Custom transporters

Nest provides a variety of **transporters** out-of-the-box, as well as an API allowing developers to build new custom transport strategies. Transporters enable you to connect components over a network using a pluggable communications layer and a very simple application-level message protocol (read full article).

> info **Hint** Building a microservice with Nest does not necessarily mean you must use the `@nestjs/microservices` package. For example, if you want to communicate with external services (let's say other microservices written in different languages), you may not need all the features provided by `@nestjs/microservice` library. In fact, if you don't need decorators (`@EventPattern` or `@MessagePattern`) that let you declaratively define subscribers, running a Standalone Application and manually maintaining connection/subscribing to channels should be enough for most use-cases and will provide you with more flexibility.

With a custom transporter, you can integrate any messaging system/protocol (including Google Cloud Pub/Sub, Amazon Kinesis, and others) or extend the existing one, adding extra features on top (for example, QoS for MQTT).

> info **Hint** To better understand how Nest microservices work and how you can extend the capabilities of existing transporters, we recommend reading the NestJS Microservices in Action and Advanced NestJS Microservices article series.

Creating a strategy

First, let's define a class representing our custom transporter.

```typescript
import { CustomTransportStrategy, Server } from '@nestjs/microservices';

class GoogleCloudPubSubServer
  extends Server
  implements CustomTransportStrategy {
  /**
   * This method is triggered when you run "app.listen()".
   */
  listen(callback: () => void) {
    callback();
  }

  /**
   * This method is triggered on application shutdown.
   */
  close() {}
}
```

> warning **Warning** Please, note we won't be implementing a fully-featured Google Cloud Pub/Sub server in this chapter as this would require diving into transporter specific technical details.

In our example above, we declared the `GoogleCloudPubSubServer` class and provided `listen()` and `close()` methods enforced by the `CustomTransportStrategy` interface. Also, our class extends the `Server` class imported from the `@nestjs/microservices` package that provides a few useful methods, for example, methods used by Nest runtime to register message handlers. Alternatively, in case you want to extend the capabilities of an existing transport strategy, you could extend the corresponding server class, for example, `ServerRedis`. Conventionally, we added the `"Server"` suffix to our class as it will be responsible for subscribing to messages/events (and responding to them, if necessary).

With this in place, we can now use our custom strategy instead of using a built-in transporter, as follows:

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(
  AppModule,
  {
    strategy: new GoogleCloudPubSubServer(),
  },
);
```

Basically, instead of passing the normal transporter options object with `transport` and `options` properties, we pass a single property, `strategy`, whose value is an instance of our custom transporter class.

Back to our `GoogleCloudPubSubServer` class, in a real-world application, we would be establishing a connection to our message broker/external service and registering subscribers/listening to specific channels in `listen()` method (and then removing subscriptions & closing the connection in the `close()` teardown method), but since this requires a good understanding of how Nest microservices communicate with each other, we recommend reading this article series. In this chapter instead, we'll focus on the capabilities the `Server` class provides and how you can leverage them to build custom strategies.

For example, let's say that somewhere in our application, the following message handler is defined:

```
@MessagePattern('echo')
echo(@Payload() data: object) {
  return data;
}
```

This message handler will be automatically registered by Nest runtime. With `Server` class, you can see what message patterns have been registered and also, access and execute the actual methods that were assigned to them. To test this out, let's add a simple `console.log` inside `listen()` method before `callback` function is called:

```
listen(callback: () => void) {
  console.log(this.messageHandlers);
  callback();
}
```

After your application restarts, you'll see the following log in your terminal:

```
Map { 'echo' => [AsyncFunction] { isEventHandler: false } }
```

> info **Hint** If we used the `@EventPattern` decorator, you would see the same output, but with the `isEventHandler` property set to `true`.

As you can see, the `messageHandlers` property is a `Map` collection of all message (and event) handlers, in which patterns are being used as keys. Now, you can use a key (for example, `"echo"`) to receive a reference to the message handler:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  console.log(await echoHandler('Hello world!'));
  callback();
}
```

Once we execute the `echoHandler` passing an arbitrary string as an argument (`"Hello world!"` here), we should see it in the console:

```
Hello world!
```

Which means that our method handler was properly executed.

When using a `CustomTransportStrategy` with Interceptors the handlers are wrapped into RxJS streams. This means that you need to subscribe to them in order to execute the streams underlying logic (e.g. continue into the controller logic after an interceptor has been executed).

An example of this can be seen below:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  const streamOrResult = await echoHandler('Hello World');
  if (isObservable(streamOrResult)) {
    streamOrResult.subscribe();
  }
  callback();
}
```

Client proxy

As we mentioned in the first section, you don't necessarily need to use the `@nestjs/microservices` package to create microservices, but if you decide to do so and you need to integrate a custom strategy, you will need to provide a "client" class too.

> info **Hint** Again, implementing a fully-featured client class compatible with all `@nestjs/microservices` features (e.g., streaming) requires a good understanding of communication techniques used by the framework. To learn more, check out this article.

To communicate with an external service/emit & publish messages (or events) you can either use a library-specific SDK package, or implement a custom client class that extends the `ClientProxy`, as follows:

```
import { ClientProxy, ReadPacket, WritePacket } from
'@nestjs/microservices';

class GoogleCloudPubSubClient extends ClientProxy {
  async connect(): Promise<any> {}
  async close() {}
  async dispatchEvent(packet: ReadPacket<any>): Promise<any> {}
  publish(
    packet: ReadPacket<any>,
    callback: (packet: WritePacket<any>) => void,
  ): Function {}
}
```

> warning **Warning** Please, note we won't be implementing a fully-featured Google Cloud Pub/Sub client in this chapter as this would require diving into transporter specific technical details.

As you can see, `ClientProxy` class requires us to provide several methods for establishing & closing the connection and publishing messages (`publish`) and events (`dispatchEvent`). Note, if you don't need a request-response communication style support, you can leave the `publish()` method empty. Likewise, if you don't need to support event-based communication, skip the `dispatchEvent()` method.

To observe what and when those methods are executed, let's add multiple `console.log` calls, as follows:

```
class GoogleCloudPubSubClient extends ClientProxy {
  async connect(): Promise<any> {
    console.log('connect');
  }

  async close() {
    console.log('close');
  }

  async dispatchEvent(packet: ReadPacket<any>): Promise<any> {
    return console.log('event to dispatch: ', packet);
  }

  publish(
    packet: ReadPacket<any>,
    callback: (packet: WritePacket<any>) => void,
  ): Function {
    console.log('message:', packet);

    // In a real-world application, the "callback" function should be
executed
    // with payload sent back from the responder. Here, we'll simply
simulate (5 seconds delay)
    // that response came through by passing the same "data" as we've
originally passed in.
    setTimeout(() => callback({ response: packet.data }), 5000);

    return () => console.log('teardown');
  }
}
```

With this in place, let's create an instance of `GoogleCloudPubSubClient` class and run the `send()` method (which you might have seen in earlier chapters), subscribing to the returned observable stream.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .subscribe((response) => console.log(response));
```

Now, you should see the following output in your terminal:

```
connect
message: { pattern: 'pattern', data: 'Hello world!' }
Hello world! // <-- after 5 seconds
```

To test if our "teardown" method (which our `publish()` method returns) is properly executed, let's apply a timeout operator to our stream, setting it to 2 seconds to make sure it throws earlier then our `setTimeout` calls the `callback` function.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .pipe(timeout(2000))
  .subscribe(
    (response) => console.log(response),
    (error) => console.error(error.message),
  );
```

> info **Hint** The `timeout` operator is imported from the `rxjs/operators` package.

With `timeout` operator applied, your terminal output should look as follows:

```
connect
message: { pattern: 'pattern', data: 'Hello world!' }
```

```
teardown // <-- teardown
Timeout has occurred
```

To dispatch an event (instead of sending a message), use the `emit()` method:

```
googlePubSubClient.emit('event', 'Hello world!');
```

And that's what you should see in the console:

```
connect
event to dispatch:  { pattern: 'event', data: 'Hello world!' }
```

Message serialization

If you need to add some custom logic around the serialization of responses on the client side, you can use a custom class that extends the `ClientProxy` class or one of its child classes. For modifying successful requests you can override the `serializeResponse` method, and for modifying any errors that go through this client you can override the `serializeError` method. To make use of this custom class, you can pass the class itself to the `ClientsModule.register()` method using the `customClass` property. Below is an example of a custom `ClientProxy` that serializes each error into an `RpcException`.

```
@@filename(error-handling.proxy)
import { ClientTcp, RpcException } from '@nestjs/microservices';

class ErrorHandlingProxy extends ClientTCP {
  serializeError(err: Error) {
    return new RpcException(err);
  }
}
```

and then use it in the `ClientsModule` like so:

```
@@filename(app.module)
@Module({
  imports: [
    ClientsModule.register({
      name: 'CustomProxy',
      customClass: ErrorHandlingProxy,
    }),
  ]
})
export class AppModule
```

> info **hint** This is the class itself being passed to `customClass`, not an instance of the class. Nest will create the instance under the hood for you, and will pass any options given to the `options` property to the new `ClientProxy`.