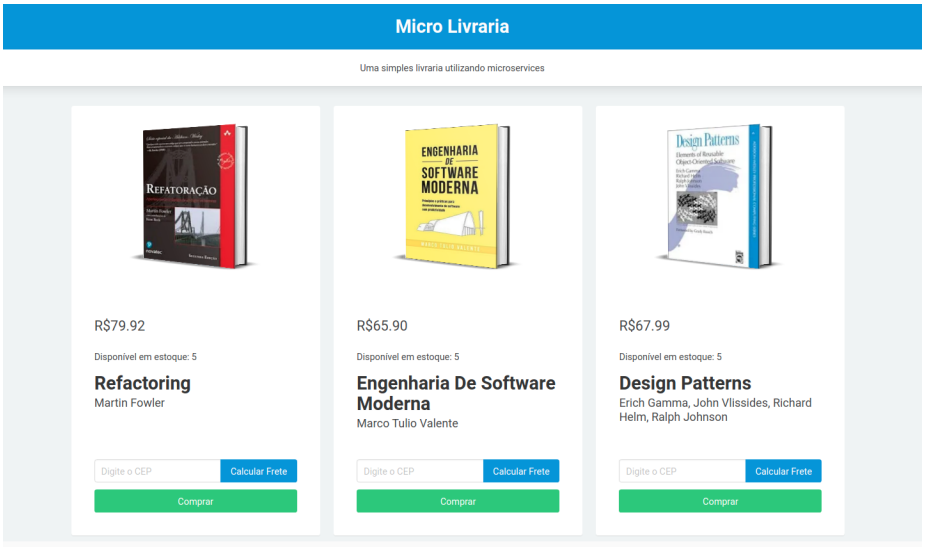
Micro-Livraria: Exemplo Prático de Microsserviços

Este repositório contem um exemplo simples de uma livraria virtual construída usando uma **arquitetura de microsserviços**.

O exemplo foi projetado para ser usado em uma **aula prática sobre microsserviços**, que pode, por exemplo, ser realizada após o estudo do Capítulo 7 do livro Engenharia de Software Moderna.

O objetivo da aula é permitir que o aluno tenha um primeiro contato com microsserviços e com tecnologias normalmente usadas nesse tipo de arquitetura, tais como **Node.js**, **REST**, **gRPC** e **Docker**.

Como nosso objetivo é didático, na livraria virtual estão à venda apenas três livros, conforme pode ser visto na próxima figura, que mostra a interface Web do sistema. Além disso, a operação de compra apenas simula a ação do usuário, não efetuando mudanças no estoque. Assim, os clientes da livraria podem realizar apenas duas operações: (1) listar os produtos à venda; (2) calcular o frete de envio.



No restante deste documento vamos:

- Descrever o sistema, com foco na sua arquitetura.
- Apresentar instruções para sua execução local, usando o código disponibilizado no repositório.
- Descrever duas tarefas práticas para serem realizadas pelos alunos, as quais envolvem:
    - Tarefa Prática #1: Implementação de uma nova operação em um dos microsserviços
    - Tarefa Prática #2: Criação de containers Docker para facilitar a execução dos microsserviços.

Arquitetura

A micro-livraria possui quatro microsserviços:

- Front-end: microsserviço responsável pela interface com usuário, conforme mostrado na figura anterior.
- Controller: microsserviço responsável por intermediar a comunicação entre o front-end e o backend do sistema.
- Shipping: microserviço para cálculo de frete.

- Inventory: microserviço para controle do estoque da livraria.

Os quatro microsserviços estão implementados em **JavaScript**, usando o Node.js para execução dos serviços no back-end.

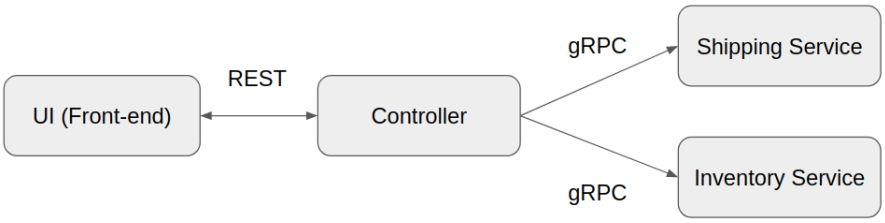
No entanto, **você conseguirá completar as tarefas práticas mesmo se nunca programou em JavaScript**. O motivo é que o nosso roteiro já inclui os trechos de código que devem ser copiados para o sistema.

Para facilitar a execução e entendimento do sistema, também não usamos bancos de dados ou serviços externos.

Protocolos de Comunicação

Como ilustrado no diagrama a seguir, a comunicação entre o front-end e o backend usa uma **API REST**, como é comum no caso de sistemas Web.

Já a comunicação entre o Controller e os microsserviços do back-end é baseada em gRPC.



Optamos por usar gRPC no backend porque ele possui um desempenho melhor do que REST.
Especificamente, gRPC é baseado no conceito de **Chamada Remota de Procedimentos (RPC)**. A ideia é simples: em aplicações distribuídas que usam gRPC, um cliente pode chamar funções implementadas em outros processos de forma transparente, isto é, como se tais funções fossem locais. Em outras palavras, chamadas gRPC tem a mesma sintaxe de chamadas normais de função.

Para viabilizar essa transparência, gRPC usa dois conceitos centrais:

- uma linguagem para definição de interfaces
- um protocolo para troca de mensagens entre aplicações clientes e servidoras.

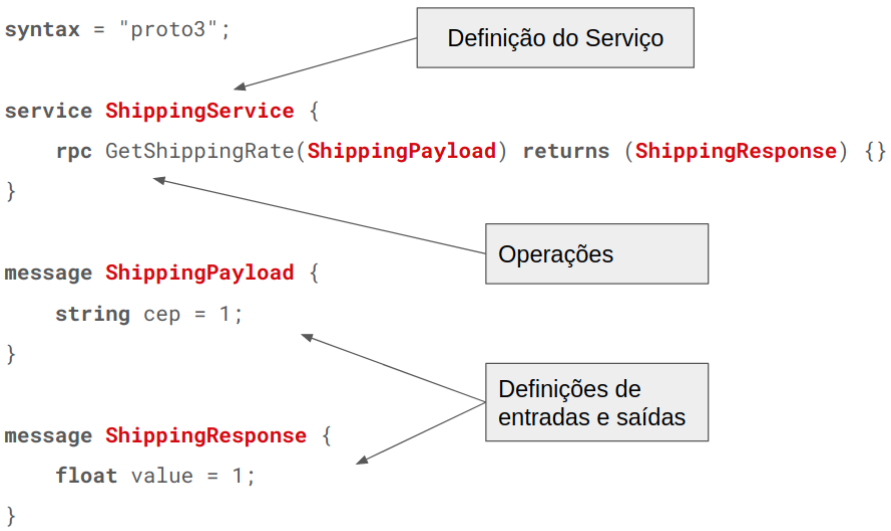
Especificamente, no caso de gRPC, a implementação desses dois conceitos ganhou o nome de **Protocol Buffer**. Ou seja, podemos dizer que:

> Protocol Buffer = linguagem para definição de interfaces + protocolo para definição das mensagens trocadas entre aplicações clientes e servidoras

Exemplo de Arquivo .proto

Quando trabalhamos com gRPC, cada microserviço possui um arquivo `.proto` que define a assinatura das operações que ele disponibiliza para os outros microsserviços.
Neste mesmo arquivo, declaramos também os tipos dos parâmetros de entrada e saída dessas operações.

O exemplo a seguir mostra o arquivo .proto do nosso microsserviço de frete. Nele, definimos que esse microsserviço disponibiliza uma função `GetShippingRate`. Para chamar essa função devemos passar como parâmetro de entrada um objeto contendo o CEP (`ShippingPayLoad`). Após sua execução, a função retorna como resultado um outro objeto (`ShippingResponse`) com o valor do frete.



Em gRPC, as mensagens (exemplo: `Shippingload`) são formadas por um conjunto de campos, tal como em um `struct` da linguagem C, por exemplo. Todo campo possui um nome (exemplo: `cep`) e um tipo (exemplo: `string`). Além disso, todo campo tem um número inteiro que funciona como um identificador único para o mesmo na mensagem (exemplo: `= 1`). Esse número é usado pela implementação de gRPC para identificar o campo no formato binário de dados usado por gRPC para comunicação distribuída.

Arquivos .proto são usados para gerar **stubs**, que nada mais são do que proxies que encapsulam os detalhes de comunicação em rede, incluindo troca de mensagens, protocolos, etc. Mais detalhes sobre o padrão de projeto Proxy podem ser obtidos no Capítulo 6.

Em linguagens estáticas, normalmente precisa-se chamar um compilador para gerar o código de tais stubs. No caso de JavaScript, no entanto, esse passo não é necessário, pois os stubs são gerados de forma transparente, em tempo de execução.

Executando o Sistema

A seguir vamos descrever a sequência de passos para você executar o sistema localmente em sua máquina. Ou seja, todos os microsserviços estarão rodando na sua máquina.

IMPORTANTE: Você deve seguir esses passos antes de implementar as tarefas práticas descritas nas próximas seções.

1. Faça um fork do repositório. Para isso, basta clicar no botão **Fork** no canto superior direito desta página.

2. Vá para o terminal do seu sistema operacional e clone o projeto (lembre-se de incluir o seu usuário GitHub na URL antes de executar)

```
git clone https://github.com/<SEU USUÁRIO>/micro-livraria.git
```

3. É também necessário ter o Node.js instalado na sua máquina. Se você não tem, siga as instruções para instalação contidas nessa página.

4. Em um terminal, vá para o diretório no qual o projeto foi clonado e instale as dependências necessárias para execução dos microsserviços:

```
cd micro-livraria
npm install
```

5. Inicie os microsserviços através do comando:

```
npm run start
```

6. Para fins de teste, efetue uma requisição para o microsserviço responsável pela API do backend.

- Se tiver o `curl` instalado na sua máquina, basta usar:

```
curl -i -X GET http://localhost:3000/products
```

- Caso contrário, você pode fazer uma requisição acessando, no seu navegador, a seguinte URL: `http://localhost:3000/products`.

7. Teste agora o sistema como um todo, abrindo o front-end em um navegador: http://localhost:5000. Faça então um teste das principais funcionalidades da livraria.

Tarefa Prática #1: Implementando uma Nova Operação

Nesta primeira tarefa, você irá implementar uma nova operação no serviço `Inventory`. Essa operação, chamada `SearchProductByID` vai pesquisar por um produto, dado o seu ID.

Como descrito anteriormente, as assinaturas das operações de cada microsserviço são definidas em um arquivo `.proto`, no caso proto/inventory.proto.

Passo 1

Primeiro, você deve declarar a assinatura da nova operação. Para isso, inclua a definição dessa assinatura no referido arquivo `.proto` (na linha logo após a assinatura da função `SearchAllProducts`):

```
service InventoryService {
    rpc SearchAllProducts(Empty) returns (ProductsResponse) {}
    rpc SearchProductByID(Payload) returns (ProductResponse) {}
}
```

Em outras palavras, você está definindo que o microsserviço `Inventory` vai responder a uma nova requisição, chamada `SearchProductByID`, que tem como parâmetro de entrada um objeto do tipo `Payload` e como parâmetro de saída um objeto do tipo `ProductResponse`.

Passo 2

Inclua também no mesmo arquivo a declaração do tipo do objeto `Payload`, o qual apenas contém o ID do produto a ser pesquisado.

```
message Payload {
    int32 id = 1;
}
```

Veja que `ProductResponse` -- isto é, o tipo de retorno da operação -- já está declarado mais abaixo no arquivo `proto`:

```
message ProductsResponse {
    repeated ProductResponse products = 1;
}
```

Ou seja, a resposta da nossa requisição conterá um único campo, do tipo `ProductResponse`, que também já está implementando no mesmo arquivo:

```
message ProductResponse {
    int32 id = 1;
    string name = 2;
    int32 quantity = 3;
    float price = 4;
    string photo = 5;
    string author = 6;
}
```

Passo 3

Agora você deve implementar a função `SearchProductByID` no arquivo services/inventory/index.js.

Reforçando, no passo anterior, apenas declaramos a assinatura dessa função. Então, agora, vamos prover uma implementação para ela.

Para isso, você precisa implementar a função requerida pelo segundo parâmetro da função `server.addService`, localizada na linha 17 do arquivo services/inventory/index.js.

De forma semelhante à função `SearchAllProducts`, que já está implementada, você deve adicionar o corpo da função `SearchProductByID` com a lógica de pesquisa de produtos por ID. Este código deve ser adicionado logo após o `SearchAllProducts` na linha 23.

```
    SearchProductByID: (payload, callback) => {
        callback(
            null,
            products.find((product) => product.id == payload.request.id)
        );
    },
```

A função acima usa o método `find` para pesquisar em `products` pelo ID de produto fornecido. Veja que:

- `payload` é o parâmetro de entrada do nosso serviço, conforme definido antes no arquivo .proto (passo 2). Ele armazena o ID do produto que queremos pesquisar. Para acessar esse ID basta escrever `payload.request.id`.
- `product` é uma unidade de produto a ser pesquisado pela função `find` (nativa de JavaScript). Essa pesquisa é feita em todos os items da lista de produtos até que um primeiro `product` atenda a condição de busca, isto é `product.id == payload.request.id`.
- products é um arquivo JSON que contém a descrição dos livros à venda na livraria.
- `callback` é uma função que deve ser invocada com dois parâmetros:
    - O primeiro parâmetro é um objeto de erro, caso ocorra. No nosso exemplo nenhum erro será retornado, portanto `null`.
    - O segundo parâmetro é o resultado da função, no nosso caso um `ProductResponse`, assim como definido no arquivo proto/inventory.proto.

Passo 4

Para finalizar, temos que incluir a função `SearchProductByID` em nosso `Controller`. Para isso, você deve incluir uma nova rota `/product/{id}` que receberá o ID do produto como parâmetro. Na definição da rota, você deve também incluir a chamada para o método definido no Passo 3.

Sendo mais específico, o seguinte trecho de código deve ser adicionado na linha 44 do arquivo services/controller/index.js, logo após a rota `/shipping/:cep`.

```
app.get('/product/:id', (req, res, next) => {
    // Chama método do microsserviço.
    inventory.SearchProductByID({ id: req.params.id }, (err, product) => {
        // Se ocorrer algum erro de comunicação
        // com o microsserviço, retorna para o navegador.
        if (err) {
            console.error(err);
            res.status(500).send({ error: 'something failed :(' });
        } else {
```

```
            // Caso contrário, retorna resultado do
            // microsserviço (um arquivo JSON) com os dados
            // do produto pesquisado
            res.json(product);
        }
    });
});
```

Finalize, efetuando uma chamada no novo endpoint da API: http://localhost:3000/product/1

Para ficar claro: até aqui, apenas implementamos a nova operação no backend. A sua incorporação no frontend ficará pendente, pois requer mudar a interface Web, para, por exemplo, incluir um botão "Pesquisar Livro".

IMPORTANTE: Se tudo funcionou corretamente, dê um **COMMIT & PUSH** (e certifique-se de que seu repositório no GitHub foi atualizado; isso é fundamental para seu trabalho ser devidamente corrigido).

```
git add --all
git commit -m "Tarefa prática #1 - Microservices"
git push origin main
```

Tarefa Prática #2: Criando um Container Docker

Nesta segunda tarefa, você irá criar um container Docker para o seu microserviço. Os containers são importantes para isolar e distribuir os microserviços em ambientes de produção. Em outras palavras, uma vez "copiado" para um container, um microsserviço pode ser executado em qualquer ambiente, seja ele sua máquina local, o servidor de sua universidade, ou um sistema de cloud (como Amazon AWS, Google Cloud, etc).

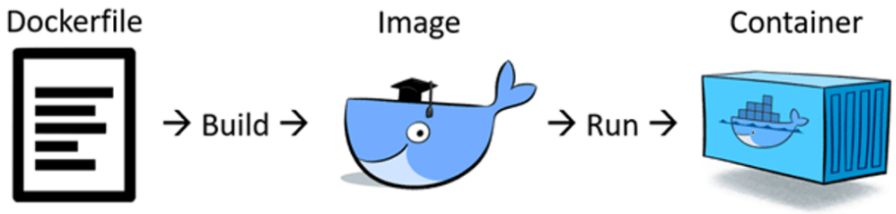
Como nosso primeiro objetivo é didático, iremos criar apenas uma imagem Docker para exemplificar o uso de containers.

Caso você não tenha o Docker instalado em sua máquina, é preciso instalá-lo antes de iniciar a tarefa. Um passo-a-passo de instalação pode ser encontrado na documentação oficial.

Passo 1

Crie um arquivo na raiz do projeto com o nome `shipping.Dockerfile`. Este arquivo armazenará as instruções para criação de uma imagem Docker para o serviço `Shipping`.

Como ilustrado na próxima figura, o Dockerfile é utilizado para gerar uma imagem. A partir dessa imagem, você pode criar várias instâncias de uma aplicação. Com isso, conseguimos escalar o microsserviço de `Shipping` de forma horizontal.



No Dockerfile, você precisa incluir cinco instruções

- `FROM`: tecnologia que será a base de criação da imagem.
- `WORKDIR`: diretório da imagem na qual os comandos serão executados.
- `COPY`: comando para copiar o código fonte para a imagem.
- `RUN`: comando para instalação de dependências.
- `CMD`: comando para executar o seu código quando o container for criado.

Ou seja, nosso Dockerfile terá as seguintes linhas:

```
# Imagem base derivada do Node
FROM node

# Diretório de trabalho
WORKDIR /app

# Comando para copiar os arquivos para a pasta /app da imagem
COPY . /app

# Comando para instalar as dependências
RUN npm install

# Comando para inicializar (executar) a aplicação
CMD ["node", "/app/services/shipping/index.js"]
```

Passo 2

Agora nós vamos compilar o Dockerfile e criar a imagem. Para isto, execute o seguinte comando em um terminal do seu sistema operacional (esse comando precisa ser executado na raiz do projeto; ele pode também demorar um pouco mais para ser executado).

```
docker build -t micro-livraria/shipping -f shipping.Dockerfile ./
```

onde:

- `docker build`: comando de compilação do Docker.
- `-t micro-livraria/shipping`: tag de identificação da imagem criada.
- `-f shipping.Dockerfile`: dockerfile a ser compilado.

O `./` no final indica que estamos executando os comandos do Dockerfile tendo como referência a raiz do projeto.

Passo 3

Antes de iniciar o serviço via container Docker, precisamos remover a inicialização do serviço de Shipping do comando `npm run start`. Para isso, basta remover o sub-comando `start-shipping` localizado na linha 7 do arquivo package.json, conforme mostrado no próximo diff (a linha com o símbolo "-" no início representa a linha original do arquivo; a linha com o símbolo "+" representa como essa linha deve ficar após a sua alteração):

```
diff --git a/package.json b/package.json
index 25ff65c..552a04e 100644
--- a/package.json
+++ b/package.json
@@ -4,7 +4,7 @@
     "description": "Toy example of microservice",
     "main": "",
     "scripts": {
-        "start": "run-p start-frontend start-controller start-shipping start-
inventory",
+        "start": "run-p start-frontend start-controller start-inventory",
         "start-controller": "nodemon services/controller/index.js",
         "start-shipping": "nodemon services/shipping/index.js",
         "start-inventory": "nodemon services/inventory/index.js",
```

Em seguida, você precisa parar o comando antigo (basta usar um CTRL-C no terminal) e rodar o comando `npm run start` para efetuar as mudanças.

Por fim, para executar a imagem criada no passo anterior (ou seja, colocar de novo o microsserviço de `Shipping` no ar), basta usar o comando:

```
docker run -ti --name shipping -p 3001:3001 micro-livraria/shipping
```

onde:

- `docker run`: comando de execução de uma imagem docker.
- `-ti`: habilita a interação com o container via terminal.
- `--name shipping`: define o nome do container criado.
- `-p 3001:3001`: redireciona a porta 3001 do container para sua máquina.
- `micro-livraria/shipping`: especifica qual a imagem deve-se executar.

Se tudo estiver correto, você irá receber a seguinte mensagem em seu terminal:

```
Shipping Service running
```

E o Controller pode acessar o serviço diretamente através do container Docker.

Mas qual foi exatamente a vantagem de criar esse container? Agora, você pode levá-lo para qualquer máquina ou sistema operacional e colocar o microsserviço para rodar sem instalar mais nada (incluindo bibliotecas, dependências externas, módulos de runtime, etc). Isso vai ocorrer com containers implementados em JavaScript, como no nosso exemplo, mas também com containers implementados em qualquer outra linguagem.

IMPORTANTE: Se tudo funcionou corretamente, dê um **COMMIT & PUSH** (e certifique-se de que seu repositório no GitHub foi atualizado; isso é fundamental para seu trabalho ser devidamente corrigido).

```
git add --all
git commit -m "Tarefa prática #2 - Docker"
git push origin main
```

Passo 4

Como tudo funcionou corretamente, já podemos encerrar o container e limpar nosso ambiente. Para isso, utilizaremos os seguintes comandos:

```
docker stop shipping
```

onde:

- `docker stop`: comando para interromper a execução de um container.
- `shipping`: nome do container que será interrompido.

```
docker rm shipping
```

onde:

- `docker rm`: comando para remover um container.
- `shipping`: nome do container que será removido.

```
docker rmi micro-livraria/shipping
```

onde:

- `docker rmi`: comando para remover uma imagem.

- `micro-livraria/shipping`: nome da imagem que será removida.

Comentários Finais

Nesta aula, trabalhamos em uma aplicação baseada em microsserviços. Apesar de pequena, ela ilustra os princípios básicos de microsserviços, bem como algumas tecnologias importantes quando se implementa esse tipo de arquitetura.

No entanto, é importante ressaltar que em uma aplicação real existem outros componentes, como bancos de dados, balanceadores de carga e orquestradores.

A função de um **balanceador de carga** é dividir as requisições quando temos mais de uma instância do mesmo microsserviço. Imagine que o microsserviço de frete da loja virtual ficou sobrecarregado e, então, tivemos que colocar para rodar múltiplas instâncias do mesmo. Nesse caso, precisamos de um balanceador para dividir as requisições que chegam entre essas instâncias.

Já um **orquestrador** gerencia o ciclo de vida de containers. Por exemplo, se um servidor para de funcionar, ele automaticamente move os seus containers para um outro servidor. Se o número de acessos ao sistema aumenta bruscamente, um orquestrador também aumenta, em seguida, o número de containers. Kubernetes é um dos orquestradores mais usados atualmente.

Se quiser estudar um segundo sistema de demonstração de microsserviços, sugerimos este repositório, mantido pelo serviço de nuvem do Google.

Créditos

Este exercício prático, incluindo o seu código, foi elaborado por **Rodrigo Brito**, aluno de mestrado do DCC/UFMG, como parte das suas atividades na disciplina Estágio em Docência, cursada em 2020/2, sob orientação do **Prof. Marco Tulio Valente**.

O código deste repositório possui uma licença MIT. O roteiro descrito acima possui uma licença CC-BY.