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.