Estudo de caso

Neste apêndice um cenário de implementação será proposto e discutido, cobrindo os seguintes aspectos:

- Desenvolvimento de solução inicial de forma monolítica e síncrona.
- Refatoração para uma arquitetura distribuída, incluindo um proxy reverso, ferramenta de service discovery, balanceamento de carga e *circuit breaker*.
- Discussão sobre controle transacional, e refatoração para adicionar mensageria na arquitetura.

O cenário que será implementado é o seguinte:

> *Usuário solicita orçamento para compra de veículos, informando: marca, modelo, ano, quantidade desejada e e-mail de contato. Servidor busca o valor na tabela FIPE usando uma integração, aplica uma regra simples de cálculo de desconto baseada na quantidade, e envia um e-mail para o solicitante.*

A versão monolítica

Para começar, uma versão monolítica será implementada, e posteriormente quebrada em outros serviços (de forma parecida como ocorre na prática). Como a parte de front-end é bem simples, o middleware de arquivos estáticos do próprio express será utilizado, ao invés do framework Angular.

Crie um novo projeto `npm` e instale o express e o pacote `express-formidable` (será usado para adicionar suporte a envio de dados no formato `multipart/form-data`):

```
npm init -y
npm i express express-formidable
```

Implemente uma API básica com o middleware formidable e o de arquivos estáticos, além de um endpoint que recebe os dados para efetivar a cotação, passando-os para um módulo de serviço:

```
const express = require('express');
const formidable = require('express-formidable');
const cotacaoService = require('./cotacao/cotacao.service');


const app = express();


app.use(formidable());
app.use(express.static(`${__dirname}/public`));


app.post('/cotacao', async (req, res) => {
    try {
        await cotacaoService.cotar(req.fields);
        res.status(204).send();
    } catch (err) {
        console.log(err);
        if (err.dadosInvalidos) {
            res.status(400).send();
        } else {
```

```
            res.status(500).send();
        }
    }
});


app.listen(3000);
```

Crie o arquivo `cotacao/cotacao.service.js` mocando seu funcionamento por enquanto:

```
module.exports.cotar = function (solicitacao) {
    return Promise.resolve();
};
```

Crie também um arquivo `public/index.html` com o seguinte código:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cotações</title>
  </head>
  <body>
    <form onsubmit="event.preventDefault(); submeterCotacao(event.target);">
      <p>
        Marca:
        <input name="marca">
      </p>
      <p>
        Modelo:
        <input name="modelo">
      </p>
      <p>
        Ano:
        <input name="ano">
      </p>
      <p>
        Quantidade de veículos:
        <input name="quantidade" type="number">
      </p>
      <p>
        E-mail para retorno:
        <input name="email">
      </p>
      <p>
        <button>Cotar!</button>
      </p>
    </form>
    <script>

      function submeterCotacao(formElement) {
        const data = new FormData(formElement);
        fetch('/cotacao', {
          method: 'POST',
```

```
          body: data
        }).then(resp => {
          if (resp.status == 204) {
            alert('Sua cotação foi solicitada, verifique seu e-mail em alguns
instantes!');
            formElement.reset();
          } else {
            alert('Não foi possível realizar a cotação!');
          }
        }).catch(err => {
          console.log(err);
          alert('Não foi possível realizar a cotação!');
        });
      }

    </script>
  </body>
</html>
```

Execute a API e acesse o endereço `http://localhost:3000` no navegador. Note que o formulário será apresentado e um retorno de sucesso aparecerá ao preenchê-lo e submeter a cotação.

O próximo passo é implementar de fato o processo de cotação, para isso será utilizada a seguinte API pública de obtenção de dados da tabela FIPE: https://deividfortuna.github.io/fipe/. O processo compreende três chamadas:

1. Obter o código da marca através do texto informado.
2. Obter o código do modelo através do texto informado.
3. Obter o valor de referência com base no código do modelo e no ano informado.

Para a implementação das chamadas, ao invés de usar diretamente o pacote `http` (nativo do Node), instale a biblioteca `request` (inspirada na biblioteca Python de mesmo nome):

```
npm i request
```

Altere o arquivo `cotacao.service.js` da seguinte forma para implementá-lo:

```
const request = require('request');


module.exports.cotar = async function (solicitacao) {
    const idMarca = await buscarMarca(solicitacao.marca);
    const idModelo = await buscarModelo(idMarca, solicitacao.modelo);
    const valorFipe = await buscarValor(idMarca, idModelo, solicitacao.ano);
    const valor = calcularValor(valorFipe, solicitacao.quantidade);
    return enviarEmail(valor, solicitacao.email);
};


function buscarMarca(marca) {
    return new Promise((resolve, reject) => {
        request('https://parallelum.com.br/fipe/api/v1/carros/marcas', { json: true },
(err, _, body) => {
            if (err) {
```

```
                reject(err);
                return;
            }
            const registro = body.filter(x => x.nome.toLowerCase() ==
marca.toLowerCase())[0];
            if (registro) {
                resolve(registro.codigo);
            } else {
                reject({ dadosInvalidos: true, detalhes: 'Marca não encontrada' });
            }
        });
    });
}


function buscarModelo(idMarca, modelo) {
    return new Promise((resolve, reject) => {

request(`https://parallelum.com.br/fipe/api/v1/carros/marcas/${idMarca}/modelos`, {
json: true }, (err, _, body) => {
            if (err) {
                reject(err);
                return;
            }
            const registro = body.modelos.filter(x => x.nome.toLowerCase() ==
modelo.toLowerCase())[0];
            if (registro) {
                resolve(registro.codigo);
            } else {
                reject({ dadosInvalidos: true, detalhes: 'Modelo não encontrado' });
            }
        });
    });
}


function buscarValor(idMarca, idModelo, ano) {
    return new Promise((resolve, reject) => {

request(`https://parallelum.com.br/fipe/api/v1/carros/marcas/${idMarca}/modelos/${idMod
 { json: true }, (err, resp, body) => {
            if (err) {
                reject(err);
                return;
            }
            if (resp.statusCode == 404) {
                reject({ dadosInvalidos: true, detalhes: 'Ano não encontrado' });
                return;
            }
            const valor = parseFloat(body.Valor
                .substring("R$ ".length)
                .replace('.', '')
```

```
                .replace(',', '.'));
            resolve(valor);
        });
    });
}


function calcularValor(valorFipe, quantidade) {
    // 1% por unidade, máximo de 5%
    return valorFipe * (1 - Math.min(0.05, (0.01 * quantidade)));
}


function enviarEmail(valor, email) {
    console.log(valor, email);
    return Promise.resolve();
}
```

Se desejar, Use os seguintes dados para testar:

- Marca: `Toyota`
- Modelo: `Hilux CD 4x4 2.8 Diesel Mec.`
- Ano: `2019-3`

O último passo é o envio do e-mail com o resultado da cotação. Instale a biblioteca `nodemailer` para isso, e use o serviço de teste SMTP `ethereal.email` que possui uma ótima integração com essa ferramenta:

```
npm i nodemailer
```

```
const emailService = require('../email/email.service');
// ...
function enviarEmail(valor, email) {
    return emailService.enviar('Sua cotação foi processada',
        `Valor da cotação: ${valor}`, email);
}
```

```
const nodemailer = require('nodemailer');


let _transporter;

async function getTransporter() {
    if (!_transporter) {
        const testAccount = await nodemailer.createTestAccount();
        _transporter = nodemailer.createTransport({
            host: "smtp.ethereal.email",
            port: 587,
            secure: false,
            auth: {
              user: testAccount.user,
              pass: testAccount.pass
            }
```

```
        });
    }
    return _transporter;
}

module.exports.enviar = async function (assunto, texto, para) {
    const transporter = await getTransporter();
    let info = await transporter.sendMail({
        from: '"Equipe de Cotação" <contato@exemplo.com>',
        to: para,
        subject: assunto,
        text: texto
    });
    console.log("URL de pré-visualização: %s", nodemailer.getTestMessageUrl(info));
};
```

A refatoração

Agora que você possui uma versão monolítica da solução, considere as seguintes mudanças arquiteturais:

- Adição de proxy reverso/service discovery/balanceador: Traefik. Isso permitirá a escalabilidade horizontal.
- Criação de componente separado para envio de e-mails.
- Uso de Circuit Breaker: Opossum. Isso permite que a aplicação continue resiliente face a queda do componente de envio de e-mails.
- Adição de suporte para envio de e-mail usando mensageria (RabbitMQ).

No restante deste material você irá implementá-las, uma a uma. Note que todos os componentes podem ser instalados manualmente, mas containeres Docker serão utilizados em alguns casos para facilitar o processo.

Traefik

O papel de um Proxy Reverso é fazer a ponte entre as requisições externas (ex: da Internet) e os serviços internos da sua solução. O Traefik [1] é uma solução moderna de Proxy Reverso, repleta de funcionalidades.

Uma vez instalado (siga as instruções no site da ferramenta e garanta que `traefik -h` esteja funcionando no seu terminal) crie um arquivo de configuração na raíz do projeto, chamado `traefik.toml`, com o seguinte conteúdo:

```
[global]
  checkNewVersion = true
  sendAnonymousUsage = true

[entryPoints]
  [entryPoints.web]
    address = ":8081"
  [entryPoints.traefik]
    address = ":8082"

[log]
```

```
[api]

[ping]

[file]

[backends]
  [backends.monolito]
    [backends.monolito.servers]
      [backends.monolito.servers.server0]
        url = "http://127.0.0.1:3000"
        weight = 1

[frontends]
  [frontends.frontend1]
    entryPoints = ["web"]
    backend = "monolito"
    passHostHeader = true
    priority = 42
    [frontends.frontend1.routes]
      [frontends.frontend1.routes.route0]
        rule = "Host:monolito.localhost"
```

Adicione uma entrada para `monolito.localhost` no seu arquivo de hosts. Depois disso, execute o serviço com o seguinte comando:

```
traefik --configFile=traefik.toml
```

Teste se funcionou acessando a URL `http://monolito.localhost:8081`. Abra o endereço `http://localhost:8082/` no navegador para acessar o dashboard.

[1] https://traefik.io/

Separação do envio de e-mail

O próximo passo é separar o componente de envio de e-mail. A API do monolito irá chamar esse novo componente usando a biblioteca `request`.

Crie um novo projeto usando `npm init`, e adicione as dependências `express` e `nodemailer`:

```
npm init -y
npm i express nodemailer
```

Mova o diretório `email` do monolito para este novo projeto, e crie um arquivo `index.js` nele com o seguinte conteúdo:

```
const express = require('express');
const emailService = require('./email/email.service');


const app = express();
app.use(express.json());
```

```
app.post('/enviar', async (req, res) => {
    try {
        const { assunto, texto, para } = req.body;
        await emailService.enviar(assunto, texto, para);
        res.status(204).send();
    } catch (err) {
        res.status(500).send();
    }
});


app.listen(3001);
```

Adicione esse novo serviço no traefik (note o uso de um novo entry point, voltado para uso interno apenas):

```
[global]
  checkNewVersion = true
  sendAnonymousUsage = true

[entryPoints]
  [entryPoints.web]
    address = ":8081"
  [entryPoints.traefik]
    address = ":8082"
  [entryPoints.svc]
    address = ":8083"
    [entryPoints.svc.whitelist]
      sourceRange = ["127.0.0.1/32", "::1/128"]

[log]

[api]

[ping]

[file]

[backends]
  [backends.monolito]
    [backends.monolito.servers]
      [backends.monolito.servers.server0]
        url = "http://127.0.0.1:3000"
        weight = 1
  [backends.emails]
    [backends.emails.servers]
      [backends.emails.servers.server0]
        url = "http://127.0.0.1:3001"
        weight = 1

[frontends]
  [frontends.frontend1]
```

```
    entryPoints = ["web"]
    backend = "monolito"
    passHostHeader = true
    priority = 42
    [frontends.frontend1.routes]
      [frontends.frontend1.routes.route0]
        rule = "Host:monolito.localhost"
  [frontends.frontend2]
    entryPoints = ["svc"]
    backend = "emails"
    passHostHeader = true
    priority = 42
    [frontends.frontend2.routes]
      [frontends.frontend2.routes.route0]
        rule = "Host:emails.localhost"
```

Adicione a entrada `emails.localhost` no seu arquivo de hosts.

Ajuste agora o arquivo `cotacao.service.js` para consumir esse novo serviço:

```
function enviarEmail(valor, email) {
    return new Promise((resolve, reject) => {
        request(`http://emails.localhost:8083/enviar`, {
            json: true,
            method: 'POST',
            body: {
                assunto: 'Sua cotação foi processada',
                texto: `Valor da cotação: ${valor}`,
                para: email
            }
        }, (err, resp) => {
            if (err) {
                reject(err);
                return;
            }
            if (resp.statusCode != 204) {
                reject(`${resp.statusCode} ${resp.statusMessage}`);
                return;
            }
            resolve();
        });
    });
}
```

Circuit Breaker

O que acontece se o componente de envio de e-mail sair do ar? O monolito continuará realizando chamadas para ele, gastando recursos desnecessariamente. Além disso, não adianta nada o cálculo todo ser executado se o e-mail não puder ser enviado no final. Uma forma de resolver este problema é através do uso de um circuit breaker, que é um componente de código que identifica recorrência na falha de execução e, caso ela atinja um determinado limite configurável, começa a retornar falha automaticamente sem a necessidade de chamar o recurso real.

No caso do Node uma das implementações desse conceito é a biblioteca Opossum [1].

Para utilizá-la, instale-a primeiro usando `npm`:

```
npm i opossum
```

Altere agora o arquivo `cotacao.service.js` da seguinte maneira:

```
const circuitBreaker = require('opossum');


const cotar = circuitBreaker(async function (solicitacao) {
    const idMarca = await buscarMarca(solicitacao.marca);
    const idModelo = await buscarModelo(idMarca, solicitacao.modelo);
    const valorFipe = await buscarValor(idMarca, idModelo, solicitacao.ano);
    const valor = calcularValor(valorFipe, solicitacao.quantidade);
    return enviarEmail(valor, solicitacao.email);
}, {
    errorThresholdPercentage: 50
});
module.exports.cotar = solicitacao => cotar.fire(solicitacao);
```

[1] https://github.com/nodeshift/opossum

Mensageria

Ao invés de efetuar uma chamada HTTP e aguardar a resposta, uma alternativa para comunicação entre microsserviços é o uso de mensageria, onde o componente solicitante registra a mensagem em uma fila/tópico específico e o componente responsável por executar o processo *escuta* por mensagens nessa fila/tópico.

Para exemplificar o uso de mensageria, comece instalando uma instância de RabbitMQ usando Docker:

```
docker run -d -p 5672:5672 -p 15672:15672 --name posdesenvweb-rabbit rabbitmq:3-
management
```

Instale agora a biblioteca `amqplib` nos dois projetos (AMQP é um protocolo de mensageria suportado pelo RabbitMQ):

```
npm i amqplib
```

Implemente o consumo de mensagens no arquivo `index.js` do projeto de envio de e-mails:

```
// ...
const amqplib = require('amqplib');
// ...
amqplib.connect('amqp://localhost:5672')
    .then(conn => conn.createChannel())
    .then(async ch => {
        await ch.assertQueue('emails');
        return ch.consume('emails', msg => {
            if (msg !== null) {
                const { assunto, texto, para } = JSON.parse(msg.content.toString());
```

```
                emailService.enviar(assunto, texto, para).then(() => ch.ack(msg));
            }
        });
    });
```

Para testar, acesse o painel do RabbitMQ no endereço `http://localhost:15672`, vá até a parte de filas, encontre a fila `emails` e envie uma mensagem de teste.

Por fim implemente o envio de mensagem no arquivo `cotacao.service.js` no monolito.

```
// ...
const amqplib = require('amqplib');
// ...
function enviarEmail(valor, email) {
    return amqplib.connect('amqp://localhost:5672')
        .then(conn => conn.createChannel())
        .then(async ch => {
            await ch.assertQueue('emails');
            return ch.sendToQueue('emails', Buffer.from(JSON.stringify({
                assunto: 'Sua cotação foi processada',
                texto: `Valor da cotação: ${valor}`,
                para: email
            })));
        });
}
```

Uma possível melhoria que não será abordada neste material é um tratamento de erros mais robusto nos casos onde o consumidor de mensagem falhar durante a sua execução.