Gateways

Most of the concepts discussed elsewhere in this documentation, such as dependency injection, decorators, exception filters, pipes, guards and interceptors, apply equally to gateways. Wherever possible, Nest abstracts implementation details so that the same components can run across HTTP-based platforms, WebSockets, and Microservices. This section covers the aspects of Nest that are specific to WebSockets.

In Nest, a gateway is simply a class annotated with `@WebSocketGateway()` decorator. Technically, gateways are platform-agnostic which makes them compatible with any WebSockets library once an adapter is created. There are two WS platforms supported out-of-the-box: socket.io and ws. You can choose the one that best suits your needs. Also, you can build your own adapter by following this guide.

> info **Hint** Gateways can be treated as providers; this means they can inject dependencies through the class constructor. Also, gateways can be injected by other classes (providers and controllers) as well.

Installation

To start building WebSockets-based applications, first install the required package:

```
@@filename()
$ npm i --save @nestjs/websockets @nestjs/platform-socket.io
@@switch
$ npm i --save @nestjs/websockets @nestjs/platform-socket.io
```

Overview

In general, each gateway is listening on the same port as the **HTTP server**, unless your app is not a web application, or you have changed the port manually. This default behavior can be modified by passing an argument to the `@WebSocketGateway(80)` decorator where `80` is a chosen port number. You can also set a namespace used by the gateway using the following construction:

```
@WebSocketGateway(80, { namespace: 'events' })
```

> warning **Warning** Gateways are not instantiated until they are referenced in the providers array of an existing module.

You can pass any supported option to the socket constructor with the second argument to the `@WebSocketGateway()` decorator, as shown below:

```
@WebSocketGateway(81, { transports: ['websocket'] })
```

The gateway is now listening, but we have not yet subscribed to any incoming messages. Let's create a handler that will subscribe to the `events` messages and respond to the user with the exact same data.

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(@MessageBody() data: string): string {
  return data;
}
@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
  return data;
}
```

> info **Hint** `@SubscribeMessage()` and `@MessageBody()` decorators are imported from `@nestjs/websockets` package.

Once the gateway is created, we can register it in our module.

```
import { Module } from '@nestjs/common';
import { EventsGateway } from './events.gateway';

@@filename(events.module)
@Module({
  providers: [EventsGateway]
})
export class EventsModule {}
```

You can also pass in a property key to the decorator to extract it from the incoming message body:

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

If you would prefer not to use decorators, the following code is functionally equivalent:

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(client: Socket, data: string): string {
  return data;
}
@@switch
@SubscribeMessage('events')
handleEvent(client, data) {
  return data;
}
```

In the example above, the `handleEvent()` function takes two arguments. The first one is a platform-specific socket instance, while the second one is the data received from the client. This approach is not recommended though, because it requires mocking the `socket` instance in each unit test.

Once the `events` message is received, the handler sends an acknowledgment with the same data that was sent over the network. In addition, it's possible to emit messages using a library-specific approach, for example, by making use of `client.emit()` method. In order to access a connected socket instance, use `@ConnectedSocket()` decorator.

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(
  @MessageBody() data: string,
  @ConnectedSocket() client: Socket,
): string {
  return data;
}
@@switch
@Bind(MessageBody(), ConnectedSocket())
@SubscribeMessage('events')
handleEvent(data, client) {
  return data;
}
```

> info **Hint** `@ConnectedSocket()` decorator is imported from `@nestjs/websockets` package.

However, in this case, you won't be able to leverage interceptors. If you don't want to respond to the user, you can simply skip the `return` statement (or explicitly return a "falsy" value, e.g. `undefined`).

Now when a client emits the message as follows:

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

The `handleEvent()` method will be executed. In order to listen for messages emitted from within the above handler, the client has to attach a corresponding acknowledgment listener:

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

Multiple responses

The acknowledgment is dispatched only once. Furthermore, it is not supported by native WebSockets implementation. To solve this limitation, you may return an object which consists of two properties. The `event` which is a name of the emitted event and the `data` that has to be forwarded to the client.

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(@MessageBody() data: unknown): WsResponse<unknown> {
  const event = 'events';
  return { event, data };
}
@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
  const event = 'events';
  return { event, data };
}
```

> info **Hint** The `WsResponse` interface is imported from `@nestjs/websockets` package.

> warning **Warning** You should return a class instance that implements `WsResponse` if your `data` field relies on `ClassSerializerInterceptor`, as it ignores plain JavaScript object responses.

In order to listen for the incoming response(s), the client has to apply another event listener.

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

Asynchronous responses

Message handlers are able to respond either synchronously or **asynchronously**. Hence, `async` methods are supported. A message handler is also able to return an `Observable`, in which case the result values will be emitted until the stream is completed.

```
@@filename(events.gateway)
@SubscribeMessage('events')
onEvent(@MessageBody() data: unknown): 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 })),
    );
  }
```

In the example above, the message handler will respond **3 times** (with each item from the array).

Lifecycle hooks

There are 3 useful lifecycle hooks available. All of them have corresponding interfaces and are described in the following table:

`OnGatewayInit`	Forces to implement the `afterInit()` method. Takes library-specific server instance as an argument (and spreads the rest if required).
`OnGatewayConnection`	Forces to implement the `handleConnection()` method. Takes library-specific client socket instance as an argument.
`OnGatewayDisconnect`	Forces to implement the `handleDisconnect()` method. Takes library-specific client socket instance as an argument.

> info **Hint** Each lifecycle interface is exposed from `@nestjs/websockets` package.

Server

Occasionally, you may want to have a direct access to the native, **platform-specific** server instance. The reference to this object is passed as an argument to the `afterInit()` method (`OnGatewayInit` interface). Another option is to use the `@WebSocketServer()` decorator.

```
@WebSocketServer()
server: Server;
```

> warning **Notice** The `@WebSocketServer()` decorator is imported from the `@nestjs/websockets` package.

Nest will automatically assign the server instance to this property once it is ready to use.

Example

A working example is available here.

Exception filters

The only difference between the HTTP exception filter layer and the corresponding web sockets layer is that instead of throwing `HttpException`, you should use `WsException`.

```
throw new WsException('Invalid credentials.');
```

> info **Hint** The `WsException` class is imported from the `@nestjs/websockets` package.

With the sample above, Nest will handle the thrown exception and emit the `exception` message with the following structure:

```
{
  status: 'error',
  message: 'Invalid credentials.'
}
```

Filters

Web sockets exception filters behave equivalently to HTTP exception filters. The following example uses a manually instantiated method-scoped filter. Just as with HTTP based applications, you can also use gateway-scoped filters (i.e., prefix the gateway class with a `@UseFilters()` decorator).

```
@UseFilters(new WsExceptionFilter())
@SubscribeMessage('events')
onEvent(client, data: any): WsResponse<any> {
  const event = 'events';
  return { event, data };
}
```

Inheritance

Typically, you'll create fully customized exception filters crafted to fulfill your application requirements. However, there might be use-cases when you would like to simply extend the **core exception filter**, and override the behavior based on certain factors.

In order to delegate exception processing to the base filter, you need to extend `BaseWsExceptionFilter` and call the inherited `catch()` method.

```
@@filename()
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseWsExceptionFilter } from '@nestjs/websockets';
```

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

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

The above implementation is just a shell demonstrating the approach. Your implementation of the extended exception filter would include your tailored **business logic** (e.g., handling various conditions).

Pipes

There is no fundamental difference between regular pipes and web sockets pipes. The only difference is that instead of throwing `HttpException`, you should use `WsException`. In addition, all pipes will be only applied to the `data` parameter (because validating or transforming `client` instance is useless).

> info **Hint** The `WsException` class is exposed from `@nestjs/websockets` package.

Binding pipes

The following example uses a manually instantiated method-scoped pipe. Just as with HTTP based applications, you can also use gateway-scoped pipes (i.e., prefix the gateway class with a `@UsePipes()` decorator).

```
@@filename()
@UsePipes(new ValidationPipe())
@SubscribeMessage('events')
handleEvent(client: Client, data: unknown): WsResponse<unknown> {
  const event = 'events';
  return { event, data };
}
@@switch
@UsePipes(new ValidationPipe())
@SubscribeMessage('events')
handleEvent(client, data) {
  const event = 'events';
  return { event, data };
}
```

Guards

There is no fundamental difference between web sockets guards and regular HTTP application guards. The only difference is that instead of throwing `HttpException`, you should use `WsException`.

> info **Hint** The `WsException` class is exposed from `@nestjs/websockets` package.

Binding guards

The following example uses a method-scoped guard. Just as with HTTP based applications, you can also use gateway-scoped guards (i.e., prefix the gateway class with a `@UseGuards()` decorator).

```
@@filename()
@UseGuards(AuthGuard)
@SubscribeMessage('events')
handleEvent(client: Client, data: unknown): WsResponse<unknown> {
  const event = 'events';
  return { event, data };
}
@@switch
@UseGuards(AuthGuard)
@SubscribeMessage('events')
handleEvent(client, data) {
  const event = 'events';
  return { event, data };
}
```

Interceptors

There is no difference between regular interceptors and web sockets interceptors. The following example uses a manually instantiated method-scoped interceptor. Just as with HTTP based applications, you can also use gateway-scoped interceptors (i.e., prefix the gateway class with a `@UseInterceptors()` decorator).

```
@@filename()
@UseInterceptors(new TransformInterceptor())
@SubscribeMessage('events')
handleEvent(client: Client, data: unknown): WsResponse<unknown> {
  const event = 'events';
  return { event, data };
}
@@switch
@UseInterceptors(new TransformInterceptor())
@SubscribeMessage('events')
handleEvent(client, data) {
  const event = 'events';
  return { event, data };
}
```

Adapters

The WebSockets module is platform-agnostic, hence, you can bring your own library (or even a native implementation) by making use of `WebSocketAdapter` interface. This interface forces to implement few methods described in the following table:

`create`	Creates a socket instance based on passed arguments
`bindClientConnect`	Binds the client connection event
`bindClientDisconnect`	Binds the client disconnection event (optional*)
`bindMessageHandlers`	Binds the incoming message to the corresponding message handler
`close`	Terminates a server instance

Extend socket.io

The socket.io package is wrapped in an `IoAdapter` class. What if you would like to enhance the basic functionality of the adapter? For instance, your technical requirements require a capability to broadcast events across multiple load-balanced instances of your web service. For this, you can extend `IoAdapter` and override a single method which responsibility is to instantiate new socket.io servers. But first of all, let's install the required package.

> warning **Warning** To use socket.io with multiple load-balanced instances you either have to disable polling by setting `transports: ['websocket']` in your clients socket.io configuration or you have to enable cookie based routing in your load balancer. Redis alone is not enough. See here for more information.

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

Once the package is installed, we can create a `RedisIoAdapter` class.

```
import { IoAdapter } from '@nestjs/platform-socket.io';
import { ServerOptions } from 'socket.io';
import { createAdapter } from '@socket.io/redis-adapter';
import { createClient } from 'redis';

export class RedisIoAdapter extends IoAdapter {
  private adapterConstructor: ReturnType<typeof createAdapter>;

  async 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);
    return server;
  }
}
```

Afterward, simply switch to your newly created Redis adapter.

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

app.useWebSocketAdapter(redisIoAdapter);
```

Ws library

Another available adapter is a `WsAdapter` which in turn acts like a proxy between the framework and integrate blazing fast and thoroughly tested ws library. This adapter is fully compatible with native browser WebSockets and is far faster than socket.io package. Unluckily, it has significantly fewer functionalities available out-of-the-box. In some cases, you may just don't necessarily need them though.

> info **Hint** `ws` library does not support namespaces (communication channels popularised by `socket.io`). However, to somehow mimic this feature, you can mount multiple `ws` servers on different paths (example: `@WebSocketGateway({{ '{' }} path: '/users' {{ '}' }})`).

In order to use `ws`, we firstly have to install the required package:

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

Once the package is installed, we can switch an adapter:

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

> info **Hint** The `WsAdapter` is imported from `@nestjs/platform-ws`.

Advanced (custom adapter)

For demonstration purposes, we are going to integrate the ws library manually. As mentioned, the adapter for this library is already created and is exposed from the `@nestjs/platform-ws` package as a `WsAdapter` class. Here is how the simplified implementation could potentially look like:

```
@@filename(ws-adapter)
import * as WebSocket from 'ws';
import { WebSocketAdapter, INestApplicationContext } from
'@nestjs/common';
import { MessageMappingProperties } from '@nestjs/websockets';
import { Observable, fromEvent, EMPTY } from 'rxjs';
import { mergeMap, filter } from 'rxjs/operators';

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);
  }

  bindMessageHandlers(
    client: WebSocket,
    handlers: MessageMappingProperties[],
    process: (data: any) => Observable<any>,
  ) {
    fromEvent(client, 'message')
      .pipe(
        mergeMap(data => this.bindMessageHandler(data, handlers,
process)),
        filter(result => result),
      )
      .subscribe(response => client.send(JSON.stringify(response)));
  }

  bindMessageHandler(
    buffer,
    handlers: MessageMappingProperties[],
    process: (data: any) => Observable<any>,
  ): Observable<any> {
    const message = JSON.parse(buffer.data);
    const messageHandler = handlers.find(
      handler => handler.message === message.event,
    );
    if (!messageHandler) {
      return EMPTY;
    }
    return process(messageHandler.callback(message.data));
  }

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

> info **Hint** When you want to take advantage of ws library, use built-in `WsAdapter` instead of creating your own one.

Then, we can set up a custom adapter using `useWebSocketAdapter()` method:

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

Example

A working example that uses `WsAdapter` is available here.