먼저, 다음 구성을 사용하여 `AuthGuard`를 전역 가드로 등록합니다(예: `AuthModule` 등 모든 모듈에서):

```
제공자: [
  {
    제공: APP_GUARD, useClass:
    AuthGuard,
  },
],
```

이 설정이 완료되면 Nest는 모든 엔드포인트에 `AuthGuard를` 자동으로 바인딩합니다.

이제 경로를 공개로 선언하는 메커니즘을 제공해야 합니다. 이를 위해 `SetMetadata` 데코레이터 팩토리 함수를 사용하여 사용자 정의 데코레이터를 만들 수 있습니다.

```
import { SetMetadata } from '@nestjs/common';

export const IS_PUBLIC_KEY = 'isPublic';
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

위 파일에서 두 개의 상수를 내보냈습니다. 하나는 `IS_PUBLIC_KEY라는` 메타데이터 키이고, 다른 하나는 `Public이라고` 부를 새 데코레이터 자체입니다(프로젝트에 맞는 다른 이름을 지정할 수 있습니다).

이제 사용자 정의 `@Public()` 데코레이터가 생겼으므로 다음과 같이 모든 메서드를 데코레이션하는 데 사용할 수 있습니다:

```
Public()
@Get()
findAll() {
  반환 [];
}
```

마지막으로, `"isPublic"` 메타데이터가 발견되면 `AuthGuard가 참을` 반환하도록 해야 합니다. 이를 위해 `Reflector` 클래스를 사용하겠습니다(자세한 내용은 여기를 참조하세요).

```
@Injectable()
내보내기 클래스 AuthGuard 구현 CanActivate { 생성자(private jwtService:
  JwtService, 비공개 리플렉터:
리플렉터) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const isPublic = this.reflector.getAllAndOverride<boolean>
(is_public_key, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) {
      // □이 조건이 참을 반환하는
      것을 참조하십시오;
```

```
    }

    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      새로운 UnauthorizedException()을 던집니다;
    }
    try {
      const payload = await this.jwtService.verifyAsync(token, {
        secret: jwtConstants.secret,
      });
      // 💡 여기서 페이로드를 요청 객체에 할당합니다.
      //를 추가하여 라우트 핸들러 요청['user'] = 페이로드에서 액세스할 수
      있도록 합니다;
    } catch {
      새로운 UnauthorizedException()을 던집니다;
    }
    참을 반환합니다;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    반환 유형 === '무기명' ? 토큰 : undefined;
  }
}
```

여권 통합

Passport는 커뮤니티에서 잘 알려져 있고 많은 프로덕션 애플리케이션에서 성공적으로 사용되는 가장 인기 있는 node.js 인증 라이브러리입니다. 이 라이브러리를 Nest 애플리케이션과 통합하는 방법은 `@nestjs/passport` 모듈을 사용하면 간단합니다.

Passport와 NestJS를 통합하는 방법을 알아보려면 이 챕터를 확인하세요. 예제

이 장의 전체 코드 버전은 여기에서 확인할 수 있습니다.

## 권한 부여

권한 부여는 사용자가 수행할 수 있는 작업을 결정하는 프로세스를 말합니다. 예를 들어 관리 사용자는 글을 작성, 편집 및 삭제할 수 있습니다. 관리자가 아닌 사용자에게는 게시물을 읽을 수 있는 권한만 부여됩니다.

권한 부여는 인증과 직교하며 독립적입니다. 그러나 권한 부여에는 인증 메커니즘이 필요합니다.

인증을 처리하는 방법과 전략에는 여러 가지가 있습니다. 모든 프로젝트에 대해 취하는 접근 방식은 특정 애플리케이션 요구 사항에 따라 다릅니다. 이 장에서는 다양한 요구 사항에 맞게 조정할 수 있는 몇 가지 권한 부여 접근 방식을 소개합니다.

### 기본 RBAC 구현

역할 기반 액세스 제어(RBAC)는 역할과 권한을 중심으로 정의된 정책 중립적인 액세스 제어 메커니즘입니다.
이 섹션에서는 Nest 가드를 사용하여 매우 기본적인 RBAC 메커니즘을 구현하는 방법을 보여드리겠습니다.

먼저 시스템에서 역할을 나타내는 `역할` 열거형을 만들어 보겠습니다:

```
@@파일 이름(role.enum)
내보내기 열거형 Role {
  사용자 = '사용자', 관
  리자 = '관리자',
}
```

> 정보 힌트 보다 정교한 시스템에서는 데이터베이스 내에 역할을 저장하거나 외부 인증 공급자로부터 역할을 가져올 수 있습니다.

이제 `@Roles()` 데코레이터를 만들 수 있습니다. 이 데코레이터를 사용하면 특정 리소스에 액세스하는 데 필요한 역할을 지정할 수 있습니다.

```
@@파일명(역할.데코레이터)
'@nestjs/common'에서 { SetMetadata }를 가져오고,
'../enums/role.enum'에서 { Role }을 가져옵니다;

export const ROLES_KEY = 'roles';
export const Roles = (...roles: Role[]) => SetMetadata(ROLES_KEY, roles);
@@switch
'@nestjs/common'에서 { SetMetadata }를 가져옵니다;

export const ROLES_KEY = 'roles';
export const Roles = (...roles) => SetMetadata(ROLES_KEY, roles);
```

이제 사용자 지정 `@Roles()` 데코레이터가 있으므로 이를 사용하여 모든 라우트 핸들러를 데코레이션할 수 있습니다.

```
@@파일명(cats.controller) @Post()
@Roles(Role.Admin)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@스위치
@포스트()
@Roles(Role.Admin)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

마지막으로, 현재 사용자에게 할당된 역할을 현재 처리 중인 경로에 필요한 실제 역할과 비교하는 `RolesGuard` 클래스를 생성합니다. 경로의 역할(사용자 정의 메타데이터)에 액세스하기 위해 프레임워크에서 기본 제공되고 `@nestjs/core` 패키지에서 노출되는 `Reflector` 헬퍼 클래스를 사용합니다.

```
@@파일명(roles.guard)
'@nestjs/common'에서 { Injectable, CanActivate, ExecutionContext }를 임포
트합니다;
'@nestjs/core'에서 { Reflector }를 가져옵니다;

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>
(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) { 참을
      반환합니다;
    }
    const { user } = context.switchToHttp().getRequest();
    반환 requiredRoles.some((role) => user.roles?.includes(role));
  }
}
@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 임포트하고,
'@nestjs/core'에서 { Reflector }를 임포트합니다;

주입 가능() @의존성(반사기) 내보내기
클래스 RolesGuard {
  constructor(reflector) {
    this.reflector = reflector;
```

```
  }

  canActivate(context) {
    const requiredRoles = this.reflector.getAllAndOverride(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) { 참을

      반환합니다;
    }
    const { user } = context.switchToHttp().getRequest();
    반환 requiredRoles.일부((role) => user.roles.includes(role));
  }
}
```

> 정보 힌트 상황에 맞는 방식으로 리플렉터를 활용하는 방법에 대한 자세한 내용은 실행 컨텍스트 장의 리플렉션 및 메타데이터 섹션을 참조하세요.

> 경고 이 예제는 라우트 핸들러 수준에서 역할의 존재 여부만 확인하므로 "기본"으로 명명되었습니다. 실제 애플리케이션에서는 여러 작업을 포함하는 엔드포인트/핸들러가 있을 수 있으며, 각 작업에는 특정 권한 집합이 필요할 수 있습니다. 이 경우 비즈니스 로직 내 어딘가에 역할을 확인하는 메커니즘을 제공해야 하며, 권한을 특정 작업과 연결하는 중앙 집중식 위치가 없기 때문에 유지 관리가 다소 어려워집니다.

이 예제에서는 `요청.user에` 사용자 인스턴스와 허용된 역할이 포함되어 있다고 가정했습니다( `역할` 속성). 앱에서는 사용자 지정 인증 가드에서 해당 연결을 만들 수 있습니다.

- 자세한 내용은 인증 챕터를 참조하세요.

이 예제가 제대로 작동하려면 `사용자` 클래스의 모양이 다음과 같아야 합니다:

```
사용자 클래스 {
  // ...다른 속성 역할: Role[];
}
```

마지막으로, 컨트롤러 수준에서 또는 전역적으로 RolesGuard를 등록하세요:

```
제공자: [
  {
    제공: APP_GUARD, useClass:
    RolesGuard,
  },
],
```

권한이 부족한 사용자가 엔드포인트를 요청하면 Nest는 자동으로 다음과 같은 응답을 반환합니다:

```
{
  "상태코드": 403,
  "메시지": "금지된 리소스", "오류": "금지됨"
}
```

> 정보 힌트 다른 오류 응답을 반환하려면 부울 값을 반환하는 대신 고유한 특정 예외를 던져야 합니다.

### 클레임 기반 권한 부여

ID가 생성되면 신뢰할 수 있는 당사자가 발급한 하나 이상의 클레임이 할당될 수 있습니다. 클레임은 주체가 무엇인지가 아니라 주체가 할 수 있는 일을 나타내는 이름-값 쌍입니다.

Nest에서 클레임 기반 권한 부여를 구현하려면 위의 RBAC 섹션에서 설명한 것과 동일한 단계를 따르되 한 가지 중요한 차이점이 있는데, 특정 역할을 확인하는 대신 권한을 비교해야 한다는 점입니다. 모든 사용자에게는 일련의 권한이 할당됩니다. 마찬가지로 각 리소스/엔드포인트는 해당 리소스/엔드포인트에 액세스하는 데 필요한 권한을 정의합니다(예: 전용 `@RequirePermissions()` 데코레이터를 통해).

```
@@파일명(cats.controller) @Post()
@RequirePermissions(Permission.CREATE_CAT)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@스위치
@포스트()
@RequirePermissions(Permission.CREATE_CAT)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

> 정보 힌트 위의 예에서 `권한`(RBAC 섹션에서 설명한 `역할과` 유사)은 시스템에서 사용 가능한 모든 권한을 포함하는 TypeScript 열거형입니다.

### CASL 통합

CASL은 특정 클라이언트가 액세스할 수 있는 리소스를 제한하는 동형 권한 부여 라이브러리입니다. 점진적으로 채택할 수 있도록 설계되었으며 간단한 클레임 기반부터 완전한 기능을 갖춘 주제 및 속성 기반 권한 부여까지 쉽

게 확장할 수 있습니다.

시작하려면 먼저 `@casl/ability` 패키지를 설치하세요:

```
$ npm i @casl/ability
```

> 정보 힌트 이 예제에서는 CASL을 선택했지만 액세스제어나 acl을 사용할 수 있습니다.

설치가 완료되면 CASL의 메커니즘을 설명하기 위해 두 개의 엔티티 클래스를 정의하겠습니다: User와 `Article`입니다.

```
사용자 클래스 {
  id: 숫자;
  isAdmin: 부울입니다;
}
```

`사용자` 클래스는 고유한 사용자 식별자인 id와 사용자에게 관리자 권한이 있는지 여부를 나타내는 isAdmin의 두 가지 속성으로 구성됩니다.

```
기사 클래스 { id: 숫
  자;
  isPublished: 부울; authorId: 숫
  자;
}
```

문서 클래스에는 각각 `id`, `isPublished`, `authorId`라는 세 가지 속성이 있습니다. id는 고유한 문서 식별자이고, `isPublished`는 문서가 이미 게시되었는지 여부를 나타내며, authorId는 문서를 작성한 사용자의 ID입니다.

이제 이 예제에 대한 요구 사항을 검토하고 구체화해 보겠습니다:

- 관리자는 모든 엔티티를 관리(생성/읽기/업데이트/삭제)할 수 있습니다
- 사용자는 모든 항목에 읽기 전용 액세스 권한이 있습니다.
- 사용자는 자신의 글을 업데이트할 수 있습니다(`article.authorId === userId`).
- 이미 게시된 글은 삭제할 수 없습니다(`article.isPublished === true`).

이를 염두에 두고 사용자가 엔티티로 수행할 수 있는 모든 가능한 작업을 나타내는 `Action` 열거형을 만드는 것으로 시작할 수 있습니다:

```
export enum Action {
  Manage = '관리',
  Create = '만들기',
  Read = '읽기',
  Update = '업데이트',
  Delete = '삭제',
}
```

> 경고 알림 `관리는` CASL에서 "모든" 작업을 나타내는 특수 키워드입니다.

CASL 라이브러리를 캡슐화하기 위해 이제 CaslModule과 `CaslAbilityFactory를` 생성해 보겠습니다.

```
nest g 모듈 casl
nest g 클래스 casl/casl-ability.factory
```

이렇게 하면 `CaslAbilityFactory에서 createForUser()` 메서드를 정의할 수 있습니다. 이 메서드는 주어진 사용자에 대한 `어빌리티` 객체를 생성합니다:

```
유형 Subjects = InferSubjects<기사 유형 | 사용자 유형> | '모두'; 내보내기 유형

AppAbility = Ability<[Action, Subjects]>;

@Injectable()
export class CaslAbilityFactory {
  createForUser(user: User) {
    const { 할 수 있다, 할 수 없다, 빌드 } = 새로운 AbilityBuilder<
      Ability<[액션, 서브젝트]>
    >(어빌리티를 어빌리티클래스<앱어빌리티>로);

    if (user.isAdmin) {
      can(Action.Manage, 'all'); // 모든 것에 대한 읽기-쓰기 액세스 권한
    } else {
      can(Action.Read, 'all'); // 모든 것에 대한 읽기 전용 액세스
    }

    can(Action.Update, Article, { authorId: user.id });
    cannot(Action.Delete, Article, { isPublished: true });

    반환 빌드({
      // 자세한 내용은 https://casl.js.org/v5/en/guide/subject-type-
detection#use- classes-as-subject-types를 참조하세요.
      detectSubjectType: (item) =>
        item.constructor를 ExtractSubjectType<Subjects>로 설정합니다,
    });
  }
}
```

> 경고 모두는 CASL에서 '모든 주제'를 나타내는 특수 키워드입니다.

> 정보 힌트 `어빌리티`, `어빌리티빌더`, `어빌리티클래스`, `추출주제유형` 클래스는 `@casl/ability` 패키지에서 내보냅니다.

> 정보 힌트 `detectSubjectType` 옵션을 사용하면 CASL이 객체에서 주제 유형을 가져오는 방법을 이해할 수 있습니다. 자세한 내용은 CASL 설명서를 참조하세요.

위의 예시에서는 `AbilityBuilder` 클래스를 사용하여 `Ability` 인스턴스를 만들었습니다. 짐작하셨겠지만, 수

락할 수 있는 것과 수락할 수 없는 것은 같은 인수를 받지만 의미가 다르며, 수락할 수 있는 것은 지정된 대상에 대해 작업을 수행할 수 있고 금지할 수 없습니다. 둘 다 최대 4개의 인수를 받을 수 있습니다. 이러한 함수에 대해 자세히 알아보려면 공식 CASL 문서를 참조하세요.

마지막으로, `CaslAbilityFactory`를 제공자 및 내보내기 배열에 추가해야 합니다.

`CaslModule` 모듈 정의:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./casl-ability.factory'에서 { CaslAbilityFactory }를 가져옵니다;

모듈({
  제공자: [CaslAbilityFactory], 내보내기:
  [CaslAbilityFactory],
})
내보내기 클래스 CaslModule {}
```

이렇게 하면 호스트 컨텍스트에서 CaslModule을 임포트하기만 하면 표준 생성자 주입을 사용하여 모든 클래스에 `CaslAbilityFactory`를 주입할 수 있습니다:

```
constructor(private caslAbilityFactory: CaslAbilityFactory) {}
```

그런 다음 다음과 같이 수업에서 사용하세요.

```
const ability = this.caslAbilityFactory.createForUser(user);
if (ability.can(Action.Read, 'all')) {
  // "사용자"는 모든 항목에 대한 읽기 권한이 있습니다.
}
```

> 정보 힌트 공식 CASL 문서에서 `어빌리티` 클래스에 대해 자세히 알아보세요.

예를 들어 관리자가 아닌 사용자가 있다고 가정해 보겠습니다. 이 경우 사용자는 문서를 읽을 수 있어야 하지만 새 문서를 만들거나 기존 문서를 삭제하는 것은 금지되어야 합니다.

```
const user = new User();
user.isAdmin = false;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Read, Article); // 참
ability.can(Action.Delete, Article); // 거짓
ability.can(Action.Create, Article); // 거짓
```

> 정보 힌트 Ability와 `AbilityBuilder` 클래스 모두 `할 수 있는` 메서드와 `할 수 없는` 메서드를 제공하지만 용도가 다르고 약간 다른 인수를 허용합니다.

또한 요구사항에 명시한 대로 사용자가 문서를 업데이트할 수 있어야 합니다:

```
const user = new User();
user.id = 1;

const article = new Article();
article.authorId = user.id;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Update, article); // true

article.authorId = 2;
ability.can(Action.Update, article); // false
```

보시다시피 `Ability` 인스턴스를 사용하면 매우 읽기 쉬운 방식으로 권한을 확인할 수 있습니다. 마찬가지로 `AbilityBuilder`를 사용하면 비슷한 방식으로 권한을 정의하고 다양한 조건을 지정할 수 있습니다. 더 많은 예제를 보려면 공식 문서를 참조하세요.

고급: **정책 가드** 구현하기

이 섹션에서는 메서드 수준에서 구성할 수 있는 특정 권한 부여 정책을 사용자가 충족하는지 확인하는 다소 정교한 가드를 구축하는 방법을 보여드리겠습니다(클래스 수준에서 구성된 정책도 존중하도록 확장할 수 있습니다). 이 예제에서는 예시를 보여주기 위해 CASL 패키지를 사용하지만 이 라이브러리를 반드시 사용해야 하는 것은 아닙니다. 또한 이전 섹션에서 생성한 `CaslAbilityFactory` 프로바이더를 사용할 것입니다.

먼저 요구 사항을 구체화해 보겠습니다. 목표는 라우트 핸들러별로 정책 검사를 지정할 수 있는 메커니즘을 제공하는 것입니다. 보다 간단한 검사와 함수형 코드를 선호하는 사용자를 위해 객체와 함수를 모두 지원할 것입니다.

정책 처리기를 위한 인터페이스를 정의하는 것부터 시작하겠습니다:

```
'../casl/casl-ability.factory'에서 { AppAbility } 임포트; 인터페이

스 IPolicyHandler {
  handle(ability: AppAbility): boolean;
}

유형 PolicyHandlerCallback = (ability: AppAbility) => boolean;


export 유형 PolicyHandler = IPolicyHandler | PolicyHandlerCallback;
```

위에서 언급했듯이 정책 처리기를 정의하는 두 가지 가능한 방법, 즉 객체(`IPolicyHandler` 인터페이스를 구현하는 클래스의 인스턴스)와 함수(`PolicyHandlerCallback` 유형을 충족하는)를 제공했습니다.

이를 통해 `@CheckPolicies()` 데코레이터를 만들 수 있습니다. 이 데코레이터를 사용하면 특정 리소스에 액세스하기 위해 어떤 정책을 충족해야 하는지 지정할 수 있습니다.

```
export const CHECK_POLICIES_KEY = 'check_policy';
export const CheckPolicies = (...handlers: PolicyHandler[]) =>
  SetMetadata(CHECK_POLICIES_KEY, handlers);
```

이제 라우트 핸들러에 바인딩된 모든 정책 핸들러를 추출하고 실행하는 `PoliciesGuard를` 만들어 보겠습니다.

```
@Injectable()
내보내기 클래스 PoliciesGuard 구현 CanActivate { 생성자(
    개인 리플렉터: 리플렉터,
    private caslAbilityFactory: CaslAbilityFactory,
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const policyHandlers =
      this.reflector.get<정책핸들러[]>(
        CHECK_POLICIES_KEY,
        컨텍스트.getHandler(),
      ) || [];

    const { user } = context.switchToHttp().getRequest();
    const ability = this.caslAbilityFactory.createForUser(user);

    반환 정책 핸들러.every((핸들러) => this.execPolicyHandler(핸들러, 능력),
    );
  }

  private execPolicyHandler(handler: PolicyHandler, ability: AppAbility) {
    if (typeof handler === 'function') {
      반환 핸들러(어빌리티);
    }
    핸들러.핸들(ability)을 반환합니다;
  }
}
```

> 정보 힌트 이 예제에서는 `요청.user에` 사용자 인스턴스가 포함되어 있다고 가정했습니다. 앱에서는 사용자 지정 인증 가드에서 해당 연결을 만들 수 있습니다(자세한 내용은 인증 챕터를 참조하세요).

이 예제를 자세히 살펴봅시다. `정책 핸들러는 @CheckPolicies()` 데코레이터를 통해 메서드에 할당된 핸들러의 배열입니다. 다음으로, 사용자가 특정 작업을 수행할 수 있는 충분한 권한을 가지고 있는지 확인할 수 있도록 `Ability` 객체를 구성하는 `CaslAbilityFactory#create` 메서드를 사용합니다. 이 객체를 정책 핸들러에 전달하는데, 이 `핸들러는` 함수이거나 `IPolicyHandler를` 구현하는 클래스의 인스턴스이며, 부울을 반환하는 `handle()` 메서드를 노출합니다. 마지막으로, 모든 핸들러가 `참` 값을 반환하는지 확인하기 위해 `Array#every`

메서드를 사용합니다.

마지막으로 이 가드를 테스트하려면 다음과 같이 라우트 핸들러에 바인딩하고 인라인 정책 핸들러(기능적 접근 방식)를 등록합니다:

```
Get()
@UseGuards(PoliciesGuard)
CheckPolicies((ability: AppAbility) => ability.can(Action.Read, Article))
findAll() {
  이.기사서비스.모두 찾기()를 반환합니다;
}
```

또는 `IPolicyHandler` 인터페이스를 구현하는 클래스를 정의할 수도 있습니다:

```
export class ReadArticlePolicyHandler implements IPolicyHandler {
  handle(ability: AppAbility) {
    ability.can(Action.Read, Article)을 반환합니다;
  }
}
```

그리고 다음과 같이 사용하세요:

```
Get()
@UseGuards(PoliciesGuard)
CheckPolicies(new ReadArticlePolicyHandler())
findAll() {
  이.기사서비스.모두 찾기()를 반환합니다;
}
```

> 경고 주의 `새` 키워드를 사용하여 정책 핸들러를 제자리에 인스턴스화해야 하므로 `ReadArticlePolicyHandler` 클래스는 종속성 주입을 사용할 수 없습니다. 이 문제는 `ModuleRef#get` 메서드를 사용하여 해결할 수 있습니다(자세한 내용은 여기를 참조하세요). 기본적으로 `@CheckPolicies()` 데코레이터를 통해 함수와 인스턴스를 등록하는 대신 `Type<IPolicyHandler>` 전달을 허용해야 합니다. 그런 다음 가드 내부에서 유형 참조를 사용하여 인스턴스를 검색할 수 있습니다: `moduleRef.get(YOUR_HANDLER_TYPE)` 또는 `ModuleRef#create` 메서드를 사용하여 인스턴스를 동적으로 인스턴스화할 수도 있습니다.

## 암호화 및 해싱

암호화는 정보를 인코딩하는 과정입니다. 이 프로세스는 일반 텍스트라고 하는 정보의 원래 표현을 암호 텍스트라고 하는 대체 형식으로 변환합니다. 이상적으로는 권한이 있는 당사자만 암호 텍스트를 다시 일반 텍스트로 해독하여 원본 정보에 액세스할 수 있습니다. 암호화는 그 자체로 간섭을 방지하는 것이 아니라 가로채려는 사람이 이해할 수 있는 콘텐츠를 거부합니다. 암호화는 양방향 기능이며, 암호화된 내용은 적절한 키를 사용하여 해독할 수 있습니다.

해싱은 주어진 키를 다른 값으로 변환하는 과정입니다. 해시 함수는 수학적 알고리즘에 따라 새로운 값을 생성하는 데 사용됩니다. 해싱이 완료되면 출력에서 입력으로 변경하는 것이 불가능해야 합니다.

### 암호화

Node.js는 문자열, 숫자, 버퍼, 스트림 등을 암호화하고 해독하는 데 사용할 수 있는 내장 암호화 모듈을 제공합니다. Nest 자체는 불필요한 추상화를 피하기 위해 이 모듈 위에 추가 패키지를 제공하지 않습니다.

예를 들어 AES(고급 암호화 시스템) `'aes-256-ctr'` 알고리즘 CTR 암호화 모드를 사용하겠습니다.

```
'crypto'에서 { createCipheriv, randomBytes, scrypt }를 가져오고,
'util'에서 { promiseify }를 가져옵니다;

const iv = randomBytes(16);
const password = '키 생성에 사용된 비밀번호';

// 키 길이는 알고리즘에 따라 다릅니다.
// 이 경우 aes256의 경우 32바이트입니다.
const key = (await promiseify(scrypt)(password, 'salt', 32)) as
Buffer; const cipher = createCipheriv('aes-256-ctr', key, iv);

const textToEncrypt = 'Nest';
const encryptedText = Buffer.concat([
  cipher.update(textToEncrypt),
  cipher.final(),
]);
```

이제 `암호화된 텍스트` 값을 해독합니다:

```
'crypto'에서 { createDecipheriv }를 가져옵니다;

const decipher = createDecipheriv('aes-256-ctr', key, iv);
const decryptedText = Buffer.concat([[
  decipher.update(encryptedText),
  decipher.final(),
]);
```

## 해싱

해싱의 경우, bcrypt 또는 argon2 패키지를 사용하는 것을 권장합니다. Nest 자체는 불필요한 추상화를 도입하지 않기 위해(학습 곡선을 짧게 만들기 위해) 이러한 모듈 위에 추가 래퍼를 제공하지 않습니다.

예를 들어, bcrypt를 사용하여 임의의 비밀번호를 해시해 보겠습니다.

먼저 필요한 패키지를 설치합니다:

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

설치가 완료되면 다음과 같이 `해시` 함수를 사용할 수 있습니다:

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

const saltOrRounds = 10;
const password = 'random_password';
const hash = await bcrypt.hash(password, saltOrRounds);
```

소금을 생성하려면 `genSalt` 함수를 사용합니다:

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

비밀번호를 비교/확인하려면 `비교` 기능을 사용하세요:

```
const isMatch = await bcrypt.compare(비밀번호, 해시);
```

사용 가능한 기능에 대한 자세한 내용은 여기에서 확인할 수 있습니다.

## 헬멧

헬멧은 HTTP 헤더를 적절히 설정하여 잘 알려진 웹 취약점으로부터 앱을 보호할 수 있습니다. 일반적으로 Helmet은 보안 관련 HTTP 헤더를 설정하는 작은 미들웨어 함수의 모음에 불과합니다(자세히 보기).

> 정보 힌트 헬멧을 전역으로 적용하거나 등록하는 것은 다른 호출보다 먼저 이루어져야 합니다. `앱.사용()` 또는 `앱.사용()을` 호출할 수 있는 설정 함수를 사용할 수 없습니다. 이는 미들웨어/경로가 정의되는 순서가 중요한 기본 플랫폼(예: Express 또는 Fastify)의 작동 방식 때문입니다. 경로를 정의한 후에 헬멧이나 `코어와` 같은 미들웨어를 사용하는 경우 해당 미들웨어는 해당 경로에 적용되지 않으며 미들웨어 다음에 정의된 경로에만 적용됩니다.

Express와 함께 사용(기본값)

필요한 패키지를 설치하는 것으로 시작하세요.

```
npm i --save helmet
```

설치가 완료되면 글로벌 미들웨어로 적용합니다.

```
'헬멧'에서 헬멧을 가져옵니다;
// 초기화 파일 어딘가에 앱.사용(헬멧());
```

> 경고 `헬멧`, `@apollo/server`(4.x) 및 Apollo 샌드박스를 사용할 때 Apollo 샌드박스에서 CSP에 문제가 있을 수 있습니다. 이 문제를 해결하려면 아래 그림과 같이 CSP를 구성하세요:

```
app.use(helmet({
  crossOriginEmbedderPolicy: false,
  contentSecurityPolicy: {
    지시어를 사용합니다: {
      imgSrc: [`'self'`, 'data:', 'apollo-server-landing-
page.cdn.apollographql.com'],
      scriptSrc: [`'self'`, `https: 'unsafe-inline'`],
      manifestSrc: [`'self'`, 'apollo-server-landing-'
page.cdn.apollographql.com'],
      frameSrc: [`'self'`, 'sandbox.embed.apollographql.com'],
    },
  },
}));
```

Fastify와 함께 사용

FastifyAdapter를 사용하는 경우 @fastify/helmet 패키지를 설치합니다:

```
npm i --save @fastify/helmet
```

fastify-helmet을 미들웨어가 아닌 Fastify 플러그인으로 사용해야 합니다(즉, `app.register()` 사용):

```
'@fastify/helmet'에서 헬멧 가져오기
// 초기화 파일 어딘가에 await
app.register(helmet)
```

경고 경고 `apollo-server-fastify` 및 @fastify/helmet을 사용하는 경우 GraphQL 플레이그라운드에서 CSP에 문제가 있을 수 있으며, 이 충돌을 해결하려면 아래와 같이 CSP를 구성하세요:

```
await app.register(fastifyHelmet, {
   contentSecurityPolicy: {
     지시어를 사용합니다: {
       defaultSrc: [`'self'`, 'unpkg.com'],
       styleSrc: [
         `'self'`,
         `'unsafe-inline'`,
         'cdn.jsdelivr.net',
         'fonts.googleapis.com',
         'unpkg.com',
       ],
       fontSrc: [`'self'`, 'fonts.gstatic.com', 'data:'],
       imgSrc: [`'self'`, 'data:', 'cdn.jsdelivr.net'],
       scriptSrc: [
         `'self'`,
         `https: 'unsafe-inline'`,
         `cdn.jsdelivr.net`,
         'unsafe-val'`,
       ],
     },
   },
 });

// CSP를 전혀 사용하지 않을 경우 다음과 같이 사용할 수 있습니다: await
app.register(fastifyHelmet, {
  contentSecurityPolicy: false,
});
```

# CORS

CORS(교차 출처 리소스 공유)는 다른 도메인에서 리소스를 요청할 수 있는 메커니즘입니다. Nest는 내부적으로 Express cors 패키지를 사용합니다. 이 패키지는 요구 사항에 따라 사용자 정의할 수 있는 다양한 옵션을 제공합니다.

## 시작하기

CORS를 활성화하려면 Nest 애플리케이션 객체에서 `enableCors()` 메서드를 호출합니다.

```
const app = await NestFactory.create(AppModule);
app.enableCors();
await app.listen(3000);
```

`enableCors()` 메서드는 선택적 구성 객체 인수를 받습니다. 이 객체의 사용 가능한 속성은 공식 CORS 문서에 설명되어 있습니다. 또 다른 방법은 요청에 따라 비동기적으로(즉석에서) 구성 객체를 정의할 수 있는 콜백 함수를 전달하는 것입니다.

또는 `create()` 메서드의 옵션 객체를 통해 CORS를 활성화합니다. 기본 설정으로 CORS를 활성화하려면 `cors` 속성을 `true로` 설정합니다. 또는 CORS 구성 객체 또는 콜백 함수를 `cors` 속성 값으로 전달하여 동작을 사용자 지정할 수 있습니다.

```
const app = await NestFactory.create(AppModule, { cors: true });
await app.listen(3000);
```

## CSRF 보호

크로스 사이트 요청 위조(CSRF 또는 XSRF라고도 함)는 웹 애플리케이션이 신뢰하는 사용자로부터 무단 명령이 전송되는 웹사이트의 악의적인 익스플로잇 유형입니다. 이러한 종류의 공격을 완화하기 위해 csurf 패키지를 사용할 수 있습니다.

### Express와 함께 사용(기본값)

필요한 패키지를 설치하는 것으로 시작하세요:

```
$ npm i --save csurf
```

> 경고 경고 이 패키지는 더 이상 사용되지 않습니다. 자세한 내용은 `csurf` 문서를 참조하세요.

> 경고 경고 `csurf` 문서에 설명된 대로 이 미들웨어를 사용하려면 세션 미들웨어 또는 `쿠키 파서를` 먼저 초기화해야 합니다. 자세한 지침은 해당 문서를 참조하세요.

설치가 완료되면 `csurf` 미들웨어를 글로벌 미들웨어로 적용합니다.

```
'csurf'에서 *를 csurf로 가져옵니다;
// ...
// 초기화 파일 어딘가에 app.use(csurf());
```

### Fastify와 함께 사용

필요한 패키지를 설치하는 것으로 시작하세요:

```
npm i --save @fastify/csrf-protection
```

설치가 완료되면 다음과 같이 `@fastify/csrf-protection` 플러그인을 등록합니다:

```
'@fastify/csrf-protection'에서 fastifyCsrf를 가져옵니다;
// ...
// 스토리지 플러그인을 등록한 후 초기화 파일 어딘가에 있습니다.
await app.register(fastifyCsrf);
```

> 경고 경고 여기 `@fastify/csrf-protection` 문서에 설명된 대로 이 플러그인을 사용하려면 먼저 스토리지 플러그인을 초기화해야 합니다. 자세한 지침은 해당 문서를 참조하세요.

## 속도 제한

무차별 대입 공격으로부터 애플리케이션을 보호하는 일반적인 기술은 속도 제한입니다. 시작하려면 `@nestjs/throttler` 패키지를 설치해야 합니다.

```
$ npm i --save @nestjs/throttler
```

설치가 완료되면, 스로틀러모듈은 다른 네스트 패키지와 마찬가지로 `forRoot` 또는 `forRootAsync` 메서드를 사용하여 구성할 수 있습니다.

```
@@파일명(app.module) @Module({
  임포트합니다: [
    ThrottlerModule.forRoot({
      ttl: 60,
      제한: 10,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

위와 같이 보호되는 애플리케이션의 경로에 대한 TTL, 유효 시간 및 TTL 내의 최대 요청 수인 제한에 대한 전역 옵션을 설정합니다.

모듈을 가져온 다음에는 스로틀러가드를 바인딩할 방법을 선택할 수 있습니다. 가드 섹션에서 언급한 바인딩 방식은 무엇이든 괜찮습니다. 예를 들어 가드를 전역적으로 바인딩하려면 이 공급자를 모든 모듈에 추가하여 바인딩할 수 있습니다:

```
{
  제공: APP_GUARD, useClass:
  ThrottlerGuard
}
```

### 사용자 지정

가드를 컨트롤러에 바인딩하거나 전역적으로 바인딩하고 싶지만 하나 이상의 엔드포인트에 대해 속도 제한을 비활성화하려는 경우가 있을 수 있습니다. 이 경우 `@SkipThrottle()` 데코레이터를 사용하여 전체 클래스 또

는 단일 경로에 대한 스로틀러를 무효화할 수 있습니다. 모든 경로가 아닌 컨트롤러의 대부분을 제외하려는 경우 `@SkipThrottle()` 데코레이터는 부울을 받을 수도 있습니다.

```
스킵스로틀() @Controller('users')
사용자 컨트롤러 클래스 {} 내보내기
```

이 `@SkipThrottle()` 데코레이터는 경로 또는 클래스를 건너뛰거나 건너뛰는 클래스에서 경로 건너뛰기를 무효화하는 데 사용할 수 있습니다.

```
스킵스로틀() @Controller('users')
사용자 컨트롤러 클래스 내보내기 {
  // 이 경로에 속도 제한이 적용됩니다. 스킵스로틀(거짓)
  dontSkip() {
    "속도 제한이 있는 사용자 작업 목록."을 반환합니다;
  }
  // 이 경로는 속도 제한을 건너뜁니다. doSkip() {
    "속도 제한 없이 작업하는 사용자를 나열합니다."를 반환합니다;
  }
}
```

글로벌 모듈에 설정된 `제한` 및 `ttl을` 재정의하여 더 엄격하거나 느슨한 보안 옵션을 제공하는 데 사용할 수 있는 `@Throttle()` 데코레이터도 있습니다. 이 데코레이터는 클래스나 함수에도 사용할 수 있습니다. 이 데코레이터의 인수는 `limit, ttl` 순서이므로 순서가 중요합니다. 다음과 같이 구성해야 합니다:

```
// 속도 제한 및 기간에 대한 기본 구성을 재정의합니다. 스로틀(3, 60)
@Get()
findAll() {
  반환 "사용자 지정 속도 제한으로 작동하는 사용자 목록.";
}
```

### 프록시

애플리케이션이 프록시 서버 뒤에서 실행되는 경우 특정 HTTP 어댑터 옵션(express 및 fastify)에서 `신뢰 프록시` 옵션을 확인하고 활성화하세요. 이렇게 하면 `X-Forwarded-For` 헤더에서 원래 IP 주소를 가져올 수 있으며, `getTracker()` 메서드를 재정의하여 `req.ip가` 아닌 헤더에서 값을 가져올 수 있습니다. 다음 예제는 express와 fastify 모두에서 작동합니다:

```
// 스로틀러 비하인드-프록시.guard.ts
'@nestjs/throttler'에서 { ThrottlerGuard }를 가져오고,
'@nestjs/common'에서 { Injectable }을 가져옵니다;

@Injectable()
export class ThrottlerBehindProxyGard extends ThrottlerGard {
  protected getTracker(req: Record<string, any>): string {
    return req.ips.length ? req.ips[0] : req.ip; // 필요에 따라 IP 추출을
개별화합니다.
  }
```

```
}

// app.controller.ts
'./throttler-behind- proxy.guard'에서 { ThrottlerBehindProxyGuard }를 가
져옵니다;

사용가드(스로틀러비하인드프록시가드)
```

> 정보 힌트 익스프레스에 대한 요청 객체의 API는 여기에서, 패스트파이브에 대한 요청 객체의 API는 여기에서 찾을 수 있습니다.

웹 소켓

이 모듈은 웹소켓과 함께 작동할 수 있지만 일부 클래스 확장이 필요합니다. 이 모듈을 확장하려면 `ThrottlerGuard를` 생성하고 `핸들 요청` 메서드를 다음과 같이 재정의합니다:

```
@Injectable()
내보내기 클래스 WsThrottlerGuard extends ThrottlerGuard {
  async handleRequest(context: ExecutionContext, limit: number, ttl:
number): Promise<boolean> {
    const client = context.switchToWs().getClient();
    const ip = client._socket.remoteAddress
    const key = this.generateKey(context, ip);
    const { totalHits } = await this.storageService.increment(key, ttl);

    if (totalHits > limit) {
      새로운 ThrottlerException()을 던집니다;
    }

    참을 반환합니다;
  }
}
```

> 정보 힌트 ws를 사용하는 경우 `_socket을 conn로` 바꿔야 합니다.

웹소켓으로 작업할 때 몇 가지 유의해야 할 사항이 있습니다:

- 가드는 `앱_가드` 또는 `앱.사용글로벌가드()`에 등록할 수 없습니다.
- 한계에 도달하면 Nest는 `예외` 이벤트를 발생시키므로 이에 대비한 리스너가 있는지 확인하세요.

> 정보 힌트 `@nestjs/platform-ws` 패키지를 사용하는 경우 다음을 사용할 수 있습니다.
> 대신 `client._socket.remoteAddress를 입력하세요.`

## GraphQL

스로틀러 가드는 GraphQL 요청을 처리하는 데에도 사용할 수 있습니다. 다시 말하지만, 가드를 확장할 수 있지만 이번에는 `getRequestResponse` 메서드가 재정의됩니다.

```
@Injectable()
export class GqlThrottlerGuard extends ThrottlerGuard {
  getRequestResponse(context: ExecutionContext) {
    const gqlCtx = GqlExecutionContext.create(context);
    const ctx = gqlCtx.getContext();
    반환 { req: ctx.req, res: ctx.res };
  }
}
```

## 구성

다음 옵션은 스로틀러 모듈에 유효합니다:

| | |
|---|---|
| ttl | 각 요청이 스토리지에서 지속되는 시간(초) |
| limit | TTL 한도 내 최대 요청 횟수 |
| 무시 사용자 에이전트 | 에 관해서는 무시할 사용자 에이전트의 정규식 배열입니다.<br>스로틀링 요청 |
| 저장소 | 요청을 추적하는 방법에 대한 저장소 설정 |

## 비동기 구성

속도 제한 구성을 동기식이 아닌 비동기식으로 가져오고 싶을 수도 있습니다. 의존성 주입 및 비동기 메서드를 허용하는 forRootAsync() 메서드를 사용할 수 있습니다.

한 가지 접근 방식은 팩토리 함수를 사용하는 것입니다:

```
모듈({ import:
  [
    ThrottlerModule.forRootAsync({
      import: [ConfigModule],
      inject: [ConfigService],
      useFactory: (config: ConfigService) =>
        ({ ttl: config.get('THROTTLE_TTL'),
        limit: config.get('THROTTLE_LIMIT'),
      }),
    }),
  ],
})
내보내기 클래스 AppModule {}
```

`useClass` 구문을 사용할 수도 있습니다:

```
모듈({ import:
  [
    ThrottlerModule.forRootAsync({
```

```
        임포트합니다: [ConfigModule],
        useClass: 스로틀러컨피그서비스,
      }),
    ],
  })
  내보내기 클래스 AppModule {}
```

ThrottlerConfigService가 인터페이스를 구현하는 한 이 작업을 수행할 수 있습니다.

`스로틀러옵션팩토리`. 저장소

내장 스토리지는 메모리 내 캐시로, 요청이 특정 기준을 통과할 때까지 요청을 추적합니다.
글로벌 옵션으로 설정한 TTL. `저장소` 옵션에 고유한 저장소 옵션을 추가할 수 있습니다.

`ThrottlerStorage` 인터페이스를 구현하는 클래스만 있으면 됩니다.

분산 서버의 경우 Redis용 커뮤니티 스토리지 공급자를 사용하여 단일 데이터 소스를 확보할 수 있습니다.

> 정보 참고 ThrottlerStorage는 `@nestjs/throttler에서` 가져올 수 있습니다.