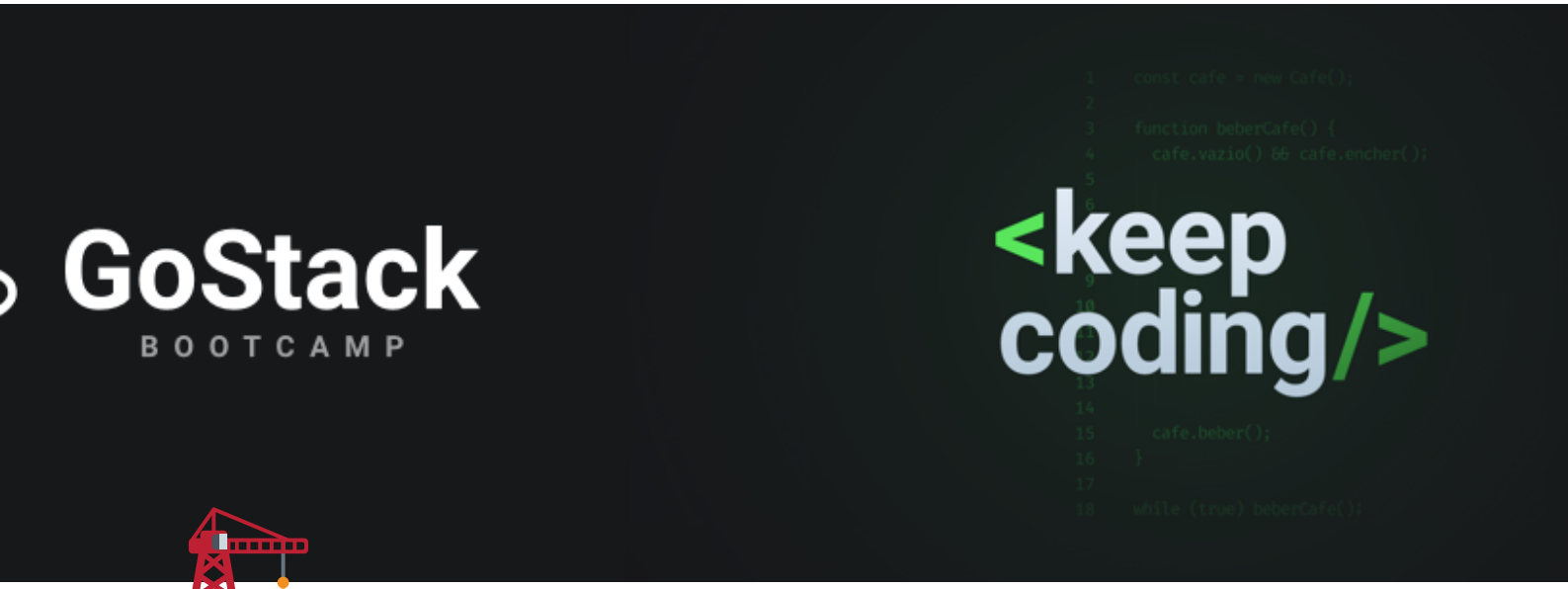

Repository, service e patterns

Nesse documento vamos desmistificar a utilização desses dois patterns dentro de aplicações back-end e comparar suas utilizações com outros padrões comuns como os controllers do MVC.

Sumário

Repository

O Repository é um conceito introduzido no Data Mapper Pattern ou Repository Pattern que consiste em uma ponte entre nossa aplicação e a fonte de dados, seja ela um banco de dados, um arquivo físico ou qualquer outro meio de persistência de dados da aplicação.

Essa implementação visa isolar a forma com que nos comunicamos com os dados, abstraindo lógicas comuns de operações no banco.

Geralmente o Repository possui os métodos comuns de comunicação com uma fonte de dados como listagem, busca, criação, edição, remoção, mas conforme a aplicação cresce o desenvolvedor tende a encontrar outras operações repetitíveis e, com isso, popula o repositório com mais funcionalidades.

🏩 Exemplo real

Imagine uma aplicação que controla reserva de quartos em um hotel, uma pessoa pode acessar o site, reservar um quarto e pagar pelo mesmo. Essa reserva depende do quarto estar vago para esse intervalo de datas que o usuário selecionar.

Se pensarmos nisso como uma consulta no banco, precisaremos realizar uma query um pouco complexa onde comparamos a data de entrada e saída com outras reservas já existentes na aplicação, buscando a disponibilidade do quarto.

Em um cenário real, essa busca por disponibilidade de um quarto pode ser feita em várias partes da aplicação, a home page do site pode possuir uma busca prévia de disponibilidade, a reserva precisa verificar a disponibilidade, o atendente do hotel precisa conseguir verificar disponibilidade para reservas no balcão, ou seja, uma mesma query no banco de dados sendo utilizada em múltiplos contextos, por isso criamos isso em um Repository, reaproveitamento.

Service

O Service é um conceito introduzido no Service Pattern. Ele tem como objetivo abstrair regras de negócio das rotas, além de tornar nosso código mais reutilizável.

No contexto da nossa jornada, essa implementação visa reduzir a complexidade das rotas da nossa aplicação e deixá-las responsáveis apenas pelo que realmente devem fazer: receber uma requisição, repassar os dados da requisição a outro arquivo e devolver uma resposta.

O Service deve ter um nome descritivo (ex.: updateDeliveryManProfileService) e **sempre** possuir apenas **um** método (ex.: execute()). Além disso, caso outra rota ou arquivo precise executar essa mesma ação, basta chamar e executar esse Service, obedecedo assim a outro importante princípio: DRY (Don't Repeat Yourself).

🏩 Exemplo real

Imagine a mesma aplicação que controla a reserva de quartos em um hotel. Essa reserva pode pode envolver diversas etapas, como verificação da disponibilidade, realização do pagamento, envio de emails, entre outros.

Dessa forma, a simples ação de reservar um quarto irá desencadear em diversas outras ações. Se pensarmos nisso como código, teremos regras de negócio, que não são de responsabilidade do Repository, diretamente na nossa rota. Isso fere alguns princípios de programação como o Single Responsability Principle e, portanto, os Services são criados para serem os responsáveis por realizar essas ações.

Além disso, imagine que em outras ações como consumir produtos do Hotel seja necessário executar algumas ações novamente, como realizar o pagamento. Com o Service criado, basta chamá-lo e executá-lo novamente.

Por que tudo isso?

Os Services e Repositories são utilizados para construir uma base sólida, visando a escabilidade e aplicação de boas práticas. Com o seu uso, apesar de uma maior complexidade no ínicio, é possível obter diversos benefícios ao atender princípios importantes da programação. Alguns deles são:

- **SoC (Separation of Concerns)**: Esse princípio zela pela separação de responsabilidades de cada arquivo. Exemplo: as rotas não devem ser responsáveis por lidar com a persistência dos dados, isso fica por conta do Repository. Já o Repository não é responsável pela tratativa das regras de negócio, isso é responsabilidade dos Services;
- **DRY (Don't Repeat Yourself)**: Esse princípio zela pelo maior reaproveitamento de código. Esse princípio não preza necessariamente pela não-repetição de código e sim regras de negócio. Exemplo: ao criar Services e Repositories, você possibilita a reutilização desses códigos no restante da aplicação;
- **SRP (Single Responsability Principle)**: Esse princípio zela que uma classe deve possuir apenas uma responsabilidade. Exemplo: Ao criar um service chamado `createTransactionService`, devemos garantir que no seu único método (execute()) seu trabalho seja **apenas** a criação de uma transação;
- **DIP (Dependency Inversion Principle)**: Esse princípio zela que uma entidade dependa apenas de abstrações, não de implementações. Exemplo: Ao atribuir ao Repository a comunicação com o Banco de dados, o Service precisa interagir apenas com essa abstração, sem precisar criar uma nova instância e realizar as ações diretamente;

Onde entra o SOLID?

O SOLID é uma sigla que representa 5 princípios da programação orientada a objetos:

- **SRP (Single Responsability Principle);**
- **OCP (Open–closed Principle);**
- **LSP (Liskov substitution Principle);**
- **ISP (Interface segregation Principle);**
- **DIP (Dependency Inversion Principle).**

E ele entra justamente nessa abordagem que estamos utilizando (Repositories e Services). Os princípios **DIP** e **SRP** provavelmente são os mais fáceis de enxergar nesse primeiro momento, mas ao longo do curso o **SOLID** será aplicado cada vez mais, conforme entendemos mais o funcionamento do Typescript e nos acostumamos com o Data Mapper Pattern.

> 💡 Quer ler mais sobre SOLID, confira esse outro documento com exemplos de código.

E os controllers? E o MVC?

Se você já trabalhou com MVC ou participou da antiga jornada, deve estar se perguntando: cadê os controllers? Fique tranquilo, eles estão bem 😁

Até então, utilizávamos o MVC Pattern (Model, View e Controller) onde os nossos controllers basicamente eram responsáveis por todas as regras de negócio (conexão com banco de dados, tratativas de erros, formatação dos dados).

Atualmente, estamos utilizamos o Data Mapper Pattern e focando bastante nos princípios do SOLID e DDD, onde separamos melhor as responsabilidades da aplicação entre os arquivos de rotas, services e repositories.

Mas fique tranquilo, isso não significa que o MVC Pattern morreu, e se você sente tanta falta dos controllers, basta abstrair o código presente nas nossas rotas para arquivos controllers. Ah, inclusive faremos isso mais a frente.