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.

```
$ npm i --save socket.io-redis
```

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

```
import { IoAdapter } from '@nestjs/platform-socket.io';
import * as redisIoAdapter from 'socket.io-redis';

const redisAdapter = redisIoAdapter({ host: 'localhost', port: 6379 });

export class RedisIoAdapter extends IoAdapter {
  createIOServer(port: number, options?: any): any {
    const server = super.createIOServer(port, options);
```

```
    server.adapter(redisAdapter);
    return server;
  }
}
```

Afterward, simply switch to your newly created Redis adapter.

```
const app = await NestFactory.create(ApplicationModule);
app.useWebSocketAdapter(new RedisIoAdapter(app));
```

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.

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(ApplicationModule);
app.useWebSocketAdapter(new WsAdapter(app));
```

> **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:

ws-adapter.ts JS

```typescript
import * as WebSocket from 'ws';
import { WebSocketAdapter, MessageMappingProperties, INestApplicationContext } from '@nestjs/commor
import { Observable, fromEvent, empty } from 'rxjs';
import { mergeMap, filter, tap } from 'rxjs/operators';

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

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

```
}
```

> **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:

main.ts JS

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

Example

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

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more **here**.

Principal Sponsor

Sponsors / Partners

Become a sponsor