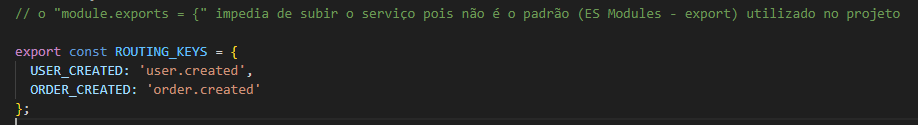
Relatório – Desafio Microsserviços

Arthur Guedes Sant’anna

1. O primeiro erro identificado e alteração realizada foi no arquivo ‘event.js’, dentro da pasta ‘common’



Anteriormente a estrutura do código estava como module.exports, mas foi alterado para o código padrão de módulos do projeto (ESM), assim o serviço sobe normalmente.

1. Após essa etapa superada, o desafio foi conseguir inserir o usuário. Após muitas tentativas frustradas, o problema estava no ‘gateway’, mais especificamente no ‘index.js’

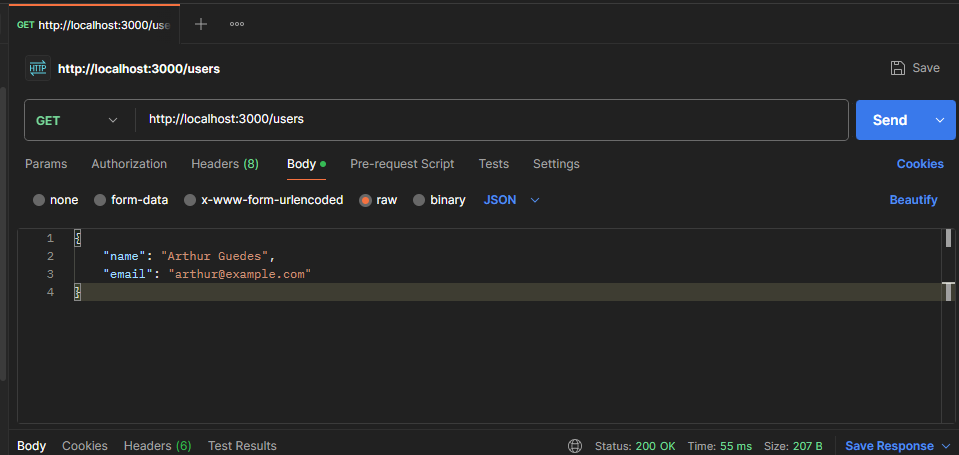


O ‘app.use(express.json())’ na parte de cima retornava vazio no terminal, quando a requisição era enviada ele não percorria o restante do código, e por isso a requisição sempre era mal sucedida.

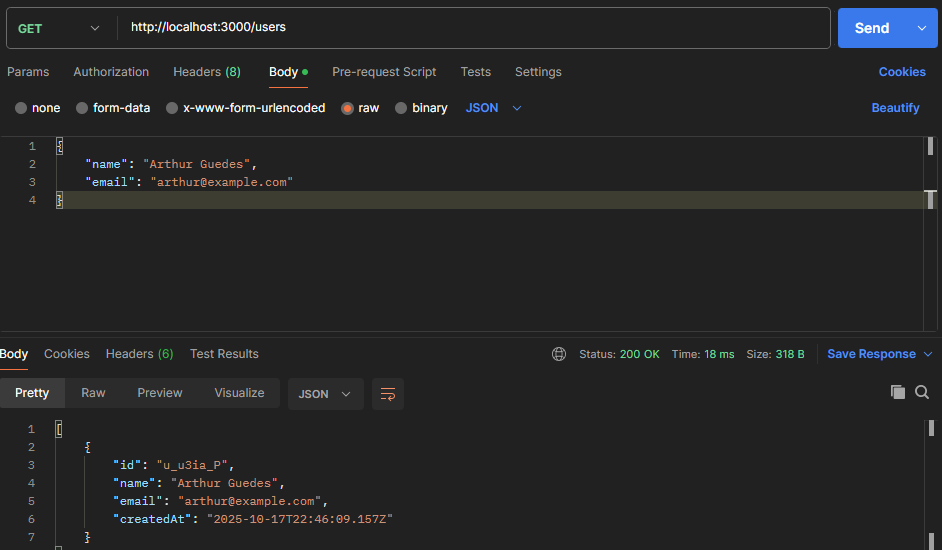


Colocando ele após os ‘use’ o código é percorrido e a requisição é feita com sucesso.

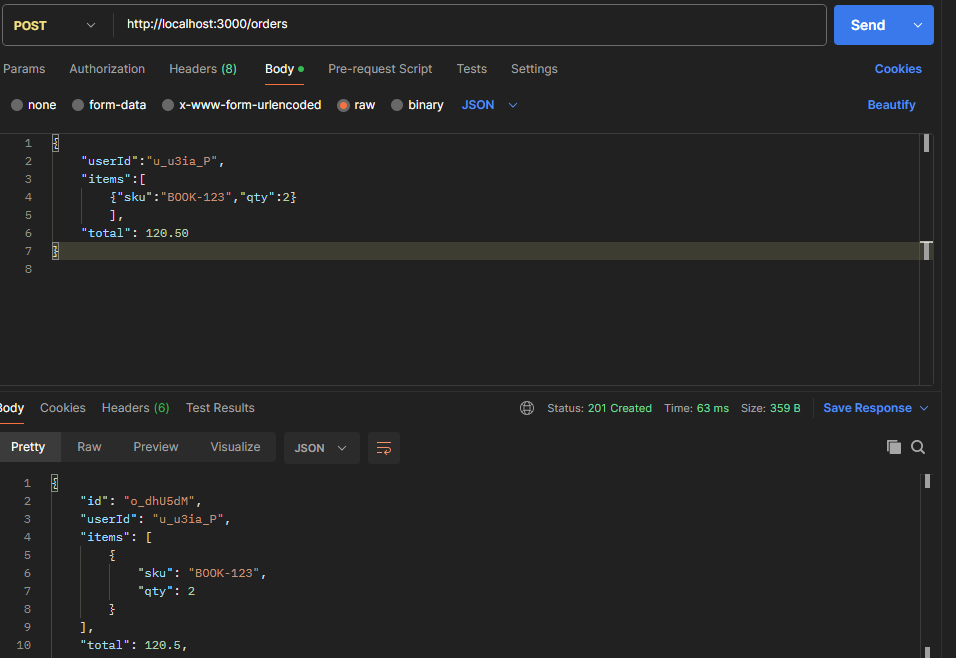
* Criar usuário



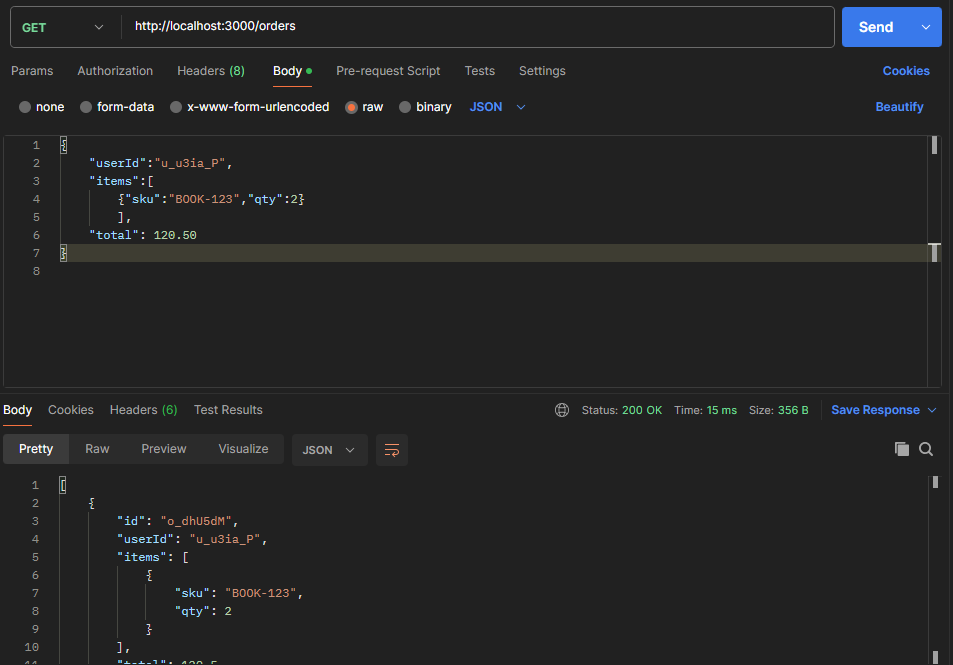
* Listar usuários



* Criar pedido

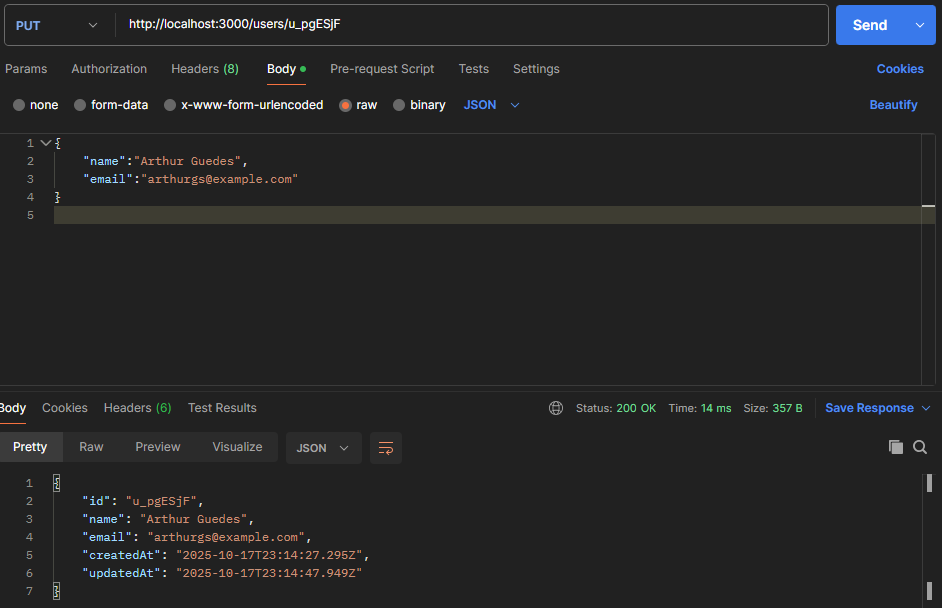


* Listar pedidos

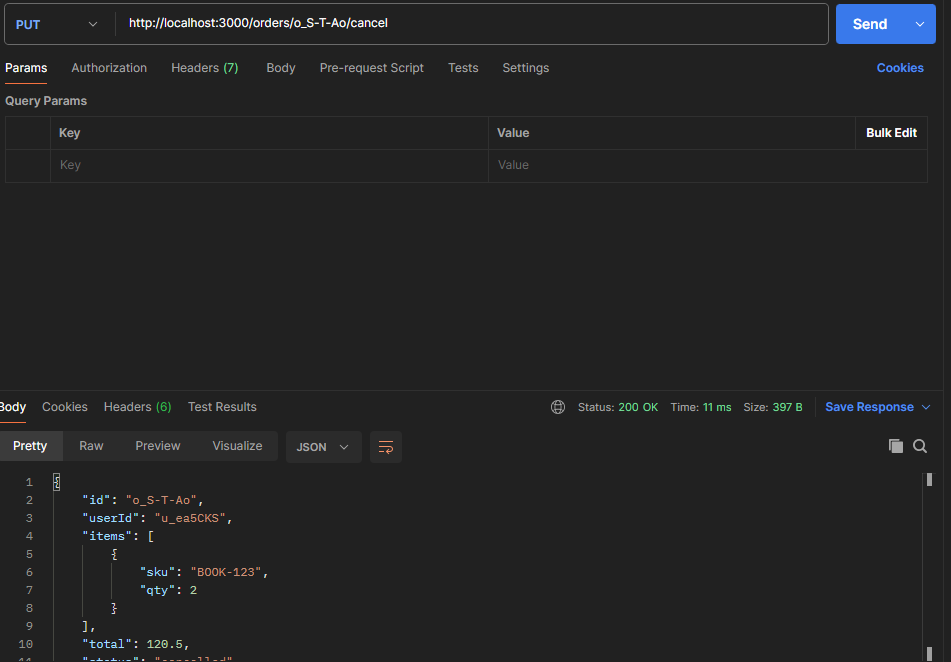


## Exercícios

* ‘user.update’



‘order.cancelled’



1. Instalando o prisma em cada service

npm install prisma --save-dev

npm install @prisma/client

npx prisma init

schema.prisma atualizado em ambos os serviços

index.js atualizado incluindo o Prisma + SQLite mantendo a mesma estrutura do código. Substituido o map em memória pelo banco.

Depois realizada a migração do prisma:

npx prisma migrate dev --name init

npx prisma generate

1. Quando subi novamente os containers o users e orders não estavam mais subindo e apresentavam o erro: Error [ERR\_MODULE\_NOT\_FOUND]: Cannot find module '/common/events.js', ou seja, ele não estava sendo encontrado.

Após pesquisa e dias de cabeça quebrando, através de pesquisa e ajuda dos colegas, entendi que o problema estava na configuração de build das imagens Docker. Isso fazia que o Common não fosse lido corretamente e o ambiente do prisma não fosse executado como devido.

A solução encontrada foi modificar o dockerfile para que o context de build de cada serviço fosse a **raiz do projeto**

Também foi incluso o Commom (módulo compartilhado)

Corrigido a configuração do Prisma no Docker:

O comando RUN npm install foi ajustado para instalar todas as dependências (incluindo devDependencies), necessárias para disponibilizar a CLI do Prisma.

Foram adicionados os comandos COPY .../prisma ./prisma/ e RUN npx prisma generate para garantir que o Prisma Client fosse gerado corretamente durante a construção da imagem no Docker.

1. Criação de testes (Jest/supertest) por serviço.

Instalando Jest, Supertest e cross-env

npm install --save-dev jest supertest cross-env

Jest: O framework que vai rodar os testes, fazer asserções e gerar relatórios.

Supertest: Permite fazer requisições HTTP ao servidor Express de forma programática, sem precisar subir o servidor em uma porta real.

Cross-env: Ajuda a definir variáveis de ambiente de forma compatível com diferentes sistemas operacionais, o que é ótimo para testes.

Após instalação criei o arquivo de configuração do jest

Inclui o script de teste no package.json do serviço

"test": "cross-env NODE\_ENV=test jest --testTimeout=5000"

Define a variável de ambiente NODE\_ENV como test. Isso permite que o código saiba quando está rodando em modo de teste (banco de dados de teste).

jest --testTimeout=5000: Executa o Jest. Aumentamos o timeout para 5 segundos para evitar falhas em testes que envolvam operações de I/O (como banco de dados).

Após incluir o script de teste, criei um novo arquivo nos services app.js. ele vai conter todo

Depois de rodar o npm test no user.service gerou um erro com essa mensagem:

If you are trying to use ECMAScript Modules, see <https://jestjs.io/docs/ecmascript-modules> for how to enable it.

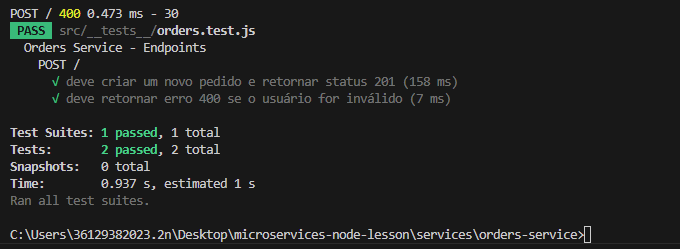
Entendi o erro como uma deficiência de importação do ES Modules para o Jest. A solução encontrada foi instalar o Babel, um ‘tradutor’ do Jest padrão CommonJS traduzindo o código atual para o Jest.

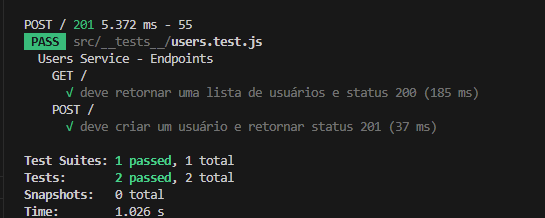
npm install --save-dev babel-jest @babel/core @babel/preset-env

Tive que instalar as dependências do Babel, criar um babel.config.cjs e renomear meu jest.config.js para jest.config.cjs, reescrevendo tudo com require e module.exports.

Após isso começaram os erros de lógica. A ordem fazia a diferença por isso foram aplicados beforeEach, afterEach com a finalidade de manter uma sequencia logica.

Erros de Unexpected token 'export' vindo do node\_modules surgiram muito, foi preciso listar as exceções no jest.config.cjs, na propriedade transformIgnorePatterns, para forçar o Babel a traduzir todos eles.





npm test. Os testes foram bem sucedidos.

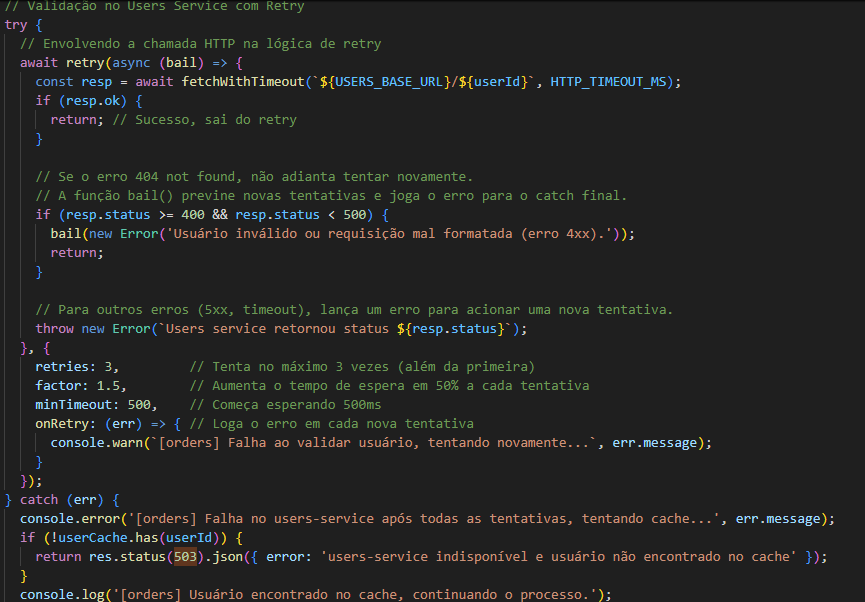
1. Adicionar \*\*retry com backoff\*\* para conexões AMQP/HTTP

Iniciando instalando uma biblioteca chamada async-retry.

A primeira coisa feita foi atacar a conexão com o RabbitMQ, tanto no users-service quanto no orders-service. Peguei todo aquele bloco de código que iniciava a conexão e o envolvi com a função retry. Subi o serviço sem o RabbitMQ no ar e os logs de "tentando de novo" aparecem com um intervalo cada vez maior. Aí, subindo novamente o RabbitMQ, na tentativa seguinte a conexão foi estabelecida.

Depois, comecei a chamada HTTP no orders-service. A lógica era a mesma, envolver a chamada fetch para o users-service em um bloco retry. Mas um detalhe, não adianta tentar de novo se o erro for, por exemplo, um "404 Not Found". Se o usuário não existe, ele não vai surgir.

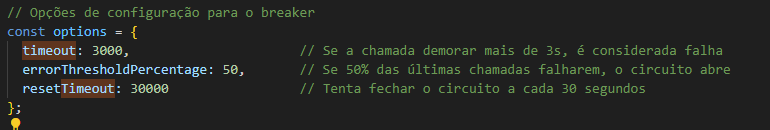
Para validar a resiliência, criei testes automatizados com Jest que simulam falhas de rede e erros específicos. Verifiquei se a lógica de retry era acionada corretamente para erros temporários e se a função bail impedia novas tentativas para erros permanentes, como 404



1. Incluir \*\*circuit breaker\*\* (p.ex. opossum) no Orders

Primeiro instalei a biblioteca opussum

A implementação foi focada na comunicação do orders-service com o users-service. Primeiro, criei uma função que continha a lógica da chamada fetch que eu queria proteger. Depois, instanciei o opossum com as minhas regras:



Defini que, quando o circuito estivesse aberto, em vez de simplesmente dar erro, a aplicação deveria primeiro verificar se o usuário já estava em um cache local. Se não estivesse, aí sim retornaria um erro 503 (Serviço Indisponível).

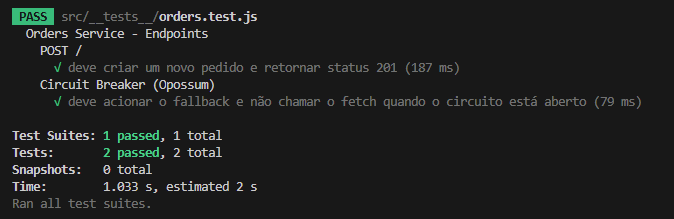
Na rota de criação de pedido (POST /), eu simplesmente substituí todo o bloco retry anterior por uma única chamada: await breaker.fire(userId). O código da rota ficou muito mais limpo e a responsabilidade da resiliência passou a ser totalmente do breaker.

Para testar, a abordagem foi bem específica. No Jest, criei um cenário de teste dedicado a isso:

Configurei o mock do node-fetch para que ele **falhasse todas as vezes**, simulando que o users-service estava completamente fora do ar.

No teste, fiz 5 chamadas seguidas para a rota de criação de pedido.

Depois, fiz uma sexta chamada.



1. Instalação do Swagger

npm install swagger-jsdoc swagger-ui-express