게이트웨이

종속성 주입, 데코레이터, 예외 필터, 파이프, 가드, 인터셉터 등 이 문서의 다른 곳에서 설명하는 대부분의 개념은 게이트웨이에도 동일하게 적용됩니다. Nest는 가능한 경우 구현 세부 사항을 추상화하여 동일한 구성 요소가 HTTP 기반 플랫폼, 웹 소켓 및 마이크로서비스에서 실행될 수 있도록 합니다. 이 섹션에서는 웹소켓에 특화된 Nest의 측면을 다룹니다.

Nest에서 게이트웨이는 `@WebSocketGateway()` 데코레이터로 주석을 단 클래스일 뿐입니다. 기술적으로 게이트웨이는 플랫폼에 구애받지 않으므로 어댑터가 생성되면 모든 WebSockets 라이브러리와 호환됩니다. 기본적으로 지원되는 WS 플랫폼은 socket.io와 ws의 두 가지입니다. 필요에 가장 적합한 것을 선택할 수 있습니다. 또한 이 가이드에 따라 직접 어댑터를 만들 수도 있습니다.

> 정보 힌트 게이트웨이는 프로바이더로 취급될 수 있으며, 이는 클래스 생성자를 통해 종속성을 주입할 수 있음을 의미합니다. 또한 게이트웨이는 다른 클래스(프로바이더 및 컨트롤러)에서도 주입할 수 있습니다.

설치

웹소켓 기반 애플리케이션 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
@@파일명()
npm i --save @nestjs/웹소켓 @nestjs/플랫폼-소켓.io @@switch
npm i --save @nestjs/웹소켓 @nestjs/플랫폼-소켓.io
```

개요

일반적으로 앱이 웹 애플리케이션이 아니거나 포트를 수동으로 변경하지 않는 한 각 게이트웨이는 HTTP 서버와 동일한 포트에서 수신 대기합니다. 이 기본 동작은 `80이` 선택한 포트 번호인 `@WebSocketGateway(80)` 데코레이터에 인수를 전달하여 수정할 수 있습니다. 다음 구성을 사용하여 게이트웨이에서 사용하는 네임스페이스를 설정할 수도 있습니다:

```
웹소켓게이트웨이(80, { 네임스페이스: '이벤트' })
```

> 경고 경고 게이트웨이는 기존 모듈의 공급자 배열에서 참조될 때까지 인스턴스화되지 않습니다.

지원되는 모든 옵션을 소켓 생성자에 두 번째 인자로 전달할 수 있습니다.

`WebSocketGateway()` 데코레이터를 추가합니다:

```
웹소켓게이트웨이(81, { 트랜스포트: ['웹소켓'] })
```

이제 게이트웨이가 수신 대기 중이지만 아직 수신 메시지를 구독하지 않았습니다. 이벤트 메시지를 구독하고 정확히 동일한 데이터로 사용자에게 응답하는 핸들러를 만들어 보겠습니다.

```
@@파일명(events.gateway) @SubscribeMessage('events')
handleEvent(@MessageBody() 데이터: 문자열): 문자열 {
  데이터를 반환합니다;
}
@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
  데이터를 반환합니다;
}
```

> 정보 힌트 @SubscribeMessage() 및 @MessageBody() 데코레이터는 다음에서 가져옵니다.

nestjs/websockets 패키지.

게이트웨이가 생성되면 모듈에 등록할 수 있습니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./events.gateway'에서 { EventsGateway }를 가져옵니다;

@@파일명(events.module)
@Module({
  공급자: [이벤트 게이트웨이]
})
이벤트 모듈 클래스 {} 내보내기
```

또한 데코레이터에 속성 키를 전달하여 수신 메시지 본문에서 속성 키를 추출할 수도 있습니다:

```
@@파일명(events.gateway) @SubscribeMessage('events')
handleEvent(@MessageBody('id') id: number): number {
  // id === messageBody.id
  반환 id;
}
@@switch
@Bind(MessageBody('id'))
@SubscribeMessage('events')
handleEvent(id) {
  // id === messageBody.id
  반환 id;
}
```

데코레이터를 사용하지 않으려면 다음 코드가 기능적으로 동일합니다:

```
@@파일명(events.gateway) @SubscribeMessage('events')
handleEvent(client: Socket, data: 문자열): 문자열 { 반
  환 데이터;
}
@@switch
@SubscribeMessage('events')
handleEvent(client, data) {
  데이터를 반환합니다;
}
```

위의 예에서 `handleEvent()` 함수는 두 개의 인수를 받습니다. 첫 번째 인수는 플랫폼별 소켓 인스턴스이고, 두 번째 인수는 클라이언트로부터 받은 데이터입니다. 하지만 이 접근 방식은 각 단위 테스트에서 `소켓` 인스턴스를 모킹해야 하므로 권장되지 않습니다.

`이벤트` 메시지가 수신되면 핸들러는 네트워크를 통해 전송된 것과 동일한 데이터가 포함된 확인을 보냅니다. 또한 `클라이언트.emit()` 메서드를 사용하는 등 라이브러리별 접근 방식을 사용하여 메시지를 전송할 수도 있습니다. 연결된 소켓 인스턴스에 액세스하려면 `@ConnectedSocket()` 데코레이터를 사용합니다.

```
@@파일명(events.gateway)
@SubscribeMessage('events')
handleEvent(
  메시지 본문() 데이터: 문자열, @커넥티드 소켓() 클
  라이언트: Socket,
): 문자열 { 데이터를
  반환합니다;
}
@@switch
바인드(메시지바디(), 커넥티드소켓()) 구독 메시지('이
벤트') 핸들 이벤트(데이터, 클라이언트) {
  데이터를 반환합니다;
```

> 정보 힌트 `@ConnectedSocket()` 데코레이터는 `@nestjs/websockets` 패키지에서 가져옵니다.

하지만 이 경우 인터셉터를 활용할 수 없습니다. 사용자에게 응답하지 않으려면 `반환` 문을 건너뛰거나 명시적으로 "거짓" 값(예: `정의되지 않음`)을 반환하면 됩니다.

이제 클라이언트가 다음과 같은 메시지를 전송합니다:

```
socket.emit('events', { name: 'Nest' });
```

`handleEvent()` 메서드가 실행됩니다. 위의 핸들러 내에서 전송된 메시지를 수신하려면 클라이언트는 해당 수신 확인 리스너를 첨부해야 합니다:

```
socket.emit('events', { name: 'Nest' }, (data) => console.log(data));
```

다중 응답

승인은 한 번만 발송됩니다. 또한 네이티브 웹소켓 구현에서는 지원되지 않습니다. 이 제한을 해결하기 위해 두 가지 프로퍼티로 구성된 객체를 반환할 수 있습니다. 방출된 이벤트의 이름인 이벤트와 클라이언트에 전달해야 하는 데이터입니다.

```
@@파일명(events.gateway) @SubscribeMessage('events')
handleEvent(@MessageBody() 데이터: 알 수 없음): WsResponse<unknown> {
  const event = 'events';
  반환 { 이벤트, 데이터 };
}
@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
  const event = 'events';
  return { event, data };
}
```

> 정보 힌트 `WsResponse` 인터페이스는 `@nestjs/websockets` 패키지에서 가져옵니다.

> 경고 경고 `데이터` 필드가 일반 JavaScript 객체 응답을 무시하므로 `데이터` 필드가 `ClassSerializerInterceptor에` 의존하는 경우 `WsResponse를` 구현하는 클래스 인스턴스를 반환해야 합니다.

클라이언트는 수신 응답을 수신 대기하기 위해 다른 이벤트 리스너를 적용해야 합니다.

```
socket.on('events', (data) => console.log(data));
```

비동기 응답

메시지 핸들러는 동기식 또는 비동기식으로 응답할 수 있습니다. 따라서 `비동기` 메서드가 지원됩니다. 또한 메시지 핸들러는 `Observable을` 반환할 수 있으며, 이 경우 스트림이 완료될 때까지 결과값이 방출됩니다.

```
@@파일명(events.gateway) @SubscribeMessage('events')

onEvent(@MessageBody() 데이터: 알 수 없음): Observable<WsResponse<number>> {
  const event = 'events';
  const response = [1, 2, 3];

  return from(response).pipe(
    map(data => ({ event, data })),
```

```
    );
  }
  @@switch
  @Bind(MessageBody())
  @SubscribeMessage('events')
  onEvent(data) {
    const event = 'events';
    const response = [1, 2, 3];

    return from(response).pipe(
      map(data => ({ event, data })),
    );
  }
```

위의 예에서 메시지 처리기는 배열의 각 항목에 대해 3번 응답합니다. 라이프사이클 훅

유용한 라이프사이클 후크는 3가지가 있습니다. 모두 해당 인터페이스가 있으며 다음 문서에 설명되어 있습니다. 다음 표를 참조하세요:

`OnGatewayInit`	`afterInit()` 메서드를 강제로 구현합니다. 라이브러리별 서버 인스턴스를 인수로 받습니다(필요한 경우 나머지는 스프레드합니다).
온게이트웨이 연결	핸들커넥션() 메서드를 구현하도록 강제합니다. 라이브러리별 클라이언트 소켓 인스턴스를 인자로 받습니다.
`OnGatewayDisconnect`	`handleDisconnect()` 메서드를 구현하도록 강제합니다. 라이브러리별 클라이언트 소켓 인스턴스를 인자로 받습니다.

> 정보 힌트 각 라이프사이클 인터페이스는 `@nestjs/websockets` 패키지에서 노출됩니다.

서버

때로는 네이티브 플랫폼별 서버 인스턴스에 직접 액세스하고 싶을 때가 있습니다. 이 객체에 대한 참조는 `afterInit()` 메서드(`OnGatewayInit` 인터페이스)에 인수로 전달됩니다. 또 다른 옵션은 `@WebSocketServer()` 데코레이터를 사용하는 것입니다.

```
@WebSocketServer() 서버:
서버;
```

> 경고 `@WebSocketServer()` 데코레이터는 `@nestjs/websockets에서` 가져온 것입니다. 패키지입니다.

Nest는 서버 인스턴스를 사용할 준비가 되면 이 프로퍼티에 서버 인스턴스를 자동으로 할당합니다.

예시

작동하는 예제는 여기에서 확인할 수 있습니다.

예외 필터

HTTP 예외 필터 계층과 해당 웹 소켓 계층의 유일한 차이점은 `HttpException`을 던지는 대신 `WsException`을 사용해야 한다는 점입니다.

```
새로운 WsException('잘못된 자격 증명입니다.')을 던집니다;
```

> 정보 힌트 `WsException` 클래스는 `@nestjs/websockets` 패키지에서 가져옵니다.

위의 샘플을 사용하면 Nest는 던져진 예외를 처리하고 다음과 같은 구조의 `예외` 메시지를 내보냅니다:

```
{
  상태: '오류',
  메시지가 표시됩니다: '잘못된 자격 증명입니다.
}
```

필터

웹 소켓 예외 필터는 HTTP 예외 필터와 동일하게 작동합니다. 다음 예제에서는 수동으로 인스턴스화된 메서드 범위 필터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 필터를 사용할 수도 있습니다(즉, 게이트웨이 클래스 앞에 `@UseFilters()` 데코레이터를 붙이면 됩니다).

```
사용필터(새로운 WsExceptionFilter()) 구독 메시지('이벤트')
onEvent(클라이언트, 데이터: any): WsResponse<any>
  { const event = 'events';
  반환 { 이벤트, 데이터 };
}
```

상속

일반적으로 애플리케이션 요구 사항을 충족하도록 완전히 사용자 정의된 예외 필터를 만듭니다. 그러나 핵심 예외 필터를 단순히 확장하고 특정 요인에 따라 동작을 재정의하려는 사용 사례가 있을 수 있습니다.

예외 처리를 기본 필터에 위임하려면 `BaseWsExceptionFilter`를 확장해야 합니다.

를 생성하고 상속된 `catch()` 메서드를 호출합니다.

```
@@파일명()
'@nestjs/common'에서 { Catch, ArgumentsHost }를 가져오고,
'@nestjs/websockets'에서 { BaseWsExceptionFilter }를 가져옵니다
;
```

```
@Catch()
export class AllExceptionsFilter extends BaseWsExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(예외, 호스트);
  }
}
@@switch
'@nestjs/common'에서 { Catch }를 가져옵니다;
'@nestjs/websockets'에서 { BaseWsExceptionFilter }를 가져옵니다;

@Catch()
export class AllExceptionsFilter extends BaseWsExceptionFilter {
  catch(exception, host) {
    super.catch(예외, 호스트);
  }
}
```

위의 구현은 접근 방식을 보여주는 셸일 뿐입니다. 확장 예외 필터의 구현에는 맞춤형 비즈니스 로직(예: 다양한 조건 처리)이 포함될 수 있습니다.

파이프

일반 파이프와 웹 소켓 파이프 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException`을 던지는 대신 `WsException`을 사용해야 한다는 것입니다. 또한 모든 파이프는 `데이터` 매개변수에만 적용됩니다(`클라이언트` 인스턴스의 유효성을 검사하거나 변환하는 것은 쓸모가 없으므로).

> 정보 힌트 `WsException` 클래스는 `@nestjs/websockets` 패키지에서 노출됩니다.

바인딩 파이프

다음 예제는 수동으로 인스턴스화된 메서드 범위 파이프를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 파이프를 사용할 수도 있습니다(즉, 게이트웨이 클래스에 `@UsePipes()` 데코레이터를 접두사로 붙이면 됩니다).

```
@@파일명()
사용 파이프(새로운 유효성 검사 파이프()) 구독 메시지('이벤트')
핸들 이벤트(클라이언트: 클라이언트, 데이터: 알 수 없음): WsResponse<unknown>
  { const event = 'events';
  반환 { 이벤트, 데이터 };
}
@@switch
사용파이프(새로운 유효성 검사 파이프()) 구독 메시지('
이벤트') 핸들 이벤트(클라이언트, 데이터) {
  const event = 'events';
  return { event, data };
}
```

경비병

웹 소켓 가드와 일반 HTTP 애플리케이션 가드 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException을` 던지는 대신 `WsException을` 사용해야 한다는 점입니다.

> 정보 힌트 `WsException` 클래스는 `@nestjs/websockets` 패키지에서 노출됩니다.

바인딩 가드

다음 예제는 메서드 범위 가드를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 가드를 사용할 수도 있습니다(즉, 게이트웨이 클래스 앞에 `@UseGuards()` 데코레이터를 붙이면 됩니다).

```
파일명() @사용가드() 인증가드()

@가입메시지('이벤트')

핸들 이벤트(클라이언트: 클라이언트, 데이터: 알 수 없음): WsResponse<unknown>
  { const event = 'events';
  반환 { 이벤트, 데이터 };
}
스위치 @사용가드(AuthGuard)
@SubscribeMessage('events')
핸들이벤트(클라이언트, 데이터) {
  const event = 'events';
  return { event, data };
}
```

인터셉터

일반 인터셉터와 웹 소켓 인터셉터 사이에는 차이가 없습니다. 다음 예제에서는 수동으로 인스턴스화된 메서드 범위 인터셉터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 인터셉터도 사용할 수 있습니다(즉, 게이트웨이 클래스 앞에 `@UseInterceptors()`

데코레이터).

```
@@파일명()
사용 인터셉터(새로운 트랜스폼 인터셉터()) 구독 메시지('이벤트')

핸들 이벤트(클라이언트: 클라이언트, 데이터: 알 수 없음): WsResponse<unknown>
  { const event = 'events';
  반환 { 이벤트, 데이터 };
}
@@switch
사용 인터셉터(새로운 트랜스폼인터셉터())
@SubscribeMessage('events')
handleEvent(client, data) {
  const event = 'events';
  return { event, data };
}
```

어댑터

WebSockets 모듈은 플랫폼에 구애받지 않으므로 `WebSocketAdapter` 인터페이스를 사용하여 자체 라이브러리(또는 네이티브 구현)를 가져올 수 있습니다. 이 인터페이스는 다음 표에 설명된 몇 가지 메서드를 구현하도록 강제합니다:

`create`	전달된 인수를 기반으로 소켓 인스턴스를 생성합니다
`bindClientConnect`	클라이언트 연결 이벤트를 `바인딩합니다.` 클라이언트

연결 해제 이벤트를 바인딩합니다(선택 사항*).

`바인드메시지 핸들러`	수신 메시지를 해당 메시지 핸들러에 바인딩합니다.
`close`	서버 인스턴스 종료 소켓.io

확장

socket.io 패키지는 `IoAdapter` 클래스로 래핑됩니다. 기본 기능을 향상시키려면 어떻게 해야 하나요?
기능이 필요한가요? 예를 들어, 기술 요구 사항에 따라 부하가 분산된 여러 웹 서비스 인스턴스에 걸쳐 이벤트를 브로드캐스트하는 기능이 필요합니다. 이를 위해 `IoAdapter를` 확장하고 새 socket.io 서버를 인스턴스화하는 단일 메서드를 재정의할 수 있습니다. 하지만 먼저 필요한 패키지를 설치해 보겠습니다.

> 경고 여러 로드 밸런싱 인스턴스에서 socket.io를 사용하려면 `전송을` 설정하여 폴링을 비활성화해야 합니다: `['websocket']`을 설정하여 폴링을 비활성화하거나 로드 밸런서에서 쿠키 기반 라우팅을 활성화해야 합니다. Redis만으로는 충분하지 않습니다. 자세한 내용은 여기를 참조하세요.

```
$ npm i --save redis socket.io @socket.io/redis-adapter
```

패키지가 설치되면 `RedisIoAdapter` 클래스를 생성할 수 있습니다.

```
'@nestjs/platform-socket.io'에서 { IoAdapter }를 임포트하고,
'socket.io'에서 { ServerOptions }를 임포트합니다;
'@socket.io/redis-adapter'에서 { createAdapter }를 가져오고,
'redis'에서 { createClient }를 가져옵니다;

내보내기 클래스 RedisIoAdapter extends IoAdapter {
  비공개 어댑터 생성자: 반환 유형<생성 어댑터 유형>;

  비동기 connectToRedis(): Promise<void> {
    const pubClient = createClient({ url: `redis://localhost:6379` });
    const subClient = pubClient.duplicate();

    await Promise.all([pubClient.connect(), subClient.connect()]);

    this.adapterConstructor = createAdapter(pubClient, subClient);
  }
```

```
  createIOServer(port: number, options?: ServerOptions): any {
    const server = super.createIOServer(port, options);
    server.adapter(this.adapterConstructor);
    서버를 반환합니다;
  }
}
```

그런 다음 새로 생성한 Redis 어댑터로 전환하기만 하면 됩니다.

```
const app = await NestFactory.create(AppModule);
const redisIoAdapter = new RedisIoAdapter(app);
await redisIoAdapter.connectToRedis();

app.useWebSocketAdapter(redisIoAdapter);
```

Ws 라이브러리

사용 가능한 또 다른 어댑터는 프레임워크 사이에서 프록시처럼 작동하고 매우 빠르고 철저하게 테스트된 ws 라이브러리를 통합하는 `WsAdapter입니다.` 이 어댑터는 네이티브 브라우저 웹소켓과 완벽하게 호환되며 socket.io 패키지보다 훨씬 빠릅니다. 안타깝게도 바로 사용할 수 있는 기능이 훨씬 적습니다. 하지만 경우에 따라서는 꼭 필요하지 않을 수도 있습니다.

> 정보 힌트 `ws` 라이브러리는 네임스페이스(`socket.io에서` 널리 사용되는 통신 채널)를 지원하지 않습니다. 그러나 어떻게든 이 기능을 모방하기 위해 서로 다른 경로에 여러 `ws` 서버를 마운트할 수 있습니다(예시: `WebSocketGateway({{ '{' }} 경로: '/users' {{ '}' }})`).

ws를 사용하려면 먼저 필요한 패키지를 설치해야 합니다:

```
npm i --save @nestjs/platform-ws
```

패키지가 설치되면 어댑터를 전환할 수 있습니다:

```
const app = await NestFactory.create(AppModule);
app.useWebSocketAdapter(new WsAdapter(app));
```

> 정보 힌트 WsAdapter는 `@nestjs/platform-ws에서` 가져옵니다.

고급(사용자 지정 어댑터)

데모 목적으로 ws 라이브러리를 수동으로 통합해 보겠습니다. 앞서 언급했듯이 이 라이브러리에 대한 어댑터는 이미 생성되어 있으며 `@nestjs/platform-ws` 패키지에서 `WsAdapter` 클래스로 노출되어 있습니다. 다음은 단순화된 구현의 잠재적 모습입니다:

```
@@파일명(ws-adapter)
'ws'에서 WebSocket으로 *를 가져옵니다;
'@nestjs/common'에서 { WebSocketAdapter, INestApplicationContext }를
가져옵니다;
'@nestjs/websockets'에서 { MessageMappingProperties }를 가져오고,
'rxjs'에서 { Observable, fromEvent, EMPTY }를 가져옵니다;
'rxjs/operators'에서 { mergeMap, filter }를 가져옵니다;

export class WsAdapter implements WebSocketAdapter {
  constructor(private app: INestApplicationContext) {}

  create(port: number, options: any = {}): any {
    return new WebSocket.Server({ port, ...options });
  }

  bindClientConnect(server, callback: Function) {
    server.on('connection', callback);
  }

  바인드메시지핸들러( 클라이언트:
    WebSocket,
    핸들러를 사용합니다: MessageMappingProperties[],
    프로세스: (데이터: any) => Observable<any>,
  ) {
    fromEvent(client, 'message')
      .pipe(
        mergeMap(data => this.bindMessageHandler(데이터, 핸들러, 프로세
스)),
        filter(결과 => 결과),
      )
      .subscribe(응답 => client.send(JSON.stringify(응답)));
  }

  bindMessageHandler(
    buffer,
    핸들러를 사용합니다: MessageMappingProperties[],
    프로세스: (데이터: any) => Observable<any>,
  ): 관찰가능<any> {
    const message = JSON.parse(buffer.data);
    const messageHandler = handlers.find(
      핸들러 => 핸들러.메시지 === 메시지.이벤트,
    );
    if (!messageHandler) {
      return EMPTY;
    }
```

```
      반환 프로세스(메시지핸들러.콜백(메시지.데이터));
    }

    close(server) {
      server.close();
    }
  }
```

> 정보 힌트 ws 라이브러리를 활용하려면 자체 라이브러리를 만드는 대신 기본 제공되는 `WsAdapter`를 사용하세요.

그런 다음 `사용WebSocketAdapter()` 메서드를 사용하여 사용자 정의 어댑터를 설정할 수 있습니다:

```
@@파일명(메인)
const app = await NestFactory.create(AppModule);
app.useWebSocketAdapter(new WsAdapter(app));
```

예

WsAdapter를 사용하는 작업 예제는 여기에서 확인할 수 있습니다.