Events

Event Emitter package (`@nestjs/event-emitter`) provides a simple observer implementation, allowing you to subscribe and listen for various events that occur in your application. Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other.

`EventEmitterModule` internally uses the eventemitter2 package.

Getting started

First install the required package:

```
$ npm i --save @nestjs/event-emitter
```

Once the installation is complete, import the `EventEmitterModule` into the root `AppModule` and run the `forRoot()` static method as shown below:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { EventEmitterModule } from '@nestjs/event-emitter';

@Module({
  imports: [
    EventEmitterModule.forRoot()
  ],
})
export class AppModule {}
```

The `.forRoot()` call initializes the event emitter and registers any declarative event listeners that exist within your app. Registration occurs when the `onApplicationBootstrap` lifecycle hook occurs, ensuring that all modules have loaded and declared any scheduled jobs.

To configure the underlying `EventEmitter` instance, pass the configuration object to the `.forRoot()` method, as follows:

```
EventEmitterModule.forRoot({
  // set this to `true` to use wildcards
  wildcard: false,
  // the delimiter used to segment namespaces
  delimiter: '.',
  // set this to `true` if you want to emit the newListener event
  newListener: false,
  // set this to `true` if you want to emit the removeListener event
  removeListener: false,
  // the maximum amount of listeners that can be assigned to an event
```

```
  maxListeners: 10,
  // show event name in memory leak message when more than maximum amount
of listeners is assigned
  verboseMemoryLeak: false,
  // disable throwing uncaughtException if an error event is emitted and
it has no listeners
  ignoreErrors: false,
});
```

Dispatching Events

To dispatch (i.e., fire) an event, first inject `EventEmitter2` using standard constructor injection:

```
constructor(private eventEmitter: EventEmitter2) {}
```

> info **Hint** Import the `EventEmitter2` from the `@nestjs/event-emitter` package.

Then use it in a class as follows:

```
this.eventEmitter.emit(
  'order.created',
  new OrderCreatedEvent({
    orderId: 1,
    payload: {},
  }),
);
```

Listening to Events

To declare an event listener, decorate a method with the `@OnEvent()` decorator preceding the method definition containing the code to be executed, as follows:

```
@OnEvent('order.created')
handleOrderCreatedEvent(payload: OrderCreatedEvent) {
  // handle and process "OrderCreatedEvent" event
}
```

> warning **Warning** Event subscribers cannot be request-scoped.

The first argument can be a `string` or `symbol` for a simple event emitter and a `string | symbol | Array<string | symbol>` in a case of a wildcard emitter. The second argument (optional) is a listener options object (read more).

```
@OnEvent('order.created', { async: true })
handleOrderCreatedEvent(payload: OrderCreatedEvent) {
  // handle and process "OrderCreatedEvent" event
}
```

To use namespaces/wildcards, pass the `wildcard` option into the `EventEmitterModule#forRoot()` method. When namespaces/wildcards are enabled, events can either be strings (`foo.bar`) separated by a delimiter or arrays (`['foo', 'bar']`). The delimiter is also configurable as a configuration property (`delimiter`). With namespaces feature enabled, you can subscribe to events using a wildcard:

```
@OnEvent('order.*')
handleOrderEvents(payload: OrderCreatedEvent | OrderRemovedEvent |
OrderUpdatedEvent) {
  // handle and process an event
}
```

Note that such a wildcard only applies to one block. The argument `order.*` will match, for example, the events `order.created` and `order.shipped` but not `order.delayed.out_of_stock`. In order to listen to such events, use the `multilevel wildcard` pattern (i.e, `**`), described in the `EventEmitter2` documentation.

With this pattern, you can, for example, create an event listener that catches all events.

```
@OnEvent('**')
handleEverything(payload: any) {
  // handle and process an event
}
```

> info **Hint** `EventEmitter2` class provides several useful methods for interacting with events, like `waitFor` and `onAny`. You can read more about them here.

Example

A working example is available here.