CQRS

The flow of simple CRUD (Create, Read, Update and Delete) applications can be described as follows:

1. The controllers layer handles HTTP requests and delegates tasks to the services layer.
2. The services layer is where most of the business logic lives.
3. Services use repositories / DAOs to change / persist entities.
4. Entities act as containers for the values, with setters and getters.

While this pattern is usually sufficient for small and medium-sized applications, it may not be the best choice for larger, more complex applications. In such cases, the **CQRS** (Command and Query Responsibility Segregation) model may be more appropriate and scalable (depending on the application's requirements). Benefits of this model include:

- **Separation of concerns**. The model separates the read and write operations into separate models.
- **Scalability**. The read and write operations can be scaled independently.
- **Flexibility**. The model allows for the use of different data stores for read and write operations.
- **Performance**. The model allows for the use of different data stores optimized for read and write operations.

To facilitate that model, Nest provides a lightweight CQRS module. This chapter describes how to use it.

Installation

First install the required package:

```
$ npm install --save @nestjs/cqrs
```

Commands

Commands are used to change the application state. They should be task-based, rather than data centric. When a command is dispatched, it is handled by a corresponding **Command Handler**. The handler is responsible for updating the application state.

```
@@filename(heroes-game.service)
@Injectable()
export class HeroesGameService {
  constructor(private commandBus: CommandBus) {}

  async killDragon(heroId: string, killDragonDto: KillDragonDto) {
    return this.commandBus.execute(
      new KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}
@@switch
@Injectable()
```

```
@Dependencies(CommandBus)
export class HeroesGameService {
  constructor(commandBus) {
    this.commandBus = commandBus;
  }

  async killDragon(heroId, killDragonDto) {
    return this.commandBus.execute(
      new KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}
```

In the code snippet above, we instantiate the `KillDragonCommand` class and pass it to the `CommandBus`'s `execute()` method. This is the demonstrated command class:

```
@@filename(kill-dragon.command)
export class KillDragonCommand {
  constructor(
    public readonly heroId: string,
    public readonly dragonId: string,
  ) {}
}
@@switch
export class KillDragonCommand {
  constructor(heroId, dragonId) {
    this.heroId = heroId;
    this.dragonId = dragonId;
  }
}
```

The `CommandBus` represents a **stream** of commands. It is responsible for dispatching commands to the appropriate handlers. The `execute()` method returns a promise, which resolves to the value returned by the handler.

Let's create a handler for the `KillDragonCommand` command.

```
@@filename(kill-dragon.handler)
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements
ICommandHandler<KillDragonCommand> {
  constructor(private repository: HeroRepository) {}

  async execute(command: KillDragonCommand) {
    const { heroId, dragonId } = command;
    const hero = this.repository.findOneById(+heroId);

    hero.killEnemy(dragonId);
    await this.repository.persist(hero);
```

```
    }
  }
  @@switch
  @CommandHandler(KillDragonCommand)
  @Dependencies(HeroRepository)
  export class KillDragonHandler {
    constructor(repository) {
      this.repository = repository;
    }

    async execute(command) {
      const { heroId, dragonId } = command;
      const hero = this.repository.findOneById(+heroId);

      hero.killEnemy(dragonId);
      await this.repository.persist(hero);
    }
  }
```

This handler retrieves the `Hero` entity from the repository, calls the `killEnemy()` method, and then persists the changes. The `KillDragonHandler` class implements the `ICommandHandler` interface, which requires the implementation of the `execute()` method. The `execute()` method receives the command object as an argument.

Queries

Queries are used to retrieve data from the application state. They should be data centric, rather than task-based. When a query is dispatched, it is handled by a corresponding **Query Handler**. The handler is responsible for retrieving the data.

The `QueryBus` follows the same pattern as the `CommandBus`. Query handlers should implement the `IQueryHandler` interface and be annotated with the `@QueryHandler()` decorator.

Events

Events are used to notify other parts of the application about changes in the application state. They are dispatched by **models** or directly using the `EventBus`. When an event is dispatched, it is handled by corresponding **Event Handlers**. Handlers can then, for example, update the read model.

For demonstration purposes, let's create an event class:

```
  @@filename(hero-killed-dragon.event)
  export class HeroKilledDragonEvent {
    constructor(
      public readonly heroId: string,
      public readonly dragonId: string,
    ) {}
  }
  @@switch
  export class HeroKilledDragonEvent {
```

```
  constructor(heroId, dragonId) {
    this.heroId = heroId;
    this.dragonId = dragonId;
  }
}
```

Now while events can be dispatched directly using the `EventBus.publish()` method, we can also dispatch them from the model. Let's update the `Hero` model to dispatch the `HeroKilledDragonEvent` event when the `killEnemy()` method is called.

```
@@filename(hero.model)
export class Hero extends AggregateRoot {
  constructor(private id: string) {
    super();
  }

  killEnemy(enemyId: string) {
    // Business logic
    this.apply(new HeroKilledDragonEvent(this.id, enemyId));
  }
}
@@switch
export class Hero extends AggregateRoot {
  constructor(id) {
    super();
    this.id = id;
  }

  killEnemy(enemyId) {
    // Business logic
    this.apply(new HeroKilledDragonEvent(this.id, enemyId));
  }
}
```

The `apply()` method is used to dispatch events. It accepts an event object as an argument. However, since our model is not aware of the `EventBus`, we need to associate it with the model. We can do that by using the `EventPublisher` class.

```
@@filename(kill-dragon.handler)
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements
ICommandHandler<KillDragonCommand> {
  constructor(
    private repository: HeroRepository,
    private publisher: EventPublisher,
  ) {}

  async execute(command: KillDragonCommand) {
    const { heroId, dragonId } = command;
```

```
    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}
@@switch
@CommandHandler(KillDragonCommand)
@Dependencies(HeroRepository, EventPublisher)
export class KillDragonHandler {
  constructor(repository, publisher) {
    this.repository = repository;
    this.publisher = publisher;
  }

  async execute(command) {
    const { heroId, dragonId } = command;
    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}
```

The `EventPublisher#mergeObjectContext` method merges the event publisher into the provided object, which means that the object will now be able to publish events to the events stream.

Notice that in this example we also call the `commit()` method on the model. This method is used to dispatch any outstanding events. To automatically dispatch events, we can set the `autoCommit` property to `true`:

```
export class Hero extends AggregateRoot {
  constructor(private id: string) {
    super();
    this.autoCommit = true;
  }
}
```

In case we want to merge the event publisher into a non-existing object, but rather into a class, we can use the `EventPublisher#mergeClassContext` method:

```
const HeroModel = this.publisher.mergeClassContext(Hero);
const hero = new HeroModel('id'); // <-- HeroModel is a class
```

Now every instance of the `HeroModel` class will be able to publish events without using `mergeObjectContext()` method.

Additionally, we can emit events manually using `EventBus`:

```
this.eventBus.publish(new HeroKilledDragonEvent());
```

> info **Hint** The `EventBus` is an injectable class.

Each event can have multiple **Event Handlers**.

```
@@filename(hero-killed-dragon.handler)
@EventsHandler(HeroKilledDragonEvent)
export class HeroKilledDragonHandler implements
IEventHandler<HeroKilledDragonEvent> {
  constructor(private repository: HeroRepository) {}

  handle(event: HeroKilledDragonEvent) {
    // Business logic
  }
}
```

> info **Hint** Be aware that when you start using event handlers you get out of the traditional HTTP web context.
>
> - Errors in `CommandHandlers` can still be caught by built-in Exception filters.
> - Errors in `EventHandlers` can't be caught by Exception filters: you will have to handle them manually. Either by a simple `try/catch`, using Sagas by triggering a compensating event, or whatever other solution you choose.
> - HTTP Responses in `CommandHandlers` can still be sent back to the client.
> - HTTP Responses in `EventHandlers` cannot. If you want to send information to the client you could use WebSocket, SSE, or whatever other solution you choose.

Sagas

Saga is a long-running process that listens to events and may trigger new commands. It is usually used to manage complex workflows in the application. For example, when a user signs up, a saga may listen to the `UserRegisteredEvent` and send a welcome email to the user.

Sagas are an extremely powerful feature. A single saga may listen for 1..* events. Using the RxJS library, we can filter, map, fork, and merge event streams to create sophisticated workflows. Each saga returns an Observable which produces a command instance. This command is then dispatched **asynchronously** by the `CommandBus`.

Let's create a saga that listens to the `HeroKilledDragonEvent` and dispatches the `DropAncientItemCommand` command.

```
@@filename(heroes-game.saga)
@Injectable()
```

```
export class HeroesGameSagas {
  @Saga()
  dragonKilled = (events$: Observable<any>): Observable<ICommand> => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}
@@switch
@Injectable()
export class HeroesGameSagas {
  @Saga()
  dragonKilled = (events$) => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}
```

> info **Hint** The `ofType` operator and the `@Saga()` decorator are exported from the `@nestjs/cqrs` package.

The `@Saga()` decorator marks the method as a saga. The `events$` argument is an Observable stream of all events. The `ofType` operator filters the stream by the specified event type. The `map` operator maps the event to a new command instance.

In this example, we map the `HeroKilledDragonEvent` to the `DropAncientItemCommand` command. The `DropAncientItemCommand` command is then auto-dispatched by the `CommandBus`.

Setup

To wrap up, we need to register all command handlers, event handlers, and sagas in the `HeroesGameModule`:

```
@@filename(heroes-game.module)
export const CommandHandlers = [KillDragonHandler,
DropAncientItemHandler];
export const EventHandlers =  [HeroKilledDragonHandler,
HeroFoundItemHandler];

@Module({
  imports: [CqrsModule],
  controllers: [HeroesGameController],
  providers: [
    HeroesGameService,
    HeroesGameSagas,
    ...CommandHandlers,
```

```
    ...EventHandlers,
    HeroRepository,
  ]
})
export class HeroesGameModule {}
```

Unhandled exceptions

Event handlers are executed in the asynchronous manner. This means they should always handle all exceptions to prevent application from entering the inconsistent state. However, if an exception is not handled, the `EventBus` will create the `UnhandledExceptionInfo` object and push it to the `UnhandledExceptionBus` stream. This stream is an `Observable` which can be used to process unhandled exceptions.

```
private destroy$ = new Subject<void>();

constructor(private unhandledExceptionsBus: UnhandledExceptionBus) {
  this.unhandledExceptionsBus
    .pipe(takeUntil(this.destroy$))
    .subscribe((exceptionInfo) => {
      // Handle exception here
      // e.g. send it to external service, terminate process, or publish a
new event
    });
}

onModuleDestroy() {
  this.destroy$.next();
  this.destroy$.complete();
}
```

To filter out exceptions, we can use the `ofType` operator, as follows:

```
this.unhandledExceptionsBus.pipe(takeUntil(this.destroy$),
UnhandledExceptionBus.ofType(TransactionNotAllowedException)).subscribe((e
xceptionInfo) => {
  // Handle exception here
});
```

Where `TransactionNotAllowedException` is the exception we want to filter out.

The `UnhandledExceptionInfo` object contains the following properties:

```
export interface UnhandledExceptionInfo<Cause = IEvent | ICommand,
Exception = any> {
  /**
```

```
   * The exception that was thrown.
   */
  exception: Exception;
  /**
   * The cause of the exception (event or command reference).
   */
  cause: Cause;
}
```

Subscribing to all events

`CommandBus`, `QueryBus` and `EventBus` are all **Observables**. This means that we can subscribe to the entire stream and, for example, process all events. For example, we can log all events to the console, or save them to the event store.

```
private destroy$ = new Subject<void>();

constructor(private eventBus: EventBus) {
  this.eventBus
    .pipe(takeUntil(this.destroy$))
    .subscribe((event) => {
      // Save events to database
    });
}

onModuleDestroy() {
  this.destroy$.next();
  this.destroy$.complete();
}
```

Example

A working example is available here.