CQRS

The flow of the simplest **CRUD** applications can be described using the following steps:

1. Controllers layer handle **HTTP requests** and delegate tasks to the services.
2. Services layer is a place where the most of the business logic is being done.
3. **Services** uses Repositories / DAOs to change / persist entities.
4. Entities act as containers for the values, with setters and getters.

In most cases, there are no reasons to make small and medium-sized applications more complex. However, sometimes it's not enough and when our needs become **more sophisticated** we want to have scalable systems with straightforward data flow.

Hence, we provide a lightweight **CQRS module** which elements are described below.

Commands

In order to make the application easier to understand, each change has to be preceded by **Command**. When any command is dispatched, the application has to react on it. Commands can be dispatched from the services (or directly from the controllers/gateways) and consumed in corresponding **Command Handlers**.

heroes-game.service.ts JS

```
@Injectable()
export class HeroesGameService {
  constructor(private readonly commandBus: CommandBus) {}

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

Here's a sample service that dispatches `KillDragonCommand`. Let's see how the command looks like:

kill-dragon.command.ts JS

```
export class KillDragonCommand {
  constructor(
    public readonly heroId: string,
```

```
    public readonly dragonId: string,
  ) {}
}
```

The `CommandBus` is a commands **stream**. It delegates commands to the equivalent handlers. Each command has to have corresponding **Command Handler**:

kill-dragon.handler.ts JS

```
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements ICommandHandler<KillDragonCommand> {
  constructor(private readonly repository: HeroRepository) {}

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

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

Now every application state change is a result of the **Command** occurrence. The logic is encapsulated in handlers. If we want, we can simply add logging here or even more, we can persist our commands in the database (e.g. for the diagnostics purposes).

Events

Since we have encapsulated commands in handlers, we prevent interaction between them - the application structure is still not flexible, not **reactive**. The solution is to use **events**.

hero-killed-dragon.event.ts JS

```
export class HeroKilledDragonEvent {
  constructor(
    public readonly heroId: string,
    public readonly dragonId: string,
  ) {}
}
```

Events are asynchronous. They are dispatched either by **models** or directly using `EventBus`. In order to dispatch events,

models have to extend the `AggregateRoot` class.

hero.model.ts JS

```
export class Hero extends AggregateRoot {
  constructor(private readonly id: string) {
    super();
  }

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

The `apply()` method does not dispatch events yet because there's no relationship between model and the `EventPublisher` class. How to associate the model and the publisher? We need to use a publisher `mergeObjectContext()` method inside our command handler.

kill-dragon.handler.ts JS

```
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements ICommandHandler<KillDragonCommand> {
  constructor(
    private readonly repository: HeroRepository,
    private readonly 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();
  }
}
```

Now everything works as expected. Notice that we need to `commit()` events since they're not being dispatched immediately. Obviosuly, an object doesn't have to exist upfront. We can easily merge type context as well:

```
const HeroModel = this.publisher.mergeContext(Hero);
new HeroModel('id');
```

That's it. A model has an ability to publish events now. And we have to handle them. Additionally, we could emit events manually using `EventBus`:

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

> **HINT**
> The `EventBus` is an injectable class.

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

hero-killed-dragon.handler.ts JS

```
@EventsHandler(HeroKilledDragonEvent)
export class HeroKilledDragonHandler implements IEventHandler<HeroKilledDragonEvent> {
  constructor(private readonly repository: HeroRepository) {}

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

Now we can move the **write logic** into the event handlers.

Sagas

This type of **Event-Driven Architecture** improves application **reactiveness and scalability**. Now, when we have events, we can simply react to them in various ways. The **Sagas** are the last building block from the architecture point of view.

The sagas are an incredibly powerful feature. Single saga may listen for 1..* events. It can combine, merge, filter [...] events streams. **RxJS** library is the place where the magic comes from. In simple words, each saga has to return an Observable which contains a command. This command is dispatched **asynchronously**.

heroes-game.saga.ts JS

```
@Injectable()
```

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

> **HINT**
> The `ofType` operator is exported from the `@nestjs/cqrs` package.

We declared a rule - when any hero kills the dragon, the ancient item is being dropped. Afterwards, the `DropAncientItemCommand` will be dispatched and processed by the appropriate handler.

Queries

The `CqrsModule` might be also handy for queries processing. The `QueryBus` works the same as `CommandsBus`. Also, query handlers should implement the `IQueryHandler` interface and be marked with the `@QueryHandler()` decorator.

Setup

The last thing which we have to take care of is to set up the whole mechanism.

heroes-game.module.ts JS

```
export const CommandHandlers = [KillDragonHandler, DropAncientItemHandler];
export const EventHandlers =  [HeroKilledDragonHandler, HeroFoundItemHandler];

@Module({
  imports: [CqrsModule],
  controllers: [HeroesGameController],
  providers: [
    HeroesGameService,
    HeroesGameSagas,
    ...CommandHandlers,
    ...EventHandlers,
    HeroRepository,
  ]
})
export class HeroesGameModule {}
```

Summary

`CommandBus`, `QueryBus` and `EventBus` are **Observables**. It means that you can easily subscribe to the whole stream and enrich your application with **Event Sourcing**.

A working example 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