Texto retirado de:

https://imasters.com.br/front-end/como-testar-services-em-nestjs-com-jest

Testar suas aplicações antes de enviá-las para produção é importante e todo mundo já sabe. Mas será que você sabe como testar eficientemente sua aplicação escrita em NestJS?

Neste tutorial veremos como utilizar o módulo Jest para criar testes de unidade (unit tests) dos services em backends NestJS. É importante antes de avançar que você já tenha experiência, ao menos básico, implementando webapis com NestJS, o que pode ser aprendido nesta outra série aqui do blog. Também é desejável, embora não obrigatório, que você já conheça o básico de Jest para Node.js, o que pode ser aprendido neste outro tutorial aqui.

Veremos neste post:

- Criando Unit tests de Services
- Testes Mockados com Jest
- Cobertura de Testes

Vamos lá!

NestJS – Criando Unit Tests de Services

Para este tutorial vamos usar o Jest como biblioteca de testes, que é muito popular atualmente e também porque ele já vem integrado “nativamente” no framework NestJS, permitindo fazer asserções de uma maneira muito simples, prática e padronizada, dando agilidade ao processo de unit testing ou TDD, caso leve a metodologia realmente a sério. Além disso ele cria informações de cobertura de testes muito legais.

Para não começar nossos estudos completamente do zero usarei o projeto visto neste outro tutorial, uma webapi de usuários (CRUD) com banco SQL cujos fontes você consegue baixar ou clonar do meu GitHub público neste link.

Recomendo que você clone esta API na sua máquina e dê uma analisada nos fontes, que são bem simples. Verá que inclusive já temos uma pasta test com um arquivo de teste e outro de configuração dentro. Isso é padrão do NestJS, não fui eu quem criou, então pode excluir o conteúdo dessa pasta. Então crie um novo arquivo de testes chamado user.service.spec.ts. O sufixo ‘spec’ é o padrão para arquivos de teste em NestJS, então vamos manter.

Abaixo o conteúdo inicial desse arquivo, onde começaremos com o script de preparação dos testes e um teste básico se o serviço foi corretamente carregado.

OBJ

Na função beforeAll, que será executada antes de todos testes da nossa suíte (literalmente “before all”), nós vamos inicializar nosso módulo de teste (TestingModule), configurando ele com o nosso UserService como provider, assim como faríamos em um módulo real de um backend NestJS. Repare como uso a função estática Test.createTestingModule para isso, que vai simular a inicialização real de um módulo pra gente. Na sequência, carregamos este serviço em uma variável local com a função get do objeto moduleFixture que criamos, a fim de usá-lo nos demais testes sem precisar repetir esse setup.

Como primeiro teste, vamos testar se o user service foi carregado com sucesso, verificando apenas se está “defined” (ou seja, diferente de undefined, padrão do let). Fazemos isso com a função it, que é apenas um atalho para a função test do Jest, para soar melhor ao ler os testes (na leitura fica “it should be…”). Dentro do callback do it nós escrevemos nosso teste e ao final do mesmo usamos a função expect para analisar um resultado/variável com o auxílio de uma função de aferição, neste caso a toBeDefined, que atende ao que precisamos.

Se você já usou Jest antes, tenho certeza que nada disso é exatamente novidade, apenas as questões específicas do Nest mesmo.

Para rodar nossos testes, primeiro precisamos ir no package.json e procurar a seção jest dele, onde estão as configurações globais do Jest para esta aplicação. Ajuste para que fique como abaixo, onde incluí também uma série de configurações relacionadas a cálculos de cobertura de código (code coverage).

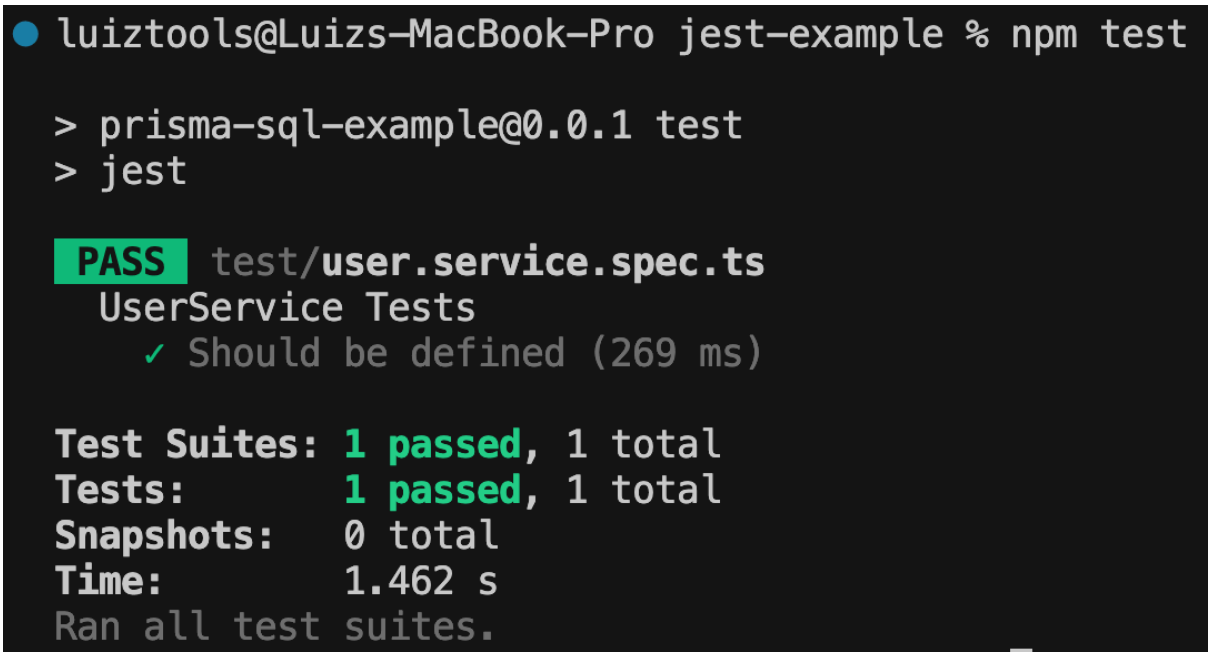
```
"jest": {
  "moduleFileExtensions": [
    "js",
    "json",
    "ts"
  ],
  "roots": [
    "src",
    "test"
  ],
  "testRegex": ".*\\.spec\\.ts$",
  "transform": {
    "^.+\\.(t|j)s$": "ts-jest"
  },
  "collectCoverageFrom": [ "src/**/*.{js,jsx,ts,tsx}" ],
  "coverageDirectory": "../coverage",
  "coverageReporters": ["lcov", "text"],
  "testEnvironment": "node"
}
```

Agora para rodar, basta executar o comando abaixo no terminal:

Agora para rodar, basta executar o comando abaixo no terminal:

```
npm test
```

E com isso você terá o seguinte resultado, se fez tudo certo.



A seguir, recomendo que você escreva um ou mais testes por conta própria, para as funções do UserService, o que deve fazer aparecer alguns desafios e problemas que trato no próximo tópico.

Testes Mockados com Jest

Um unit test é um teste automatizado que testa uma única unidade da sua aplicação, geralmente uma única função, em um único contexto. Cheio de “únicos” nesse parágrafo, não? Assim, para que nossos testes de services sejam realmente unitários, devemos isolá-lo de quaisquer dependências externas.

Nosso UserService não é um serviço exatamente complexo, já que ele é apenas um CRUD de usuário. No entanto ele tem um ponto fundamentalmente chato no que tange testes já que depende de uma infraestrutura de banco de dados para funcionar. Se você quer testar uma consulta, tem de ter massa de dados, certo? Se você quer inserir dados, tem de limpar eles ao final do teste ou até mesmo garantir que eles rodem em sequência pois um insere, outro lê e por aí vai. São vários problemas se quiser testar módulos que usam banco e essas abordagens que citei são todas dor de cabeça na certa, já vi elas na prática. No entanto, se usar essa

abordagem de preparar recursos de infraestrutura para os testes acabará testando a infraestrutura em si e atrasando muito os testes do SEU código de fato.

Como assim Luiz, eu não deveria testar minha infraestrutura nos testes?

Dependendo do teste, até pode, mas pensa comigo: o banco de dados é uma parte da sua aplicação cujas funcionalidades não são de sua gestão ou controle, mas sim do fabricante do banco. Por exemplo, se tiver um bug no salvamento em disco de um registro, você tem como corrigir? Até poderia, já que muitos bancos são open-source, mas esse é um exemplo de aspecto da sua aplicação que você não gerencia, mas depende para o seu código funcionar, toda vez que quer escrever ou ler dados.

Esses pacotes e recursos não-gerenciáveis por você, cujo código você usa mas não mexe, são fortes candidatos a serem mockados, porque eles costumam muitas vezes tornar os seus testes dependentes, acoplados, lentos e falhos, além de exigir muito setup e cleanup, antes e depois dos testes respectivamente.

Mockados? Como assim?

Mocking é uma técnica de testes onde você cria simulações (mocks) de objetos/funções/whatever a serem utilizadas em determinados testes. Assim, ao invés de usar de fato um banco real para os testes, você pode mockar algumas funções do Prisma para que ele ACHE que foi no banco de dados, mas na verdade não foi. Assim, você consegue testar somente o SEU código, buscando bugs nele, ao invés de ficar testando o banco por tabela.

Vamos pegar como exemplo um teste de obter usuário, que normalmente exigiria um banco com um registro ao menos para funcionar:

```
1 it('Should get an user', async () => {
2   const id: string = "1";
3
4   userService.users.findUnique = jest.fn().mockReturnValueOnce({ id, age: 35, name: "LuizTools",
5
6   const result = await userService.getUser(id);
7
8   expect(result.id).toEqual(id);
9 });
```

Repare que aqui eu começo definindo um id falso e depois crio o mock sobrescrevendo a função findUnique da coleção users do nosso userService. Para isso usei a função mockReturnValueOnce, que vai substituir o retorno da função mockada pelo objeto que quisermos, mas somente UMA vez (“return value once”), ou seja, nas próximas chamadas ela já terá voltado ao normal, é um efeito temporário.

Agora sabendo qual o retorno esperado da função, podemos chamar a getUser (que internamente chama a findUnique) e fazemos o expect normalmente. O resultado é um belíssimo teste passando.

Mas Luiz, isso não é roubar?

Não, pois todo o código da userService.getUser está sendo testado, apenas o código interno da findUnique, que é da biblioteca Prisma, que não está. Inclusive isso permite que você baixe os fontes do repositório deste tutorial e rode os testes na sua máquina sem sequer ter um banco de dados criado!!!!

Vamos a mais um exemplo, desta vez de consulta de todos usuários:

```
it('Should get users', async () => {

  const id: string = "1";
  const users = [{ id, age: 35, name: "LuizTools", uf: "RS" }];
  userService.users.findMany = jest.fn().mockReturnValueOnce(users);

  const result = await userService.getUsers();

  expect(result.length).toEqual(users.length);
  expect(result[0].id).toEqual(id);
});
```

Note que fiz a mesma coisa, mas desta vez criei um array fake e mockei a função findMany, pois é ela que é usada internamente na getUsers. Já os testes, resolvi fazer dois expects ao invés de um, o que é uma possibilidade também caso queira dar mais precisão à sua asserção.

Seguindo em frente, que tal um teste de adição de usuário?

```
it('Should add an user', async () => {
  const user = {
    age: 35,
    name: "LuizTools",
    uf: "RS"
  } as users;

  const id: string = "1";

  userService.users.create = jest.fn().mockReturnValueOnce({ id, ...user });

  const result = await userService.addUser(user);

  expect(result.id).toEqual(id);
});
```

Repare como esses mocks de user e id estão ficando repetitivos, você pode inclusive criar eles a nível global no arquivo, reutilizando em cada teste. Apenas

cuide para que um teste não mexa nas propriedades do objeto, o que causaria efeitos colaterais nos outros testes. Eles têm de ser completamente independentes!

Agora um teste de update:

```
it('Should update an user', async () => {
  const user = {
    name: "LuizTools"
  } as users;

  const id: string = "1";

  userService.users.update = jest.fn().mockReturnValueOnce({ id, ...user });

  const result = await userService.updateUser(id, user);

  expect(result.id).toEqual(id);
});
```

Bem parecido com os anteriores, certo? Repare que o “segredo” aqui é saber qual função interna do Prisma você deve mockar para aquele teste, depois o resto é só escrever testes Jest comuns.

E por último, a função de exclusão:

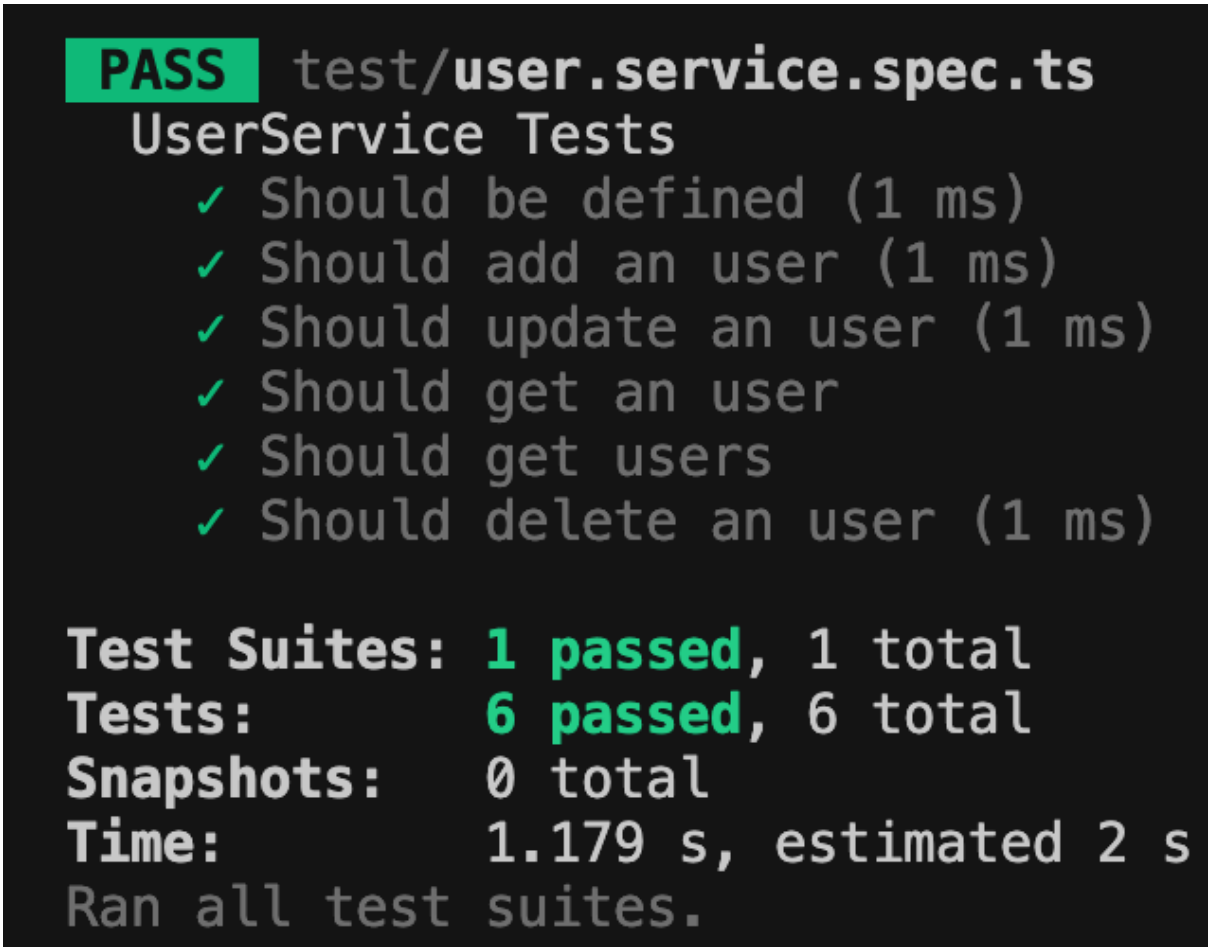
```
it('Should delete an user', async () => {
  const id: string = "1";

  userService.users.delete = jest.fn().mockReturnValueOnce(true);

  const result = await userService.deleteUser(id);

  expect(result).toBeTruthy();
});
```

E com isso temos todos os testes do CRUD finalizados, seu terminal deve ficar assim:



Demais, não?

Cobertura de Testes

Criar todos os testes necessários para garantir que todas as unidade do seu software realmente funcionam é o que chamamos de cobertura de código (code coverage), cuja (quase) utópica marca de 100% deve ser sempre o ideal, embora praticamente inalcançável em sistemas complexos. O quão 'detalhado' os seus testes serão vai muito do grau de importância que todas essas questões possuem para o seu negócio e o quanto domina ele.

E para incrementar o resultado dos testes, vamos adicionar estatísticas de code coverage neles (cobertura de testes de código), rodando o projeto com o seguinte comando.

```
npm test -- --coverage
```

Com isso, não apenas os testes serão executados (todos arquivos spec.ts) como será inspecionado se seus testes cobrem as linhas, funções e ramificações do seu projeto. No meu exemplo, tive o seguinte resultado:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	18.64	100	31.25	19.14	
src	0	100	0	0	
app.controller.ts	0	100	0	0	1-10
app.module.ts	0	100	100	0	1-11
app.service.ts	0	100	0	0	1-6
main.ts	0	100	0	0	1-8
src/user	32.35	100	41.66	32.14	
user.controller.ts	0	100	0	0	1-30
user.module.ts	0	100	100	0	1-10
user.service.ts	91.66	100	83.33	90	8

Note que a maioria dos arquivos do projeto está em vermelho, pois não escrevemos testes para eles, então vamos nos focar no user.service.ts. Como podemos melhorar a cobertura dele, mesmo após termos escrito testes para todas funções do CRUD? Se você olhar as linhas não-cobertas, verá que tem a 8. Indo até lá, verá que é um código de inicialização do Prisma, então você tem duas opções: ou mocka essa função de conexão, ou diz pro Jest ignorar essa função do UserService.

Vou optar pela segunda opção para mostrar mais um truque de testes:

```
1 /* istanbul ignore next */
2 async onModuleInit() {
3   await this.$connect();
4 }
```

Quando você adiciona um comentário iniciado por “istanbul” você pode dizer ao Jest para ele ignorar uma linha, função ou até arquivo inteiro. No caso acima, mandei ele ignorar a próxima função.

O resultado, é um aumento na cobertura já que essa função não estará mais entrando na contabilização, veja abaixo.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	18.96	100	33.33	19.56	
src	0	100	0	0	
app.controller.ts	0	100	0	0	1-10
app.module.ts	0	100	100	0	1-11
app.service.ts	0	100	0	0	1-6
main.ts	0	100	0	0	1-8
src/user	33.33	100	45.45	33.33	
user.controller.ts	0	100	0	0	1-30
user.module.ts	0	100	100	0	1-10
user.service.ts	100	100	100	100	

Experimente escrever os testes para o app.service.ts (muito fácil) e para o main.ts, eu recomendo mandar ignorar o arquivo inteiro, usando a seguinte instrução na primeira linha do mesmo.

```
1 /* istanbul ignore file */
```

Juntando este conhecimento com ferramentas de Continuous Integration (CI) como CircleCI e Jenkins, podemos incluir testes de unidade automatizados em nosso build para evitar que um código suba para produção com algum bug (um dia falarei disso por aqui).

Mas e como podemos fazer testes mais amplos, como testes nos controllers ou End-to-End? Esse tipo de teste é assunto para outro tutorial, em breve.

Espero que tenham gostado do tutorial e que apliquem em seus projetos!