

Arquiteturas de Sistemas Computacionais

Professor Rodrigo Bossini

Conteúdo

1	Introdução	1
2	Novidades e atualizações	2
2.1	Express 4.16 e o pacote <code>body-parser</code>	2
2.2	NodeJS 15 e Promises rejeitadas e não tratadas	3
3	Arquitetura de Software	5
3.1	Definição acadêmica	5
3.2	Definição no mercado de trabalho	6
4	Microsserviços	9
4.1	Arquitetura Monolítica	9
4.2	Arquitetura baseada em microsserviços	10
4.2.1	E os dados?	11
4.2.2	Comunicação entre microsserviços	12
4.2.3	Comunicação síncrona	12
4.2.4	Comunicação assíncrona - barramento de eventos	13
4.2.5	Comunicação assíncrona - base de dados em função das demais	14
4.3	Construindo uma aplicação do zero com microsserviços	16
4.3.1	Visão geral	16
4.3.2	Quais microsserviços implementar?	17
4.3.3	Implementando o microsserviço de lembretes	18
4.3.4	Workspace	18
4.3.5	O microsserviço de lembretes	18
4.3.6	Pacotes	18
4.3.7	O microsserviço de observações	19
4.3.8	Requisições e métodos HTTP do microsserviço de lembretes	19
4.3.9	Código inicial do microsserviço de lembretes	19
4.3.10	Base inicialmente volátil para o microsserviço de lembretes .	20
4.3.11	Requisição GET: Devolvendo a coleção de lembretes	20
4.3.12	Requisição PUT: Geração de id e criação de lembrete	21
4.3.13	Executando o servidor com nodemon	21
4.3.14	Testes com Postman	22
4.3.15	Organizando as requisições em uma coleção	24
4.3.16	Implementando o microsserviço de observações	25

4.3.17	Código inicial do microsserviço de observações	26
4.3.18	Base inicialmente volátil para o microsserviço de observações	26
4.3.19	Gerando códigos UUID	27
4.3.20	Requisição PUT: Inserindo uma nova observação	28
4.3.21	Testando inserções de lembretes e observações	28
4.3.22	Requisição GET: Devolvendo a lista de observações de um lembrete	30
4.3.23	Teste para a obtenção da lista de observações	31
4.3.24	Busca da coleção de lembretes incluindo observações: $n + 1$ requisições feitas pelo cliente	31
4.3.25	Busca da coleção de lembretes incluindo observações: comunicação síncrona	32
4.3.26	Busca da coleção de lembretes incluindo observações: comunicação assíncrona	33
4.3.27	Uma implementação manual de um barramento de eventos .	34
4.3.28	Barramento de eventos - criando o projeto	36
4.3.29	Emissão de eventos a partir do microsserviço de lembretes .	38
4.3.30	Testando a emissão de eventos de inserção de lembretes . .	39
4.3.31	Emissão de eventos a partir do microsserviço de observações	40
4.3.32	Testando a emissão de eventos de inserção de observações .	41
4.3.33	Recebendo eventos nos microsserviços de lembretes e de observações	42
4.3.34	Testes de inserção de lembretes e observações após implementação dos endpoints	42
4.3.35	O microsserviço de consulta	43
4.3.36	A base de dados do microsserviço de consulta	45
4.3.37	Atualização da base do microsserviço de consulta	46
4.3.38	Barramento de eventos direcionando eventos ao microsserviço de consulta	47
4.3.39	Testando o microsserviço de consulta	47
4.3.40	Classificação de observações	48
4.3.41	O microsserviço de classificação	52
4.3.42	Microsserviço de observação registra status para cada nova observação	54
4.3.43	Barramento de eventos entrega eventos ao microsserviço de classificação	55
4.3.44	Microsserviço de classificação trata eventos recebidos . . .	56
4.3.45	Microsserviço de observações lida com eventos emitidos pelo microsserviço de classificação	57
4.3.46	Microsserviço de consulta lida com eventos de observações atualizadas após classificação	58
4.3.47	Microsserviços descartam eventos que não lhes são de interesse	59
4.3.48	Testando a solução completa após descartes de eventos . . .	60
4.3.49	Como lidar com eventos perdidos?	63

4.3.50 Barramento de eventos armazena eventos	66
4.3.51 Microsserviço de consulta solicita eventos potencialmente perdidos	67
4.3.52 Testando a solução após implementação do tratamento de eventos perdidos	68
4.3.53 Implantação	68
4.3.54  docker	72
4.3.55 Obtendo o Docker	75
4.3.56 Hello,  docker !	77
4.3.57 Comandos  docker	82
4.3.58 Implantando o microsserviço de lembretes  docker	82
4.3.59 Testando o microsserviço de lembretes com  docker	85
4.3.60 Implantando os demais microsserviços com  docker	85
4.3.61 Testando os demais microsserviços com  docker	86
4.3.62  kubernetes	87
4.3.63 Arquitetura do  kubernetes	88
4.3.64 Detalhes sobre o control plane	89
4.3.65 Detalhes sobre os nós	90
4.3.66 Instalação do  kubernetes : Windows	90
4.3.67 Instalação do  kubernetes : Linux & MacOS	92
4.3.68 Hello,  kubernetes !	92
4.3.69 Definindo objetos do  kubernetes com arquivos de configuração	113
4.3.70 Atualizando a imagem usada por um <i>deployment</i>	119
4.3.71 Serviço para acessar o microsserviço de lembretes	125
4.3.72 Comunicação interna entre os microsserviços	127
5 Instalação do	139
5.1 Instalação com o instalador regular do NodeJS	139
5.2 Instalação usando um gerenciador de versões	139
Referências	141

Capítulo 1

Introdução

Neste material são abordados conceitos relacionados à **Arquitetura de Software**. O primeiro passo é fazer uma definição precisa, apropriada para o contexto desejado. Essa definição caracteriza aquilo que se espera de um **Arquiteto de Software**. A seguir, em uma abordagem prática, o material trata dos seguintes tópicos.

- Padrões Arquiteturais. Padrão Arquitetural REST.
- Serviços Restful.
- Arquitetura cliente/servidor.
- Microsserviços.
- Containers e orquestração.

Capítulo 2

Novidades e atualizações

Este material apresenta conceitos sobre arquiteturas de software de maneira prática, com considerável ênfase na codificação de exemplos que ilustram as suas principais características. A construção destes exemplos, naturalmente, se dá utilizando diversos pacotes de software que trazem alto nível de abstração, viabilizando maior enfoque nos aspectos considerados mais importantes. Bons pacotes de software são atualizados constantemente por equipes de desenvolvedores. É esperado que, de tempos em tempos, novas versões de pacotes de software que utilizamos sejam disponibilizadas, possivelmente trazendo novidades para o nosso ambiente. Algumas delas podem trazer novos recursos. Outras, podem fazer com que algumas funcionalidades que estamos utilizando deixem de existir, o que quer dizer que aderir ao uso de versões mais recentes pode significar ter de fazer alterações no código. Outros tipos de atualizações de pacotes de software meramente trazem avisos que indicam que determinadas funcionalidades passaram, a partir de certo momento, a serem consideradas obsoletas - em geral, elas são marcadas como **deprecated**. Isso pode fazer com que nosso ambiente de desenvolvimento produza mensagens nos informando a esse respeito, o que não necessariamente causa falhas ou mal funcionamento. Cada seção deste capítulo trata de uma atualização específica referente a um pacote de software utilizado ao longo do restante do material, as quais podem ser utilizadas a partir do momento que for mais interessante para cada projeto.

2.1 Express 4.16 e o pacote `body-parser` Desde a versão 4.16.0 do Express, cuja disponibilização se deu em 28/09/2017, o framework inclui um middleware próprio para a manipulação de objetos JSON. Veja o Link 2.1.1.

Link 2.1.1

<https://expressjs.com/en/changelog/4x.html#4.16.0>

Segundo a documentação, este novo middleware do Express faz uso do pacote `body-parser` internamente. Isso quer dizer que o middleware provido pelo pacote `body-parser` pode ser substituído por aquele que é nativo do Express. Veja o que a documentação fala a esse respeito.

The `express.json()` and `express.urlencoded()` middleware have been added to provide request body parsing support out-of-the-box. This uses the `expressjs/body-parser` module module underneath, so apps that are currently requiring the module separately can switch to the built-in parsers.

A substituição é ilustrada no Bloco de Código 2.1.1.

Bloco de Código 2.1.1

```
1 //o que costumava ser assim
2 const express = require("express");
3 const bodyParser = require("body-parser");
4 const app = express();
5 app.use(bodyParser.json());

6
7 //pode agora ser assim
8 const express = require ('express');
9 const app = express();
10 app.use(express.json());
```

Além disso, quando inspecionamos o arquivo - `index.d.ts` - em que a definição do pacote `body-parser` é feita, encontramos o conteúdo do Bloco de Código 2.1.2.

Bloco de Código 2.1.2

```
1 . . .
2 /** @deprecated */
3 declare function bodyParser(
4     options?: bodyParser.OptionsJson & bodyParser.OptionsText
      & bodyParser.OptionsUrlencoded,
5 ): NextHandleFunction;
6 . . .
```

A configuração `@deprecated` pode fazer com que a exibição da linha em que o uso da função `bodyParser` ocorre se dê como na Figura 2.1.1.

Figura 2.1.1

```
const express = ...require("express");
const bodyParser = require("body-parser");
const app = express();
app.use(bodyParser.json());
```

2.2 NodeJS 15 e Promises rejeitadas e não tratadas A versão 15 do NodeJS se comporta de maneira diferente diante de promises rejeitadas não tratadas, ou

seja, para as quais não há um bloco `catch`. Antes dela, o comportamento do NodeJS era exibir uma mensagem de **warning**, como mostra a Figura 2.2.1.

Figura 2.2.1

```
(node:1688) UnhandledPromiseRejectionWarning: Unhandled promise rejection.  
This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node process on unhandled promise rejection, use the CLI flag `--unhandled-rejections=strict` (see https://nodejs.org/api/cli.html#cli\_unhandled\_rejections\_mode). (rejection id: 2)  
(node:1688) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.
```

A partir da versão 15, o NodeJS deixa de exibir uma mensagem de warning e passa a causar um erro.

O Bloco de Código 2.2.1 mostra como resolver o problema. Claro, o tratamento a ser realizado no bloco `catch` depende do projeto em que estiver trabalhando.

Bloco de Código 2.2.1

```
1 axios.post("http://host:porta/endpoint", evento)  
2     .catch((err) => {  
3         console.log("err", err);  
4     });
```

Capítulo 3

Arquitetura de Software

Há diferentes definições para **Arquitetura de Software**. Diferentes autores produzem definições sutilmente diferentes. É óbvio que a definição deve estar de acordo com aquilo que se pretende abordar no texto. Há, também, as definições advindas do mercado de trabalho. Elas são importantes pois caracterizam as funções que espera-se sejam desempenhadas por um **Arquiteto de Software**. Não se trata de uma ciência exata e não há definição correta ou incorreta. Há definições diferentes, cada uma apropriada para um contexto específico.

3.1 Definição acadêmica A seguir, apresentamos algumas definições para Arquitetura de Software oferecidas por diferentes autores.

DEFINIÇÃO

Segundo Armando Fox et. al., a **Arquitetura de Software** descreve como os sub-sistemas que constituem um software interagem entre si a fim de tornar disponíveis os requisitos funcionais e não funcionais[3].

DEFINIÇÃO

Segundo Len Bass et. al., a **Arquitetura de Software** de um sistema é o conjunto de estruturas necessário para que se possa refletir sobre ele, o que abrange elementos de software, relações entre eles e as propriedades de ambos.[1].

DEFINIÇÃO

Segundo Mark Richards, et. al. a **Arquitetura de Software** é caracterizada pela **estrutura** do sistema, combinada com as **características arquiteturais** às quais o sistema deve dar suporte, pelas **decisões arquiteturais** e pelos **princípios de design**. Nesta definição

- a **estrutura do sistema** diz respeito ao estilo arquitetural utilizado em sua implementação (como **microsserviços**, em **camadas** etc).
- as **características arquiteturais** definem aquilo que é fundamental para o sucesso do sistema, como **escalabilidade**, **tolerância a falhas**, **desempenho** etc.
- as **decisões arquiteturais** definem as regras segundo as quais o sistema deve ser construído. Por exemplo, pode-se especificar que somente componentes da camada de serviço têm acesso direto à base de dados.
- os **princípios de design** são semelhantes às decisões arquiteturais. Entretanto, não são regras absolutas. Tratam-se apenas de recomendações que podem ser aplicadas pelos desenvolvedores em situações específicas. Por exemplo, fazer uso de troca de mensagens assíncronas entre serviços para melhorar o desempenho, sempre que possível.

Finalmente, Martin Fowler escreveu um famoso artigo chamado **Who Needs an Architect?**^[4] que vale olhar. Parte do texto chega à seguinte conclusão.

DEFINIÇÃO

A **Arquitetura de Software** é caracterizada pelas coisas importantes. Seja lá o que elas forem.

3.2 Definição no mercado de trabalho O profissional “Arquiteto de Software” tem grande demanda no mercado. Buscas em portais de vagas conhecidos trazem inúmeras oportunidades para esse tipo de profissional, algumas vezes com sutis variações no nome. Ocorre que as necessidades de cada empresa variam em função da natureza de suas atividades. Por isso, **caracterizar a Arquitetura de Software com base naquilo que se vê no mercado é tão dependente de contexto**

quanto o que ocorre na definição mais acadêmica. Vejamos alguns exemplos de oportunidades para esse tipo de profissional, especificamente no Brasil¹.

- O Arquiteto de Soluções e Softwares deve ter experiência em múltiplos ambientes de hardware e software e estar confortável com ambientes de sistemas heterogêneos complexos. Deve ser um tecnocrata sênior altamente experiente na liderança e definição arquitetural de soluções e softwares alinhadas às necessidades de negócio. O profissional deve ter capacidade de partilhar e comunicar ideias com clareza, oralmente e por escrito à equipe executiva, aos patrocinadores e as equipes técnicas envolvidas no projeto.
- Necessário sólida experiência em desenvolvimento (fullstack), arquitetura e processos em várias tecnologias. Vivências como líder de time, prática com arquiteto de aplicações para soluções WEB e de Micros serviços. Conhecimento em arquitetura Cloud (Azure/ AWS). Conhecimentos em ASP.NET e C#, outras API Web.
- Desenvolvemos soluções digitais sob medida para o negócio do cliente. Nossa metodologia de trabalho tem sua base na Experiência do Usuário (UX) e o Design Thinking para a concepção de soluções, desenvolvimento pautado nas metodologias ágeis com entregáveis recorrentes resultando em soluções robustas, de alta performance, escaláveis e compliance com requisitos de segurança. Buscamos um profissional que terá responsabilidades desde nível de negócio (Engenharia de Software) até Arquitetura de Software e DevOps/Infra com mais de 3 anos de experiência na função.
- Experiência em definição de arquitetura de soluções de alta disponibilidade (HA), escaláveis e microserviços, desenvolvimento de interfaces através da construção de API. Linguagens: NodeJS, ReactJS, React Native. Lógica de programação avançada. Metodologias ágeis. Acompanhar o mercado e as novas tecnologias. Hands-on na fase de codificação.

Note como a definição também pode variar bastante. Algumas delas, observe, têm bastante relação com as definições encontradas em livros. Note também que, muitas vezes, um profissional que levaria o título de **Desenvolvedor Sênior** em uma empresa pode, em outra, ser considerado um **Arquiteto de Software** ou **Arquiteto de Sistemas**. O mesmo ocorre com outros títulos como **Engenheiro de Software** e similares. Muitas oportunidades deixam claro que o profissional deve ter **condições de colocar a mão na massa!**. Ou seja, conhecer diversas linguagens de programação e, de fato, programar. Isso não é diferente com oportunidades

¹Busca realizada no site [APINFO](#) em fevereiro de 2021.

encontradas em outros países². Veja alguns exemplos.

- Docker/Kubernetes, Cloud (AWS, GCP), Data Engineering, DevOps, ML background, Python, ML Ops tools - ML Flow/Kubeflow, ML frameworks - TensorFlow/PyTorch/Keras, ML Testing. Experienced in Python, SQL. Experience with Hadoop infra and NoSQL databases like Cassandra, MongoDB. Experience in building docker container-based solutions – Docker, Kubernetes.
- Experience with cloud/SaaS, big data, or analytics and Machine Learning. Excellent communication skills: Demonstrated ability to present to all levels of leadership, including executives. Expertise with modern technology stacks, microservices, public cloud and programming languages. Familiarity with the FinTech and how technology can be used to solve problems in the personal finance space. Expertise with Agile Development, SCRUM, or Extreme Programming methodologies.
- Application Architect on a team; primary role to design and develop secure web-based applications. Establish work necessary for a complete product; break work down into discrete tasks and deliverables. Code, troubleshoot, test, and maintain core product software and databases, to ensure strong optimization and secure functionality. Knowledge of software architecture and design patterns, and the ability to apply them. Experience in web-based technologies like Microsoft.Net, ASP.NET, JavaScript, C#, VB.Net, Web API and complementary business layer and front-end technologies. Strong problem-solving skills.

²Busca realizada no site [INDEED](#) em fevereiro de 2021.

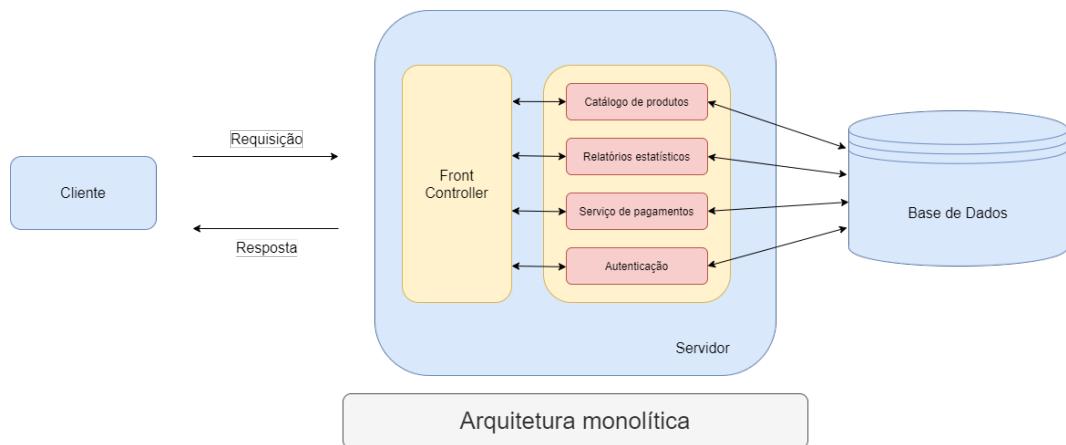
Capítulo 4

Microsserviços

Neste capítulo trataremos de um dos tipos de arquiteturas mais utilizados atualmente: a **Arquitetura baseada em microsserviços**.

4.1 Arquitetura Monolítica Para entender o que é uma **Arquitetura baseada em microsserviços**, vamos falar de um outro tipo de arquitetura, muitas vezes denominada **Arquitetura Monolítica**. Esse tipo de arquitetura foi muito utilizado nas últimas décadas e ainda o é nos dias atuais. Muito embora o seu uso possa ser empregado ao mesmo tempo em que boas práticas de programação - como o uso de **Design Patterns** - são utilizadas, alguns problemas são inerentes à sua natureza. A Figura 4.1.1 mostra um exemplo típico.

Figura 4.1.1



A aplicação exibida na Figura 4.1.1 possui diferentes funcionalidades.

- Exibição de catálogo de produtos.
- Geração de relatórios estatísticos.
- Gerenciamento e realização de pagamentos.
- Serviço de autenticação.

Ela está relativamente bem organizada. Possivelmente foi implementada utilizando-se algum padrão composto como o **MVC** ou **MVVM**. Seus componentes de software internos são, possivelmente, altamente coesos e pouco acoplados. Entretanto, a visão geral do sistema mostra um problema que pode ser grave. Todas as funcionalidades fazer parte de uma coisa só: caso seja necessário fazer ajustes no serviço de pagamento, por exemplo, é necessário fazer uma nova implantação do sistema inteiro. Caso o serviço de autenticação deixe de funcionar, possivelmente a aplicação inteira ficará indisponível.

4.2 Arquitetura baseada em microsserviços Podemos, assim, definir o conceito de **Arquitetura baseada em microsserviços**.

DEFINIÇÃO

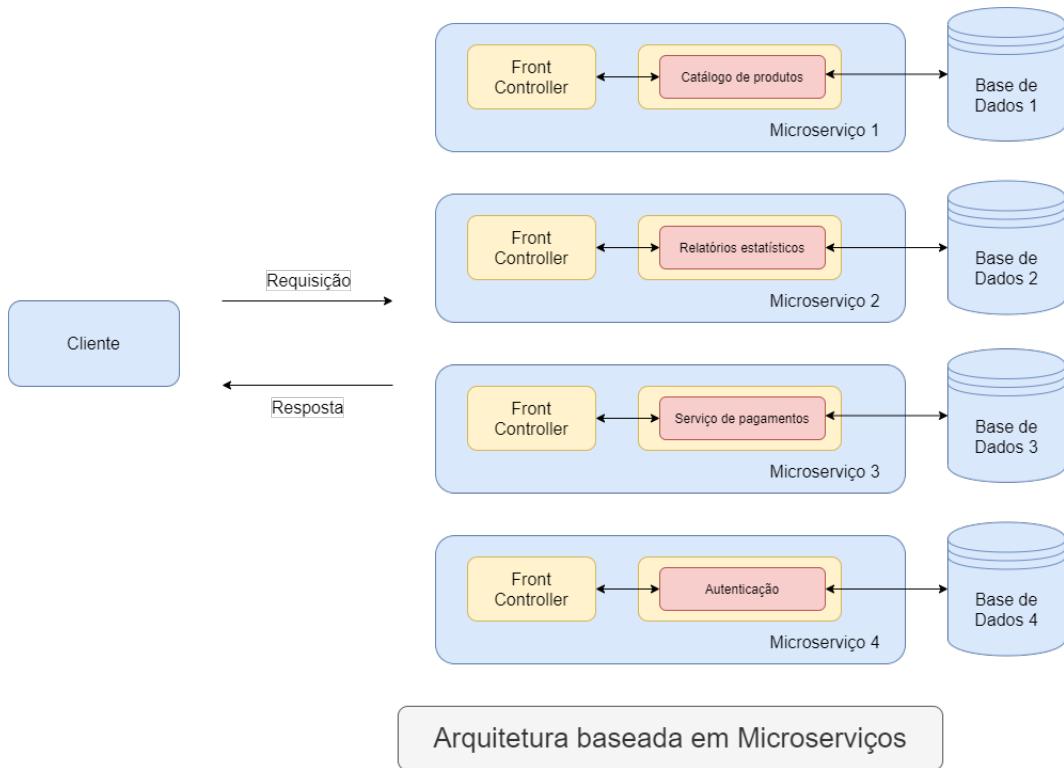
Uma **Arquitetura baseada em microsserviços** é constituída de pequenos serviços independentes, cada qual responsável por uma única funcionalidade do sistema. Os microsserviços se comunicam entre si por meio de uma interface bem definida. Quando um microsserviço fica indisponível, os demais continuam operando normalmente. Em geral, um microsserviço é desenvolvido e mantido por uma única equipe de desenvolvimento.

Os serviços de **Computação em Nuvem** estão intimamente relacionados ao uso de arquiteturas baseadas em microsserviços. Visite o Link 4.2.1 para conhecer a definição proposta pela **Amazon**.

Link 4.2.1
<https://aws.amazon.com/pt/microservices/>

A Figura 4.2.1 mostra a ideia geral de um sistema cuja implementação utiliza a Arquitetura baseada em microsserviços.

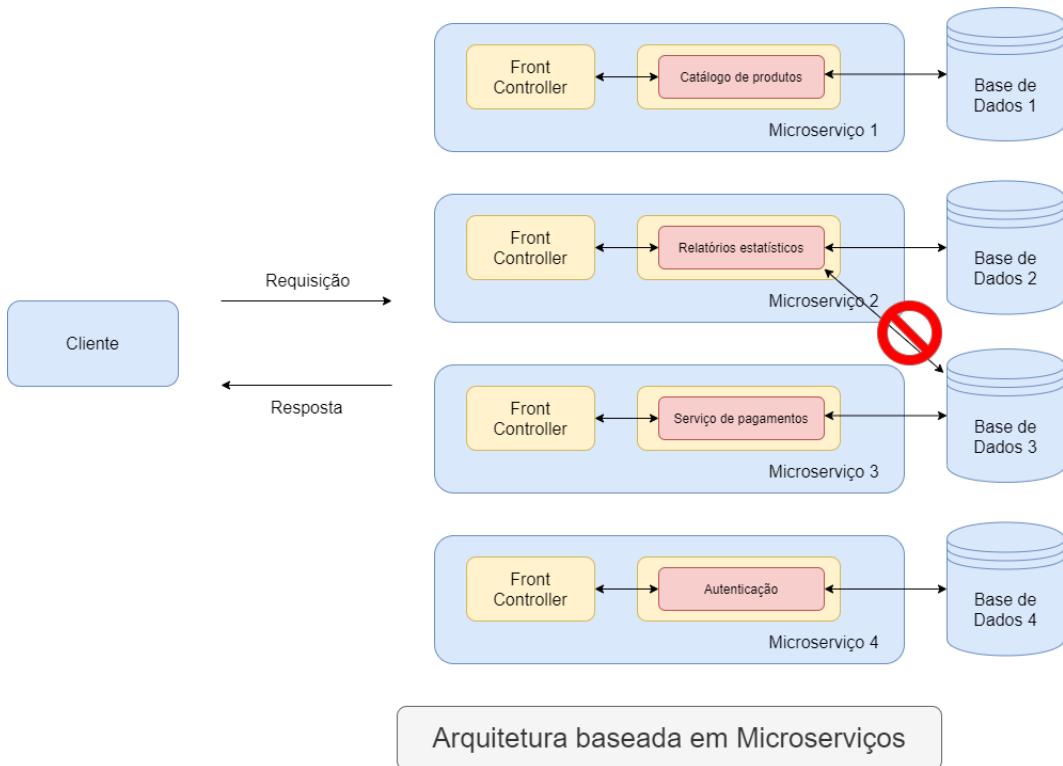
Figura 4.2.1



4.2.1 E os dados? Devido à independência entre os microserviços sugerida pelas definições apresentadas, o uso de uma base de dados independente para cada um deles parece natural. Idealmente, **um microserviço jamais acessa a base de dados de outro**. Isso ocorre pois **o esquema de cada base pode ser alterado a qualquer momento** e, além disso, cada microserviço pode ter um **tipo de esquema (relacional ou NoSQL, por exemplo) mais apropriado** para a sua finalidade.

Entretanto, essa abordagem tem aspectos que podem comprometer a desejada simplicidade para o desenvolvimento do sistema. Considere, ainda, a proposta de arquitetura da Figura 4.2.2.

Figura 4.2.2

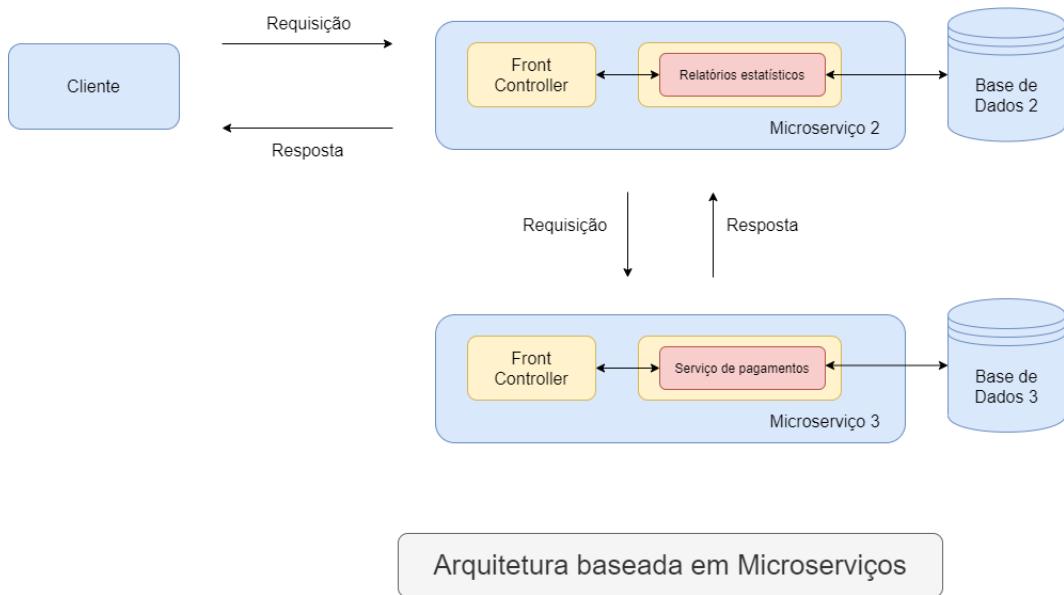


A sua funcionalidade **Relatórios Estatísticos** depende de dados existentes nas bases de dados pertencentes a outros microserviços. Como essa funcionalidade poderia ser implementada sem o acesso direto a essas bases? Os microserviços precisam se comunicar de alguma forma.

4.2.2 Comunicação entre microserviços Para resolver esse tipo de problema, os microserviços precisam se comunicar de alguma forma. Há dois tipos essenciais de comunicação entre microserviços: o **síncrono** e o **assíncrono**.

4.2.3 Comunicação síncrona Na **comunicação síncrona**, um microserviço realiza uma requisição direta a outro e fica no aguardo da resposta. Veja a Figura 4.2.3.

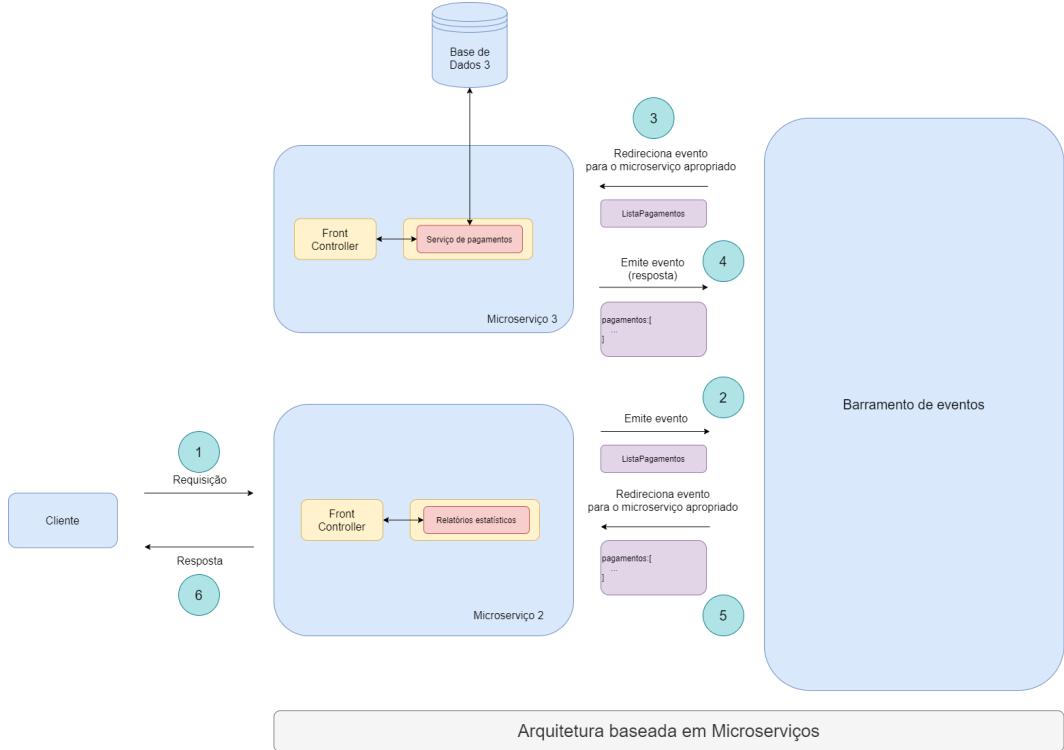
Figura 4.2.3



Em qualquer modelo de comunicação, há vantagens e desvantagens a serem consideradas. A **comunicação síncrona**, por exemplo, traz como vantagem o acesso indireto a bases de dados. Inclusive, pode ser o caso de o microsserviço em questão sequer precisar de uma. Por outro lado, é natural a **dependência entre os microsserviços**. Neste exemplo, caso o Serviço de Pagamentos fique indisponível, o microsserviço de Relatórios Estatísticos também deixará de funcionar.

4.2.4 Comunicação assíncrona - barramento de eventos A **comunicação assíncrona**, por sua vez, tem algumas formas de implementação. Uma delas é baseada em **eventos**. Seu funcionamento se dá em função de um **barramento de eventos**. Veja a Figura 4.2.4.

Figura 4.2.4



Neste modelo de comunicação assíncrona, cada microsserviço se conecta ao barramento de eventos¹. Uma vez conectado ao barramento, um microsserviço pode

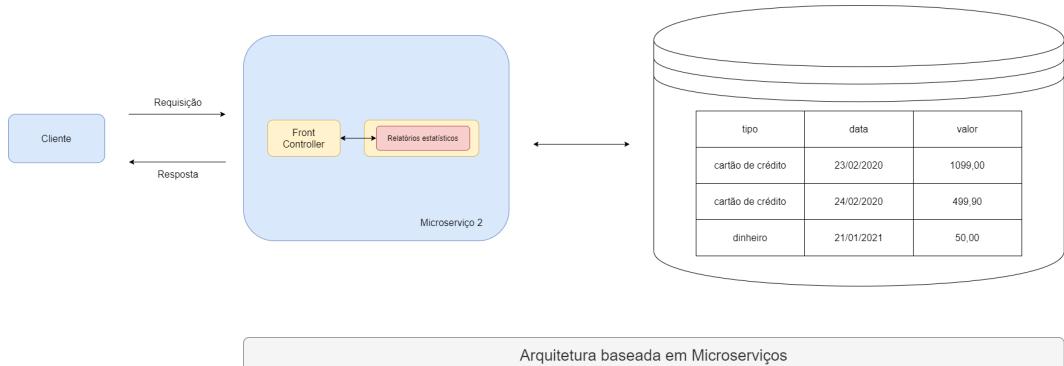
- emitir eventos, o que significa que eles são enviados ao barramento.
- receber eventos do barramento que são gerados por outros microsserviços.

Este modelo perde em **simplicidade** quando comparado ao modelo de comunicação síncrona. Há também a **dependência entre serviços**. Pelo fato de a comunicação ser assíncrona, o microsserviço de origem pode se ocupar com **outras tarefas enquanto aguarda uma eventual resposta do barramento de eventos**.

4.2.5 Comunicação assíncrona - base de dados em função das demais
Uma outra forma de comunicação assíncrona se baseia na construção de uma base de dados em função de outras. Veja a Figura 4.2.5. Ela pode ser uma espécie de **view** contendo somente as partes de interesse.

¹Note que o barramento de eventos poderia ser um ponto de falha centralizado do sistema. Ele é, idealmente, implantado em uma plataforma que lida com esse tipo de problema automaticamente.

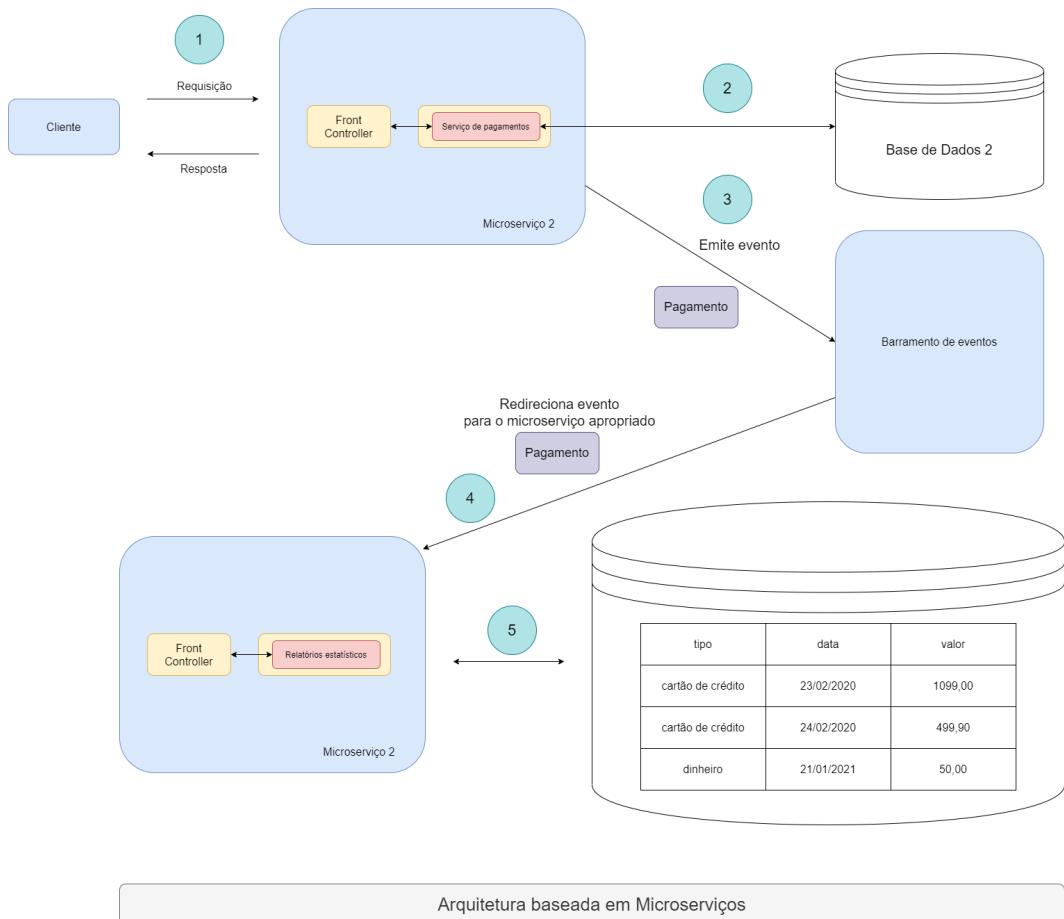
Figura 4.2.5



Arquitetura baseada em Microserviços

A questão óbvia a ser respondida é como essa base pode ser criada sem que um microsserviço accesse diretamente a base de dados de outro e sem estabelecer dependências diretas entre eles. A ideia geral é ilustrada na Figura 4.2.6

Figura 4.2.6



Arquitetura baseada em Microserviços

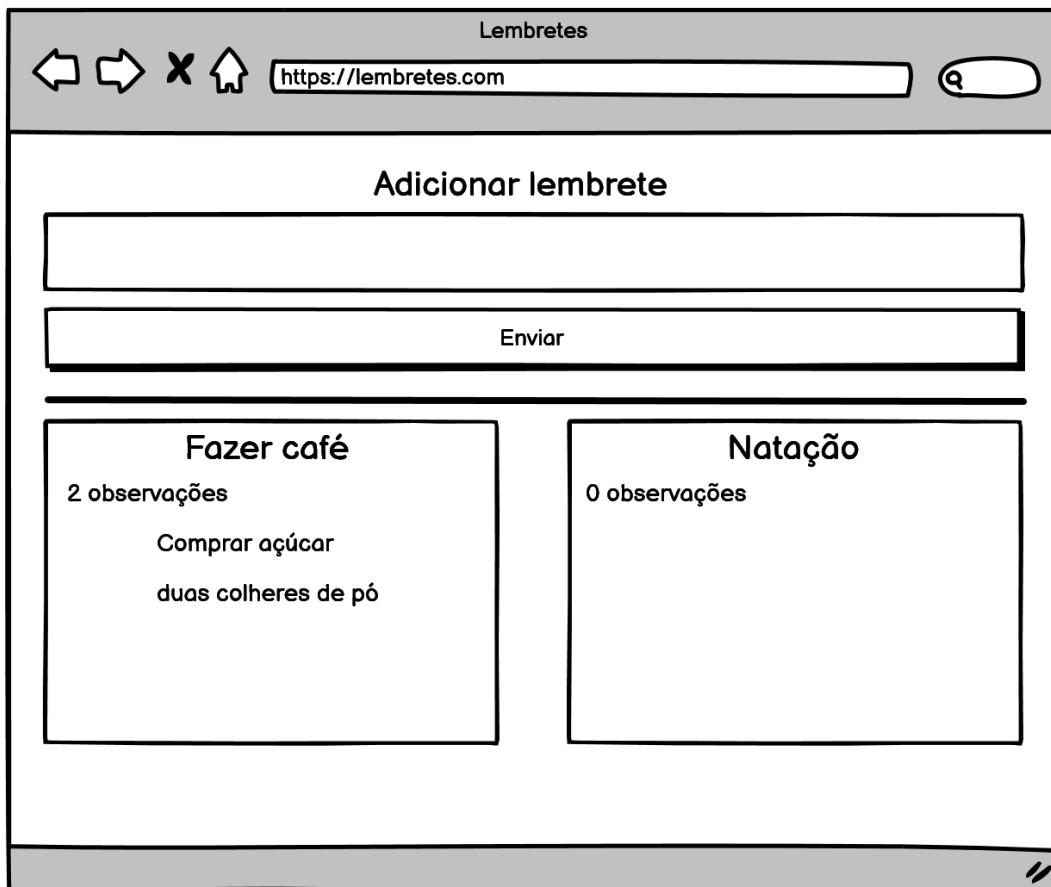
Quando o microsserviço de Pagamentos recebe uma nova requisição, ele armazena os dados de um novo pagamento em sua base, garantindo o seu funcionamento.

Além disso, ele emite um evento que é direcionado a um barramento de eventos. O barramento de eventos se encarrega de fazer uma espécie de envio **broadcast** para que microsserviços interessados naquele evento sejam notificados. Neste exemplo, o microsserviço de Relatórios Estatísticos recebe o evento e atualiza a sua base. Note que a requisição é assíncrona e, **caso o microsserviço de Relatórios Estatísticos esteja temporariamente indisponível, o microsserviço de Pagamentos não deixa de funcionar.** E o oposto também é verdadeiro: **caso o microsserviço de Pagamentos esteja indisponível, o microsserviço de Relatórios Estatísticos pode operar utilizando a base de dados que possuir no momento.**

4.3 Construindo uma aplicação do zero com microsserviços Nesta seção, iremos implementar uma aplicação utilizando a Arquitetura baseada em microsserviços. A ideia é implementar os diversos recursos necessários um a um, ao invés de utilizar pacotes já prontos para isso. Nas próximas seções, lidaremos também com o seu uso.

4.3.1 Visão geral A Figura 4.3.1 mostra a tela principal da aplicação. Ela permite que o usuário armazene os seus **lembretes** e, eventualmente, adicione um número arbitrário de **observações** a eles.

Figura 4.3.1



4.3.2 Quais microsserviços implementar? Uma decisão a ser tomada diz respeito a quais e quantos microsserviços implementar. Para esta aplicação, iremos implementar **um microsserviço para cada tipo de objeto que ela manipula**. Assim, teremos um microsserviço para os lembretes e outro para as observações. Eles realizarão as seguintes tarefas.

- microsserviço de lembretes
 1. criar um lembrete
 2. listar os lembretes
- microsserviço de observações
 1. criar uma observação
 2. listar as observações de um lembrete

É importante observar que, mesmo para essas funcionalidades aparentemente simples, já existem questões relativamente complexas a serem resolvidas. As funcionalidades do microsserviço de observações dependem de dados de lembretes.

Assim, teremos de decidir qual forma de comunicação será empregada entre os microserviços.

4.3.3 Implementando o microserviço de lembretes Os microserviços serão implementados utilizando-se o **nodejs** [2]. Caso não tenha instalado, visite o Capítulo 5 para mais informações.

4.3.4 Workspace Comece criando um diretório para abrigar os arquivos dos projetos.

4.3.5 O microserviço de lembretes Cada microserviço será implementado como um projeto **nodejs** independente. Crie uma pasta chamada **lembretes** em seu workspace e use

npm init -y

para criar um projeto. A opção **-y** indica que deseja-se utilizar valores padrão para cada item que caracteriza o projeto, como nome, versão etc.

4.3.6 Pacotes Os pacotes que utilizaremos, a princípio, são

- **Express** [5] - um framework web para o **nodejs** que adiciona níveis de abstração para, entre outras coisas, a manipulação de requisições *HTTP*.
- **cors** - **CORS** significa **Cross-Origin Resource Sharing**. Trata-se de um mecanismo utilizado para especificar como cliente e servidor podem compartilhar recursos, em particular para o caso em que tiverem domínios diferentes. O pacote cors disponibiliza uma API que simplifica esse tipo de especificação.
- **axios** - um pacote que simplifica a realização de requisições *HTTP* assíncronas usando Ajax.
- **nodemon** - seu nome vem de **Node Monitor**. É natural a necessidade de reiniciar o servidor em tempo de desenvolvimento. Em geral, isso é necessário para que novas atualizações realizadas possam ser testadas. O nodemon é um pacote que monitora a execução de um servidor **nodejs** e que o reinicia automaticamente quando detecta que arquivos de extensões especificadas são alterados.

Eles podem ser instalados com

npm install express cors axios nodemon

Certifique-se de executar esse comando utilizando um terminal vinculado ao diretório em que se encontra o seu projeto  .

4.3.7 O microsserviço de observações Repita os passos para criar o projeto referente às observações. Crie uma pasta chamada **observacoes** em seu workspace - cuidado para não criá-la dentro do diretório do microsserviço de lembretes - e execute

npm init -y

e

npm install express cors axios nodemon

logo a seguir.

4.3.8 Requisições e métodos HTTP do microsserviço de lembretes
Lembre-se que o protocolo HTTP, cuja RFC principal pode ser encontrada no Link 4.3.1, possui métodos com finalidades específicas.

Link 4.3.1

<https://tools.ietf.org/html/rfc2616>

Utilizaremos as seguintes especificações.

- Método HTTP: **PUT**. Padrão de acesso: **/lembretes**. Corpo: **{texto: string}**. Atividade: **Criar um lembrete**.
- Método HTTP: **GET**. Padrão de acesso: **/lembretes**. Corpo: **vazio**. Atividade: **Obter a lista de lembretes**.

4.3.9 Código inicial do microsserviço de lembretes Comece criando um arquivo chamado **index.js** na pasta **lembretes**. O Bloco de Código 4.3.1 mostra a implementação inicial do servidor com as duas rotas propostas.

Bloco de Código 4.3.1

```
1 const express = require ('express');
2 const app = express();
3 app.get ('/lembretes', (req, res) => {
4
5 });
6 app.put ('/lembretes', (req, res) => {
7
8 });
9
10 app.listen(4000, () => {
11   console.log('Lembretes. Porta 4000');
12});
```

4.3.10 Base inicialmente volátil para o microsserviço de lembretes Inicialmente não nos preocuparemos com a implementação de uma base de dados propriamente dita. Os dados serão todas armazenados em uma coleção em memória volátil. Faça a sua definição como mostra o Bloco de Código 4.3.2.

Bloco de Código 4.3.2

```
1 const express = require ('express');
2 const app = express();
3 const lembretes = {};
4 app.get ('/lembretes', (req, res) => {
5
6 });
7 app.put ('/lembretes', (req, res) => {
8
9 });
10 app.listen(4000, () => {
11   console.log('Lembretes. Porta 4000');
12});
```

4.3.11 Requisição GET: Devolvendo a coleção de lembretes A implementação do método GET é muito simples: basta devolver a coleção inteira de lembretes. Veja a sua implementação no Bloco de Código 4.3.3.

Bloco de Código 4.3.3

```
1 . . .
2     app.get('/lembretes', (req, res) => {
3         res.send(lembretes);
4     );
5 . . .
```

4.3.12 Requisição PUT: Geração de id e criação de lembrete Quando um lembrete for inserido, ele será associado a um número de identificação para que, no futuro, seja possível associá-lo a suas observações e diferenciá-lo dos demais. A princípio, nosso id será um simples contador. Quando a requisição é recebida, precisamos extraír o campo **texto** para construir o objeto a ser armazenado. Para tal, vamos utilizar o pacote **body-parser**. Ele devolverá um **middleware** que irá adicionar um campo chamado **body** à requisição, simplificando a extração do conteúdo enviado pelo cliente. Veja o Bloco de Código 4.3.4.

Bloco de Código 4.3.4

```
1 . . .
2     const bodyParser = require('body-parser');
3     const app = express();
4     app.use(bodyParser.json());
5     lembretes = {};
6     contador = 0;
7 . . .
8     app.put('/lembretes', (req, res) => {
9         contador++;
10        const { texto } = req.body;
11        lembretes[contador] = {
12            contador, texto
13        }
14        res.status(201).send(lembretes[contador]);
15    );
```

4.3.13 Executando o servidor com nodemon Abra o arquivo *package.json* do projeto e encontre a chave **scripts**. Ajuste-a como no Bloco de Código 4.3.5.

Bloco de Código 4.3.5

```

1   {
2     "name": "lembretes",
3     "version": "1.0.0",
4     "description": "",
5     "main": "index.js",
6     "scripts": {
7       "test": "echo \\"$Error: no test specified\\" && exit 1",
8       "start": "nodemon index.js"
9     },
10    "keywords": [],
11    "author": "",
12    "license": "ISC",
13    "dependencies": {
14      "axios": "^0.21.1",
15      "cors": "^2.8.5",
16      "express": "^4.17.1",
17      "nodemon": "^2.0.7"
18    }
19  }

```

Em um terminal vinculado ao diretório em que se encontra o projeto, use

npm start

para colocar o servidor em execução.

4.3.14 Testes com Postman O **Postman**² é um cliente HTTP. Passaremos a utilizá-lo para testar nossas APIs. Para testar o endpoint de obtenção da lista de lembretes, use os seguintes valores no Postman.

- Método: **GET**.
- Endereço: **localhost:4000/lembretes**

Nota. Ao abrir o Postman, pode ser necessário clicar em **Create New Request** a fim de visualizar a tela exibida pela Figura 4.3.2. Note que ele possui diversos recursos como a criação de coleções de requisições, ambientes e o uso do **Scratch Pad** e de **Workspaces**.

Veja a Figura 4.3.2. O resultado obtido deve ser um objeto JSON vazio, afinal, ainda não fizemos nenhuma inserção.

²<https://www.postman.com/>

Figura 4.3.2

The screenshot shows the Postman interface with a GET request to `localhost:4000/lembretes`. The URL is highlighted with a red box. The response status is `200 OK`.

Para testar o endpoint de inserção de novo lembrete, use os seguintes valores no Postman.

- Método: **PUT**.
- Endereço: **localhost:4000/lembretes**
- Body: `{"texto": "Fazer café"}`. Para especificar esse valor, clique *Body* » *raw* » No menu de seleção (por padrão mostra “Text”) escolha *JSON*

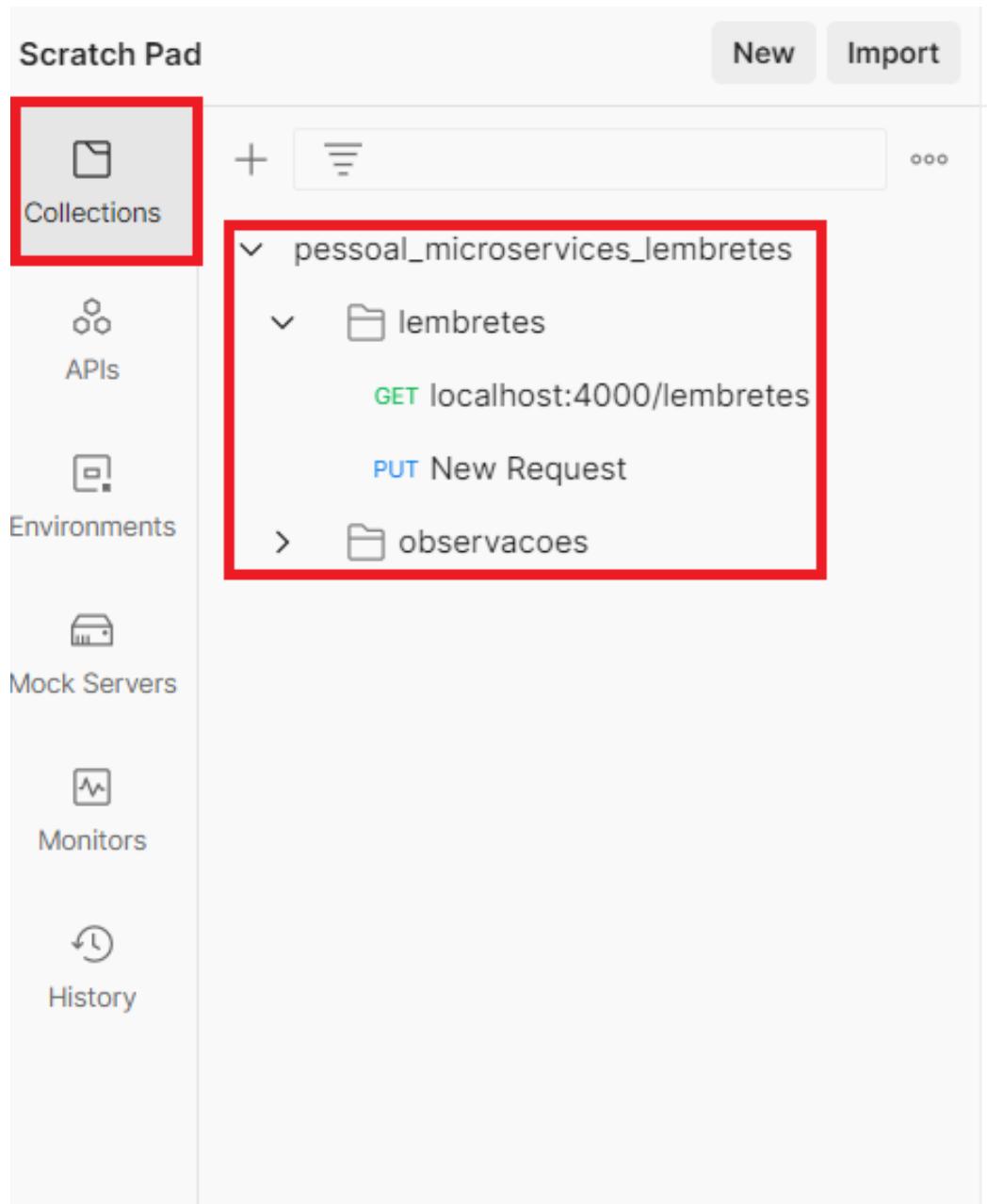
Veja a Figura 4.3.3.

Figura 4.3.3

The screenshot shows the Postman interface with a PUT request to `localhost:4000/lembretes`. The URL is highlighted with a red box. The Body tab is selected, showing raw JSON input: `1 {"...": "Fazer café"}`. The response status is `201 Created`.

4.3.15 Organizando as requisições em uma coleção Pode ser uma boa ideia fazer o uso do recurso de **Coleções de requisições** do Postman. Ele é interessante pois nos permite agrupar requisições relacionadas e mantê-las armazenadas para uso futuro. Para criar uma nova coleção de requisições no Postman, clique *Collections*. A seguir, clique *Create New Collection*. Escolha um nome para a sua coleção, idealmente algo que te ajude a lembrar a que estão associadas as requisições que serão agora agrupadas. A seguir, crie duas pastas: uma para as requisições referentes aos lembretes e outra para as requisições referentes às observações. Para criar as pastas, basta clicar com o direito sobre o nome da coleção. Crie, a seguir, as requisições relacionadas a lembretes em sua respectiva pasta. Veja a Figura 4.3.4.

Figura 4.3.4



4.3.16 Implementando o microsserviço de observações A implementação do microsserviço de observações é semelhante àquela feita para o microsserviço de lembretes. Entretanto, há um ponto importante a ser considerado: cada observação está associada a um lembrete. Note que isso poderá implicar na necessidade da

comunicação entre os microsserviços. Suas especificações são as seguintes.

- Método HTTP: **PUT**. Padrão de acesso: `/lembretes/:id/observacoes`.
Corpo: `{conteudo: string}`. Atividade: **Criar uma nova observação associada ao lembrete cujo id se encontra especificado no padrão de acesso.**
- Método HTTP: **GET**. Padrão de acesso: `/lembretes/:id/observacoes`.
Corpo: **vazio**. Atividade: **Obter a lista de observações associadas ao lembrete cujo id se encontra especificado no padrão de acesso.**

4.3.17 Código inicial do microsserviço de observações Comece criando um arquivo chamado **index.js** na pasta **observacoes**. O código inicial é semelhante àquele visto na criação do microsserviço de lembretes. Note, entretanto, que os padrões de acesso são diferentes. Também é necessário utilizar uma porta diferente. Assim os dois microsserviços poderão ser mantidos em execução simultaneamente. Veja o Bloco de Código 4.3.6.

Bloco de Código 4.3.6

```

1  const express = require ('express');
2  const bodyParser = require('body-parser');
3
4  const app = express();
5  app.use(bodyParser.json());
6
7  //:id é um placeholder
8  //exemplo: /lembretes/123456/observacoes
9  app.put('/lembretes/:id/observacoes', (req, res) => {
10
11 });
12
13 app.get('/lembretes/:id/observacoes', (req, res) => {
14
15 });
16
17 app.listen(5000, (() => {
18     console.log('Lembretes. Porta 5000');
19 }));

```

4.3.18 Base inicialmente volátil para o microsserviço de observações A definição da base de dados que armazena a coleção de observações é exibida no Bloco de Código 4.3.7. Ela é um objeto JSON em que **cada chave é o id de um lembrete e seu valor associado é a coleção de observações associadas àquele lembrete**.

Bloco de Código 4.3.7

```

1   . . .
2   const app = express();
3   app.use(bodyParser.json());
4
5   const observacoesPorLembreteId = {};
6
7   //:id é um placeholder
8   //exemplo: /lembretes/123456/observacoes
9   app.put('/lembretes/:id/observacoes', (req, res) => {
10
11 });
12 . . .

```

4.3.19 Gerando códigos UUID Na implementação da base de dados do serviço de lembretes, utilizamos um simples contador para representar um código único utilizado para diferenciar um lembrete dos demais. Podemos fazer uso de técnicas mais sofisticadas utilizando, por exemplo, o pacote **uuid**. Ele implementa a especificação UUID dada pela **RFC 4122** que pode ser visitada por meio do Link 4.3.2.

Link 4.3.2
<https://tools.ietf.org/html/rfc4122>

Para fazer a sua instalação, use

npm install uuid

Caso deseje, passe a utilizá-lo também no microsserviços de lembretes. Note que, por serem serviços independentes, nada impede que utilizem estratégias diferentes.

4.3.20 Requisição PUT: Inserindo uma nova observação O algoritmo para inserção de nova observação é o seguinte.

- Gerar um novo identificador para a observação a ser inserida.
- Extrair, do corpo da requisição, o texto da observação.
- Verificar se o id de lembrete existente na URL já existe na base e está associado a uma coleção. Em caso positivo, prosseguir utilizando a coleção existente. Caso contrário, criar uma nova coleção.
- Adicionar a nova observação à coleção de observações recém obtida/criada.
- Fazer com que o identificador do lembrete existente na URL esteja associado a essa nova coleção alterada, na base de observações por id de lembrete.
- Devolver uma resposta ao usuário envolvendo o código de status HTTP e algum objeto de interesse, possivelmente a observação inserida ou, ainda, a coleção inteira de observações.

Veja a sua implementação no Bloco de Código 4.3.8.

Bloco de Código 4.3.8

```

1   . . .
2   const { v4: uuidv4 } = require('uuid');
3   . . .
4   //:id é um placeholder
5   //exemplo: /lembretes/123456/observacoes
6   app.put('/lembretes/:id/observacoes', (req, res) => {
7     const idObs = uuidv4();
8     const { texto } = req.body;
9     //req.params dá acesso à lista de parâmetros da URL
10    const observacoesDoLembrete =
11      observacoesPorLembreteId[req.params.id] || [];
12    observacoesDoLembrete.push({ id: idObs, texto });
13    observacoesPorLembreteId[req.params.id] =
14      observacoesDoLembrete;
15    res.status(201).send(observacoesDoLembrete);
16  });

```

4.3.21 Testando inserções de lembretes e observações Para testar, abra o arquivo *package.json* do projeto *observações* e adicione um script que utiliza o *nodemon*, como feito para o projeto de lembretes. Veja o Bloco de Código 4.3.9.

Bloco de Código 4.3.9

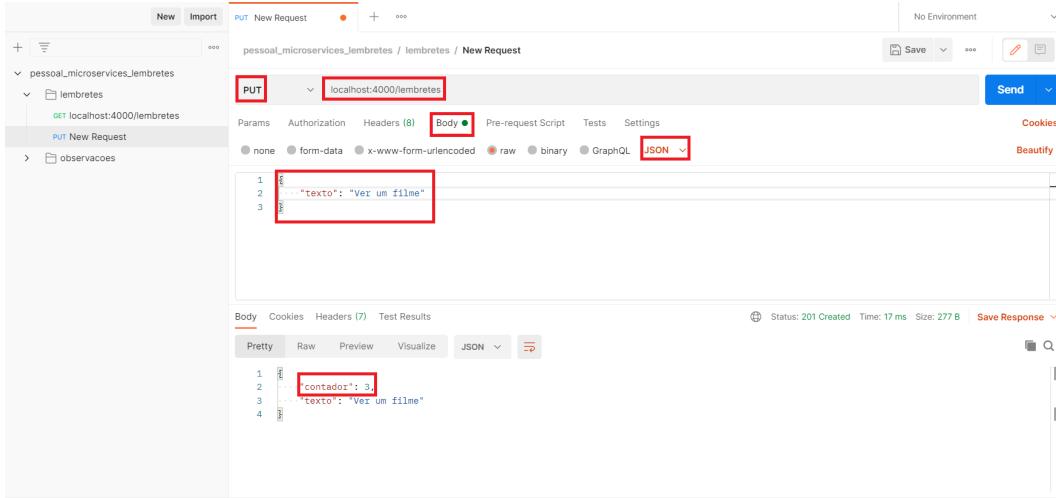
```
1  {
2      "name": "observacoes",
3      "version": "1.0.0",
4      "description": "",
5      "main": "index.js",
6      "scripts": {
7          "test": "echo \"Error: no test specified\" && exit 1",
8          "start": "nodemon index.js"
9      },
10     "keywords": [],
11     "author": "",
12     "license": "ISC",
13     "dependencies": {
14         "axios": "^0.21.1",
15         "cors": "^2.8.5",
16         "express": "^4.17.1",
17         "nodemon": "^2.0.7",
18         "uuid": "^8.3.2"
19     }
20 }
21
```

Utilize

npm start

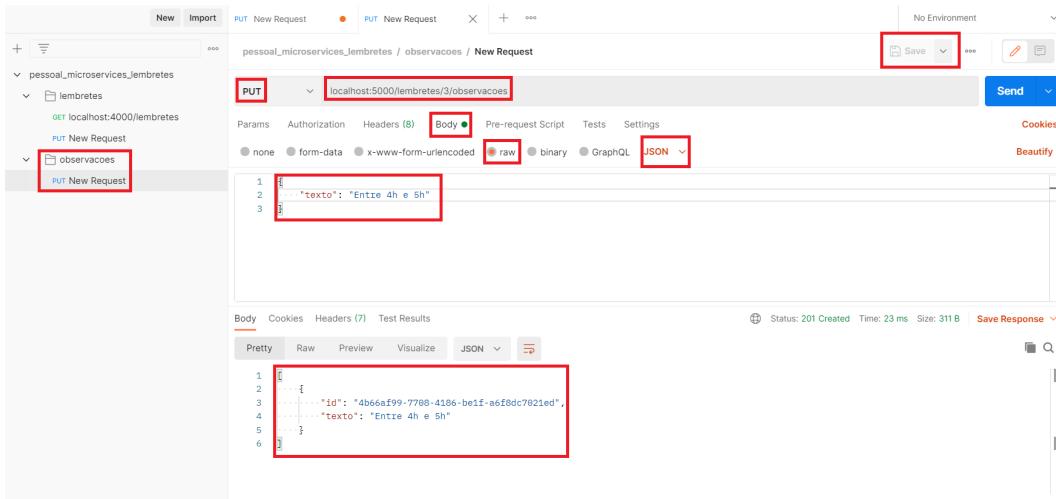
para colocar o microsserviço em execução. A seguir, abra o Postman e envie uma requisição ao microsserviço de lembretes para fazer a inserção de um novo e anotar o seu identificador gerado. Veja a Figura 4.3.5.

Figura 4.3.5



Anote o identificador gerado para o lembrete recém inserido e faça uma requisição de inserção ao microsserviço de observações, como mostra a Figura 4.3.6. Como feito até então, crie a requisição na sub-pasta anteriormente criada para agrupar requisições feitas ao microsserviço de observações. Clique Save para salvar os dados da requisição.

Figura 4.3.6



4.3.22 Requisição GET: Devolvendo a lista de observações de um lembrete A implementação do endpoint de obtenção de observações de um lembrete especificado é um tanto simples. Ela extrai o identificador de lembrete existente na URL e acessa a base de dados. Caso uma coleção seja encontrada na base, ela é devolvida. Caso contrário, o microsserviço devolve uma coleção vazia. Veja o Bloco de Código 4.3.10.

Bloco de Código 4.3.10

```

1 app.get('/lembretes/:id/observacoes', (req, res) => {
2     res.send(observacoesPorLembreteId[req.params.id] || []);
3 });

```

4.3.23 Teste para a obtenção da lista de observações No Postman, crie uma nova requisição para testar a nova implementação, como na Figura 4.3.7.

Figura 4.3.7

The screenshot shows the Postman interface with the following details:

- Request URL:** localhost:5000/lembretes/3/observacoes
- Method:** GET
- Body:** (Empty)
- Headers:** (Empty)
- Query Params:** (Empty)
- Response:**
 - Status: 200 OK
 - Time: 13 ms
 - Size: 377 B
 - Save Response

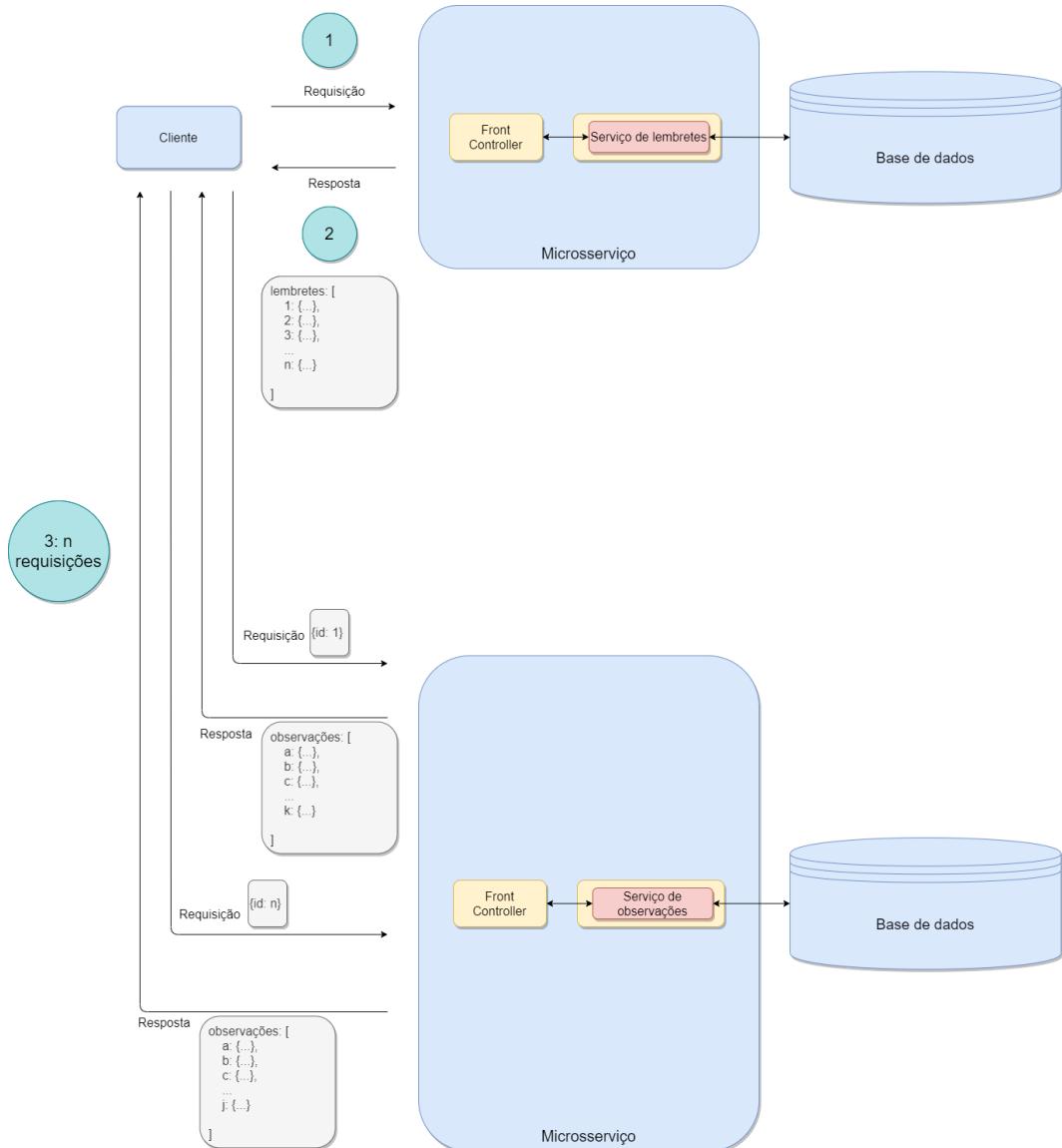
```

1 [
2   {
3     "id": "305b9cbe-a491-424e-a5ad-89e765571417",
4     "texto": "Entre 4h e 6h"
5   },
6   {
7     "id": "0b99bc3b-ea31-40d8-b478-9e56fcc67421",
8     "texto": "Entre 4h e 8h"
9   }
]

```

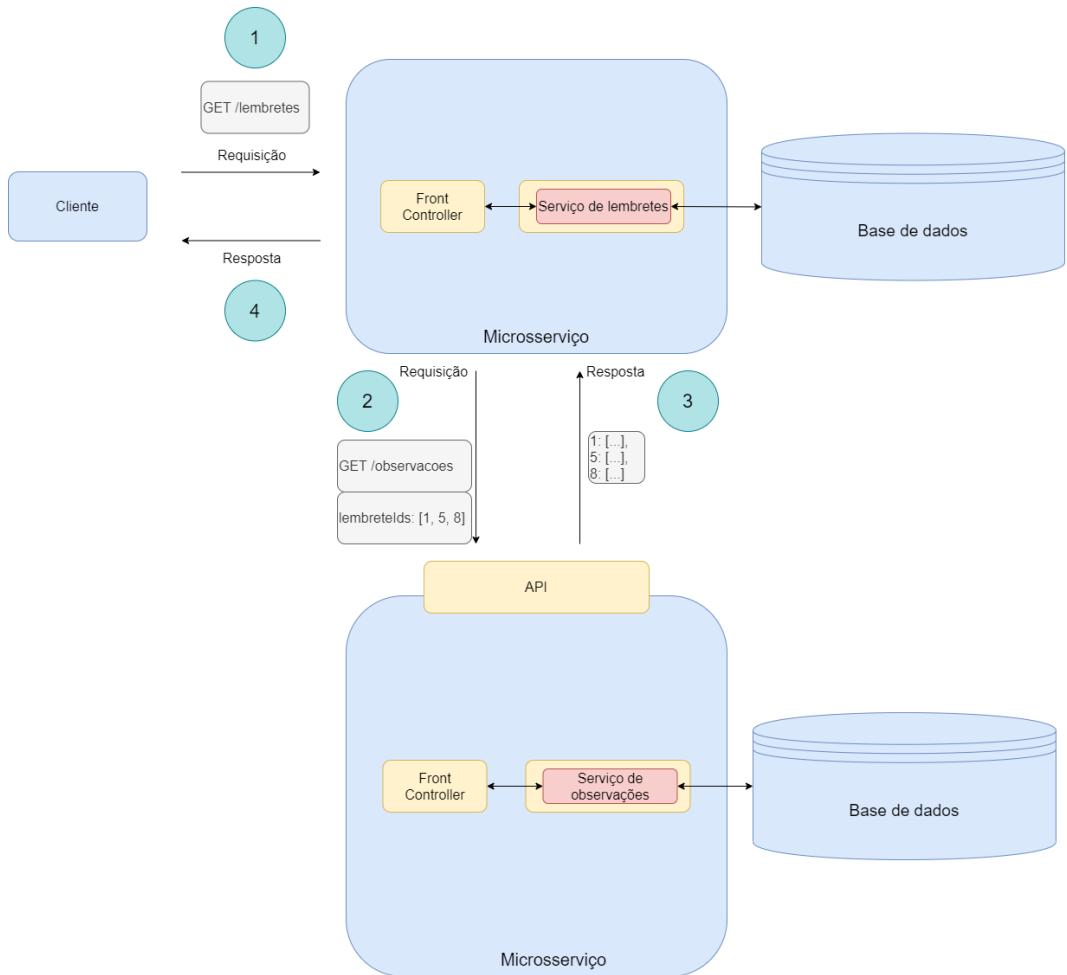
4.3.24 Busca da coleção de lembretes incluindo observações: $n + 1$ requisições feitas pelo cliente Uma aplicação cliente, como aquela exibida pela Figura 4.3.1, pode estar interessada em obter a coleção de lembretes incluindo a coleção de observações referente a cada um deles. Devido à arquitetura que estamos empregando no Back End, podem ser necessárias muitas requisições para atender essa necessidade, como mostra a Figura 4.3.8. Uma primeira requisição é feita para a obtenção da coleção de lembretes. A seguir, n requisições são feitas para a obtenção de cada uma das coleções de observações.

Figura 4.3.8



4.3.25 Busca da coleção de lembretes incluindo observações: comunicação síncrona Pode ser interessante tornar transparente para o cliente esse número de requisições, simplificando a sua implementação. A ideia é aplicar técnicas de comunicação entre microserviços para fazê-lo. Uma dela é a **Comunicação Síncrona**, como ilustra a Figura 4.3.9

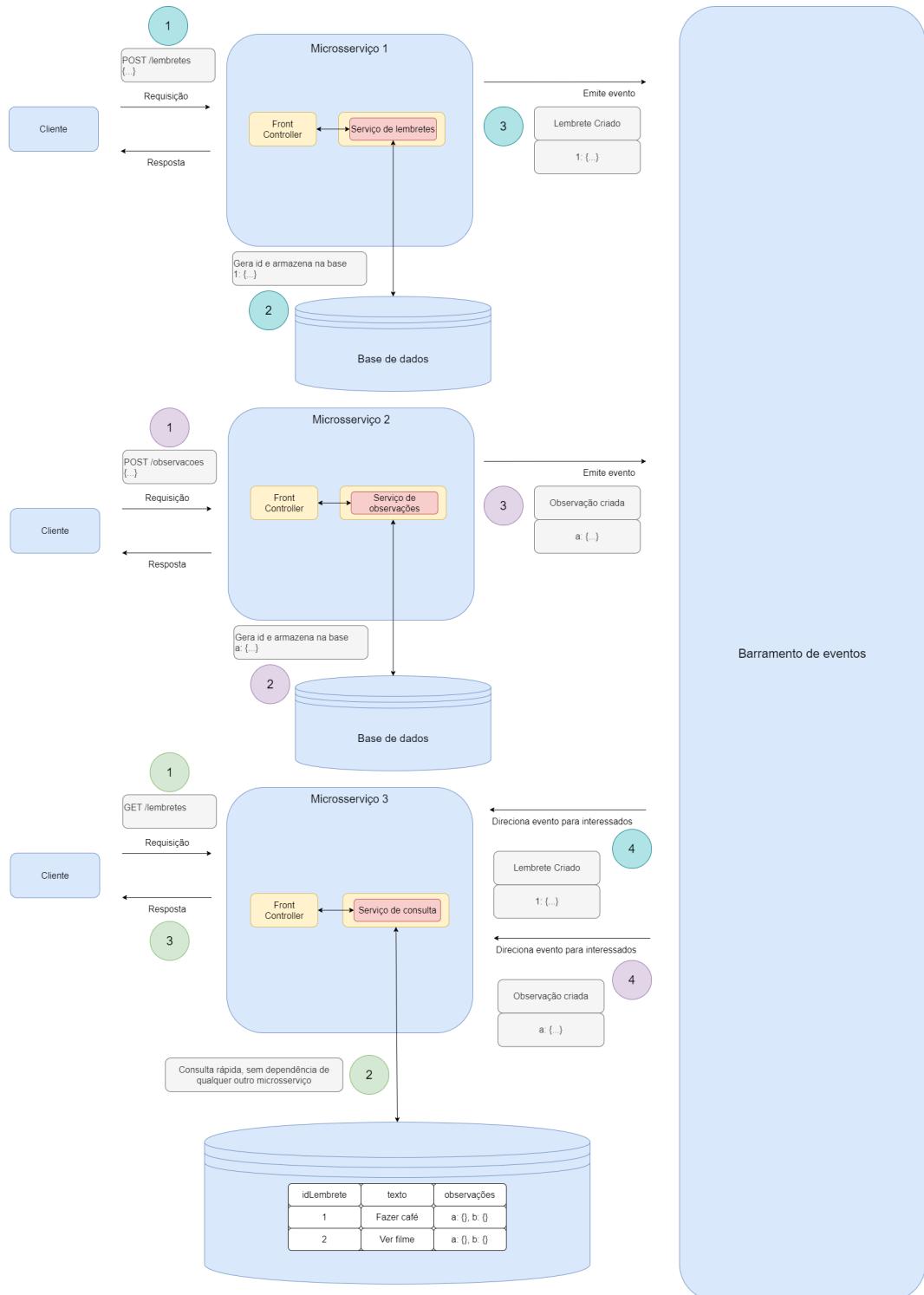
Figura 4.3.9



Como discutimos, esse é um modelo de simples entendimento. Entretanto, ele apresenta dependência entre microserviços e pode dar origem a cadeias de requisições a diferentes microserviços, o que pode comprometer o desempenho do microserviço de origem.

4.3.26 Busca da coleção de lembretes incluindo observações: comunicação assíncrona Outra possibilidade envolve o uso da comunicação assíncrona entre os microserviços. Ela traz vantagens como a independência entre microserviços e a possibilidade de melhorias de desempenho. Há, por outro lado, a necessidade de se cuidar da redundância de dados. O modelo ilustrado na Figura 4.3.10 traz a proposta de uso de um microserviço de consulta.

Figura 4.3.10



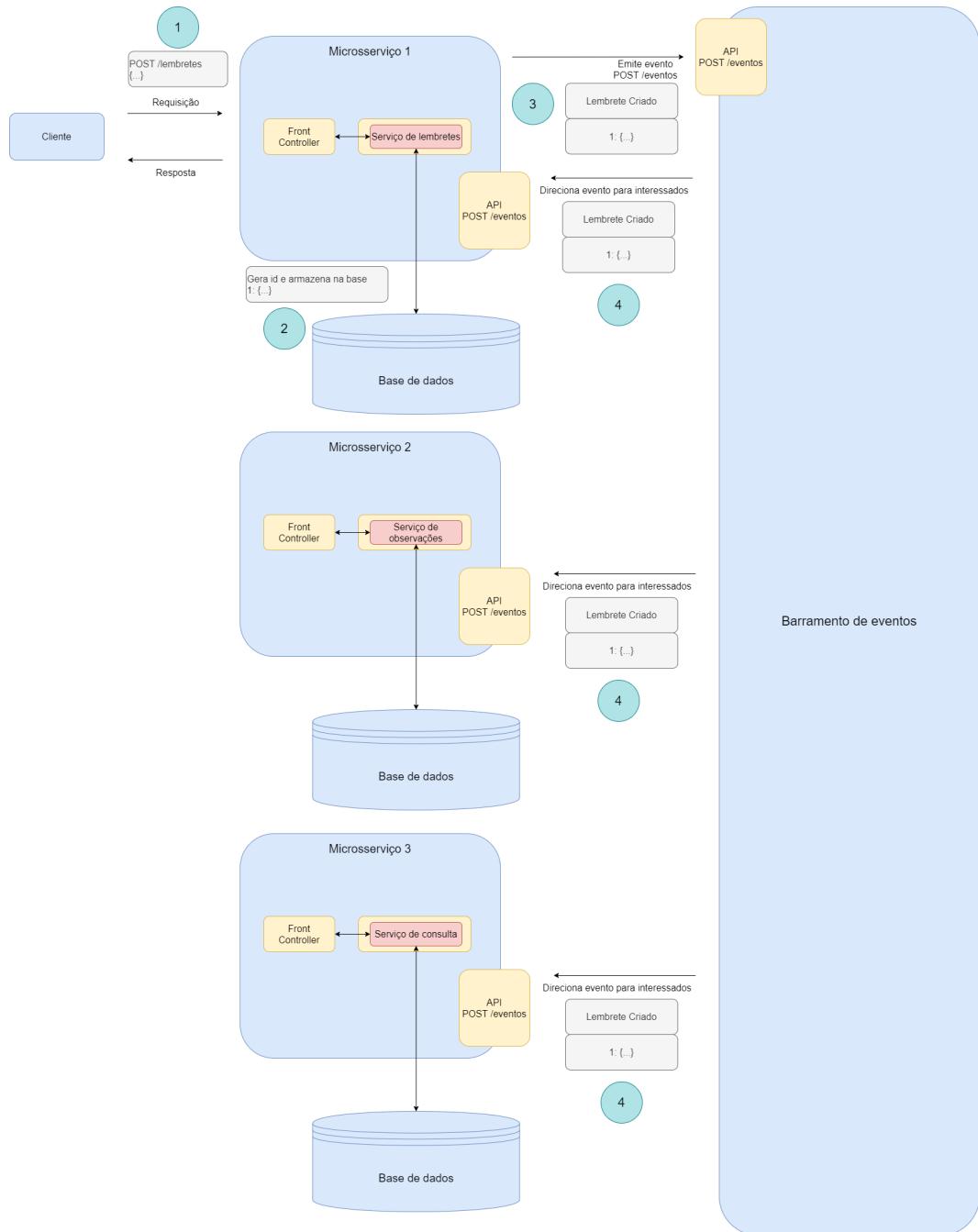
4.3.27 Uma implementação manual de um barramento de eventos Há diferentes implementações de barramentos de eventos - também chamadas de filas

de mensagens e outros - disponíveis. Alguns exemplos são os seguintes.

- [Apache ActiveMQ](#)
- [RabbitMQ](#)
- [Apache Kafka](#)
- [Amazon MQ](#)
- [Google Cloud Pub/Sub](#)

Nesta seção, iremos fazer a nossa própria implementação para fins de aprendizado. Aplicações profissionais certamente empregam soluções como as citadas. Elas são amplamente testadas e utilizadas por milhões de usuários, além de, em geral, serem implantadas em serviços de computação em nuvem de alta disponibilidade, tolerância a falhas etc. A Figura 4.3.11 ilustra o funcionamento básico de nossa implementação. O barramento de eventos - que também é um microsserviço - possui um endpoint que viabiliza a entrega de eventos. Cada serviço interessado em algum tipo de evento também disponibiliza um endpoint assim. Quando o barramento de eventos recebe um evento, ele faz uma espécie de envio **broadcast**.

Figura 4.3.11



4.3.28 Barramento de eventos - criando o projeto O barramento de eventos será um novo microserviço, independente dos demais. Por isso, crie uma nova pasta chamada *barramento-de-eventos* em seu workspace e use

npm init -y

para criar o projeto. Abra um terminal vinculado à nova pasta e instale as dependências com

npm install express nodemon axios cors

Crie um arquivo chamado *index.js* na nova pasta e, como feito com os demais projetos, abra o arquivo *package.json* e especifique um novo script de execução como mostra o Bloco de Código 4.3.11.

Bloco de Código 4.3.11

```
1  {
2      "name": "barramento-de-eventos",
3      "version": "1.0.0",
4      "description": "",
5      "main": "index.js",
6      "scripts": {
7          "test": "echo \\\"Error: no test specified\\\" && exit 1",
8          "start": "nodemon index.js"
9      },
10     "keywords": [],
11     "author": "",
12     "license": "ISC",
13     "dependencies": {
14         "axios": "^0.21.1",
15         "cors": "^2.8.5",
16         "express": "^4.17.1",
17         "nodemon": "^2.0.7"
18     }
19 }
```

O Bloco de Código 4.3.12 mostra a implementação inicial do barramento de eventos. Ele possui um endpoint que se encarrega de direcionar o evento recebido aos demais microsserviços.

Bloco de Código 4.3.12

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  //para enviar eventos para os demais microsserviços
4  const axios = require('axios');

5
6  const app = express();
7  app.use(bodyParser.json());

8
9  app.post('/eventos', (req, res) => {
10    const evento = req.body;
11    //envia o evento para o microsserviço de lembretes
12    axios.post('http://localhost:4000/eventos', evento);
13    //envia o evento para o microsserviço de observações
14    axios.post('http://localhost:5000/eventos', evento);
15    res.status(200).send({ msg: "ok" });
16  });

17
18  app.listen(10000, () => {
19    console.log('Barramento de eventos. Porta 10000.')
20  })
```

Use

npm start

para colocar o microsserviço em execução.

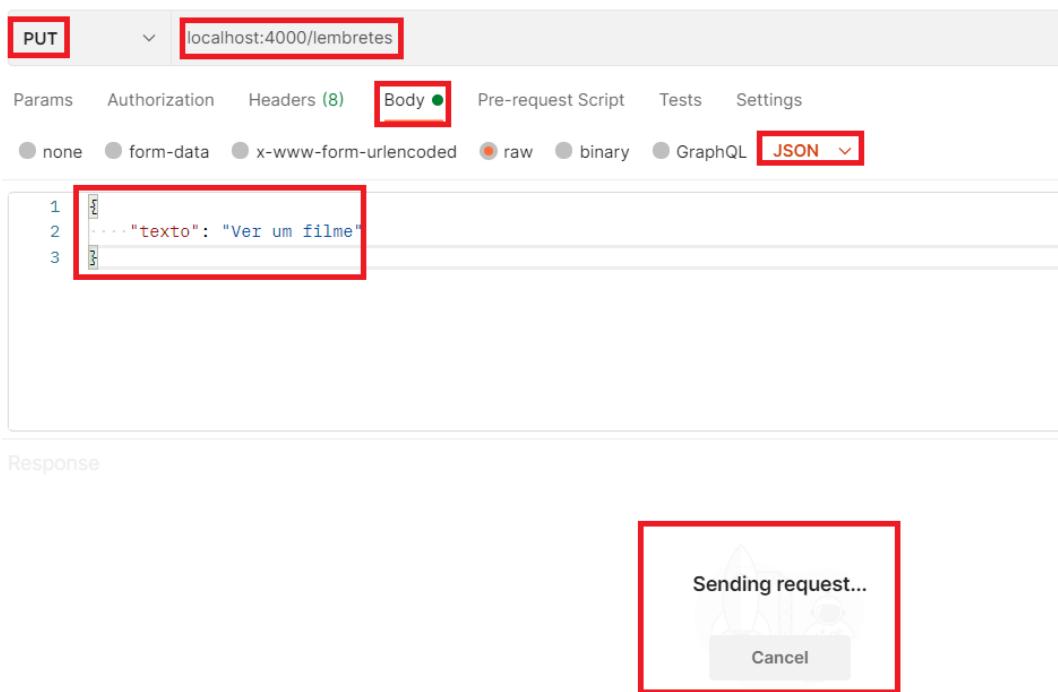
4.3.29 Emissão de eventos a partir do microsserviço de lembretes Sempre que o microsserviço de lembretes faz a inserção de um novo lembrete em sua base própria, ele também emite um evento para que outros microsserviços interessados tenham acesso aos dados. Ele faz isso por meio de uma requisição HTTP do tipo POST enviada ao barramento de eventos. Para fazer essa requisição, utilizaremos o pacote *axios*. Veja o Bloco de Código 4.3.13.

Bloco de Código 4.3.13

```
1 . . .
2 const axios = require("axios");
3 . . .
4 app.put("/lembretes", async (req, res) => {
5     contador++;
6     const { texto } = req.body;
7     lembretes[contador] = {
8         contador,
9         texto,
10    };
11    await axios.post("http://localhost:10000/eventos", {
12        tipo: "LembreteCriado",
13        dados: {
14            contador,
15            texto,
16        },
17    });
18    res.status(201).send(lembretes[contador]);
19 }) ;
```

4.3.30 Testando a emissão de eventos de inserção de lembretes No Postman, faça o teste ilustrado na Figura 4.3.12. Veja que ele ficará bloqueado até um timeout acontecer.

Figura 4.3.12



Isso é esperado já que o barramento de eventos, ao receber a requisição, faz novas requisições a todos os microserviços, tentando repassar o eventos. Como nenhum deles tem o endpoint /eventos implementado, espera-se que a requisição não tenha sucesso. Ajustaremos isto em breve. Neste momento, é de se esperar que o terminal em que o barramento de eventos está em execução exiba uma mensagem parecida com aquele exibida na Figura 4.3.13.

Figura 4.3.13

```
Barramento de eventos, Porta 10000.
(node:13336) UnhandledPromiseRejectionWarning: Error: Request failed with status code 404
    at createError (C:\Users\rodri\Documents\workspaces\pessoal\node_ud_based_microservices\barramento-de-eventos\node_modules\axios\lib\core\createError.js:16:15)
    at settle (C:\Users\rodri\Documents\workspaces\pessoal\node_ud_based_microservices\barramento-de-eventos\node_modules\axios\lib\core\settle.js:17:12)
    at IncomingMessage.handleStreamEnd (C:\Users\rodri\Documents\workspaces\pessoal\node_ud_based_microservices\barramento-de-eventos\node_modules\axios\lib\adapters\http.js:260:11)
    at IncomingMessage.emit (events.js:327:22)
```

4.3.31 Emissão de eventos a partir do microserviço de observações De maneira análoga, faremos com que o microserviço de observações emita um evento uma vez que uma nova observação seja criada. Ele também faz isso por meio de uma simples requisição HTTP POST enviada ao barramento de eventos. Veja o Bloco de Código 4.3.14.

Bloco de Código 4.3.14

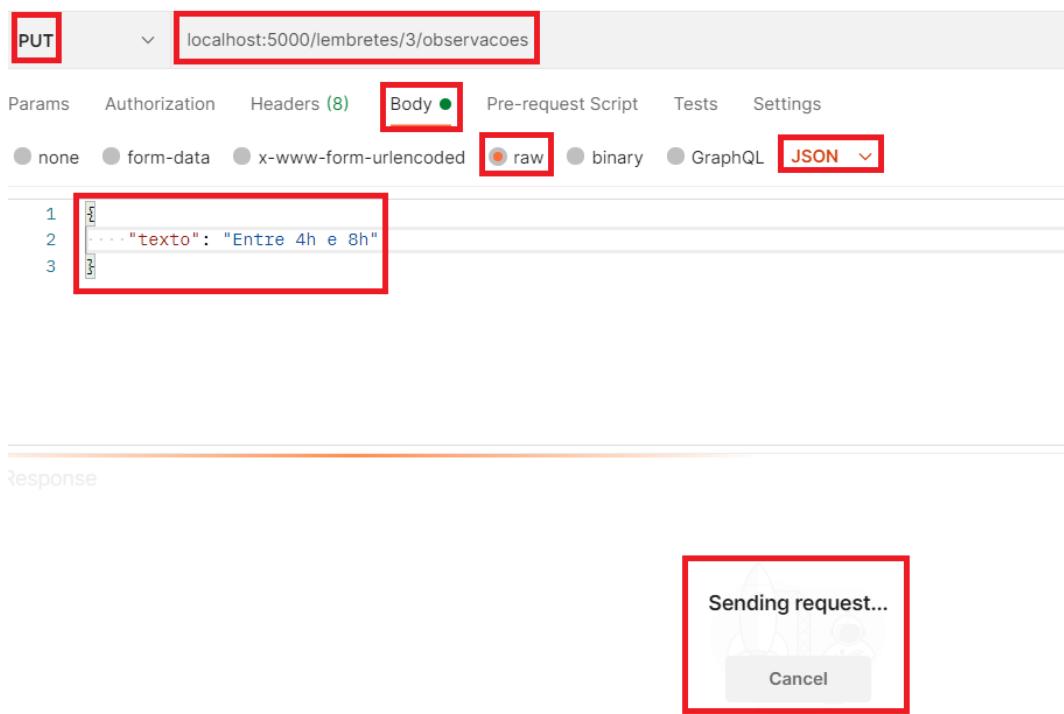
```

1   . . .
2   const axios = require ('axios');
3   . . .
4   app.put('/lembretes/:id/observacoes', async (req, res) => {
5       const idObs = uuidv4();
6       const { texto } = req.body;
7       //req.params dá acesso à lista de parâmetros da URL
8       const observacoesDoLembrete =
9           observacoesPorLembreteId[req.params.id] || [];
10      observacoesDoLembrete.push({ id: idObs, texto });
11      observacoesPorLembreteId[req.params.id]
12          = observacoesDoLembrete;
13      await axios.post('http://localhost:10000/eventos', {
14          tipo: "ObservacaoCriada",
15          dados: {
16              id: idObs, texto, lembreteId: req.params.id
17          }
18      })
19      res.status(201).send(observacoesDoLembrete);
20  });

```

4.3.32 Testando a emissão de eventos de inserção de observações No Postman, execute também o teste ilustrado na Figura 4.3.14. Embora a requisição seja enviada adequadamente, o barramento de eventos tentará enviar uma requisição no endpoint /eventos dos demais microsserviços e, como ele ainda não existe, o mesmo erro acontecerá.

Figura 4.3.14



No terminal executando o barramento de eventos, o erro exibido deverá ser parecido com aquele exibido na Figura 4.3.13.

4.3.33 Recebendo eventos nos microsserviços de lembretes e de observações Para cada microsserviço, a recepção de eventos é feita no endpoint `/eventos` usando o método POST. Sua implementação inicial é **idêntica para ambos os microsserviços**. Faça a implementação em ambos como mostra o Bloco de Código 4.3.15.

Bloco de Código 4.3.15

```

1 //adicionar a ambos microsservicos de lembretes e
  observacoes
2 app.post("/eventos", (req, res) => {
3   console.log(req.body);
4   res.status(200).send({ msg: "ok" });
5 });

```

4.3.34 Testes de inserção de lembretes e observações após implementação dos endpoints Faça a inserção de um lembrete e, logo depois, a inserção de uma observação associada a ele, usando o Postman. O terminal executando cada um

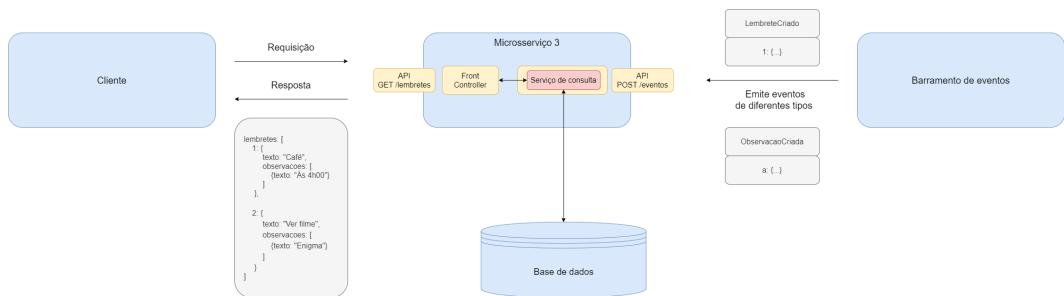
dos microsserviços (de lembretes e de observações) deverá exibir algo parecido com o que mostra a Figura 4.3.15.

Figura 4.3.15

```
{
  tipo: 'LembreteCriado',
  dados: { contador: 3, texto: 'Ver um filme' }
}
{
  tipo: 'ObservacaoCriada',
  dados: {
    id: 'd972b016-4286-495b-9feb-30ed9d2229fb',
    texto: 'Entre 4h e 8h',
    lembreteId: '3'
  }
}
```

4.3.35 O microsserviço de consulta O microsserviço de consultas permite que clientes interessados obtenham uma cópia da coleção completa de lembretes, incluindo a lista de observações que cada um deles eventualmente possui. Para construir essa base de dados, ele recebe eventos do barramento de eventos. Veja a Figura 4.3.16.

Figura 4.3.16



Para iniciar a sua implementação, crie uma nova pasta chamada *consulta*. A seguir, use

npm init -y

para criar o projeto. As dependências podem ser instaladas como a seguir. O serviço de consulta não irá fazer requisições, ele apenas as recebe. Assim, não

precisa do pacote axios.

npm install express cors nodemon

Crie um arquivo chamado *index.js* na raiz da pasta *consulta*. Seu conteúdo inicial é dado no Bloco de Código 4.3.16.

Bloco de Código 4.3.16

```
1  const express = require("express");
2  const app = express();
3  app.use(express.json());
4
5  app.get("/lembretes", (req, res) => {});
6
7  app.post("/eventos", (req, res) => {});
8
9  app.listen(6000, () => console.log("Consultas. Porta 6000"));
```

Abra o arquivo *package.json* do microsserviço de consulta e adicione o script destacado no Bloco de Código 4.3.17.

Bloco de Código 4.3.17

```
1  {
2      "name": "consulta",
3      "version": "1.0.0",
4      "description": "",
5      "main": "index.js",
6      "dependencies": {
7          "cors": "^2.8.5",
8          "express": "^4.17.1",
9          "nodemon": "^2.0.7"
10     },
11     "devDependencies": {},
12     "scripts": {
13         "test": "echo \\\"Error: no test specified\\\" && exit 1",
14         "start": "nodemon index.js"
15     },
16     "keywords": [],
17     "author": "",
18     "license": "ISC"
19 }
```

4.3.36 A base de dados do microsserviço de consulta A base de dados do microsserviço de consulta será um simples objeto JSON. Cada chave é o identificador de um lembrete que fica associada a seu lembrete. Além disso, cada lembrete tem uma chave chamada **observacoes** associada a uma coleção em que cada objeto JSON é uma observação daquele lembrete. Veja a Figura 4.3.17.

Figura 4.3.17

```
{
  "1": {
    "contador": 1,
    "texto": "Ver um filme",
    "observacoes": [
      {
        "id": "0aa13eb4-d988-4160-8a53-4f1d746a5f94",
        "texto": "Entre 4h e 8h",
        "lembreteId": "1"
      },
      {
        "id": "2d4635aa-6a73-4159-82b9-42ac5a4285ab",
        "texto": "Entre 4h e 9h também",
        "lembreteId": "1"
      }
    ],
    "2": {
      "contador": 2,
      "texto": "Ver outro filme",
      "observacoes": [
        ...
      ]
    }
  }
}
```

Elas também serão armazenadas em memória volátil, para que possamos dar foco ao estudo da arquitetura baseada em microsserviços. Assim, faça a sua construção como no Bloco de Código 4.3.18.

Bloco de Código 4.3.18

```

1  const express = require("express");
2  const app = express();
3  app.use(express.json());
4
5  const baseConsulta = {};
6  . . .

```

4.3.37 Atualização da base do microsserviço de consulta O microsserviço de consulta atualiza a base de acordo com o tipo do evento que recebe.

- **EventoCriado** - Acessa a base usando o identificador do lembrete e associa a ele o objeto existente no campo **dados** do evento.
- **ObservacaoCriada** - Acessa a base usando o identificador do lembrete a que a observação criada está associada. A seguir, acessa o campo **observacoes** do lembrete - criando uma lista vazia caso ainda não exista - e adiciona o objeto existente no campos **dados** do evento à coleção.

As funções de tratamento de evento serão valores em um mapa - um objeto JSON - em que as chaves são os tipos dos respectivos eventos. Veja o Bloco de Código 4.3.19.

Bloco de Código 4.3.19

```
1   . . .
2   const baseConsulta = {};
3
4   const funcoes = {
5     LembreteCriado: (lembrete) => {
6       baseConsulta[lembrete.contador] = lembrete;
7     },
8     ObservacaoCriada: (observacao) => {
9       const observacoes =
10         baseConsulta[observacao.lembreteId]["observacoes"] ||
11         [];
12       observacoes.push(observacao);
13       baseConsulta[observacao.lembreteId]["observacoes"] =
14         observacoes;
15     }
16   };
17   . . .
```

O endpoint para o qual eventos são enviados é implementado em função desse mapa: a função associada ao tipo do evento recebido é executada. Ela é alimentada com o objeto associado ao campo **dados** do evento. O endpoint de consulta simplesmente devolve a base inteira. Veja o Bloco de Código 4.3.20.

Bloco de Código 4.3.20

```
1 const funcoes = { . . . };
2
3 app.get("/lembretes", (req, res) => {
4     res.status(200).send(baseConsulta);
5 });
6
7 app.post("/eventos", (req, res) => {
8     funcoes[req.body.tipo](req.body.dados);
9     res.status(200).send(baseConsulta);
10 });
11 . . .
```

4.3.38 Barramento de eventos direcionando eventos ao microsserviço de consulta Lembre-se de atualizar o barramento de eventos para que ele emita eventos direcionados ao microsserviço de consulta, como mostra o Bloco de Código 4.3.21.

Bloco de Código 4.3.21

```
1 . . .
2 app.post("/eventos", (req, res) => {
3     const evento = req.body;
4     //envia o evento para o microsserviço de lembretes
5     axios.post("http://localhost:4000/eventos", evento);
6     //envia o evento para o microsserviço de observações
7     axios.post("http://localhost:5000/eventos", evento);
8     //envia o evento para o microsserviço de consulta
9     axios.post("http://localhost:6000/eventos", evento);
10    res.status(200).send({ msg: "ok" });
11 });
12 . . .
```

4.3.39 Testando o microsserviço de consulta A fim de testar o microsserviço de consulta, crie uma nova pasta na coleção existente no Postman. A seguir crie uma nova requisição como ilustra a Figura 4.3.18.

Figura 4.3.18



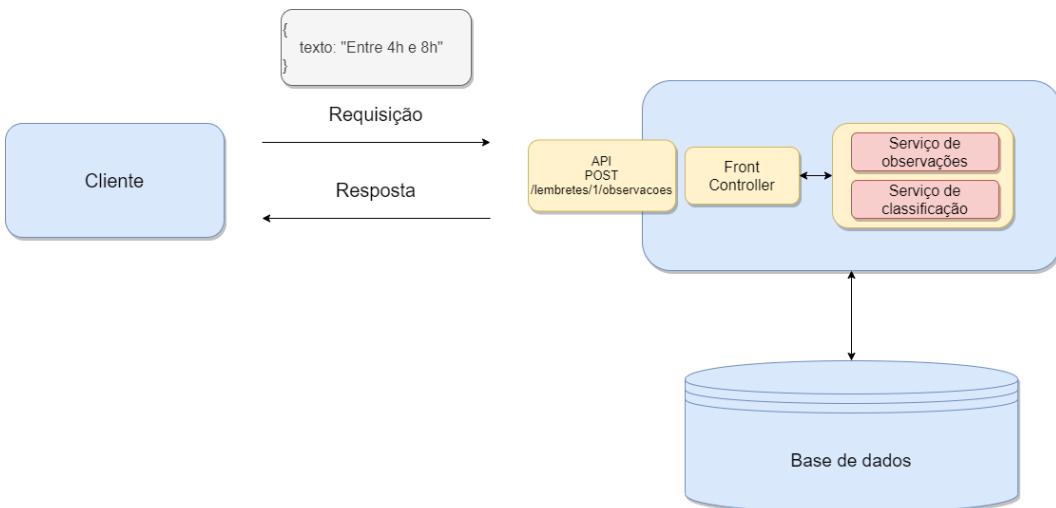
Para realizar os testes, faça o seguinte.

- Crie dois lembretes.
- Crie duas observações associadas a cada um dos lembretes.
- Encerre a execução de ambos os microsserviços de lembretes e observações.
- Faça uma nova requisição ao microsserviço de consulta.

4.3.40 Classificação de observações Suponha que desejamos classificar algumas observações realizadas, indicando que elas são “importantes”. Para que uma observação seja considerada importante, basta que ela contenha a palavra “importante”. Trata-se de uma funcionalidade extremamente simples e podemos empregar diversas técnicas para implementá-la.

4.3.40.1 Implementação no próprio microsserviço de observações. Neste cenário, o microsserviço de observações se encarregaria de analisar o conteúdo de cada observação, como ilustra a Figura 4.3.19.

Figura 4.3.19

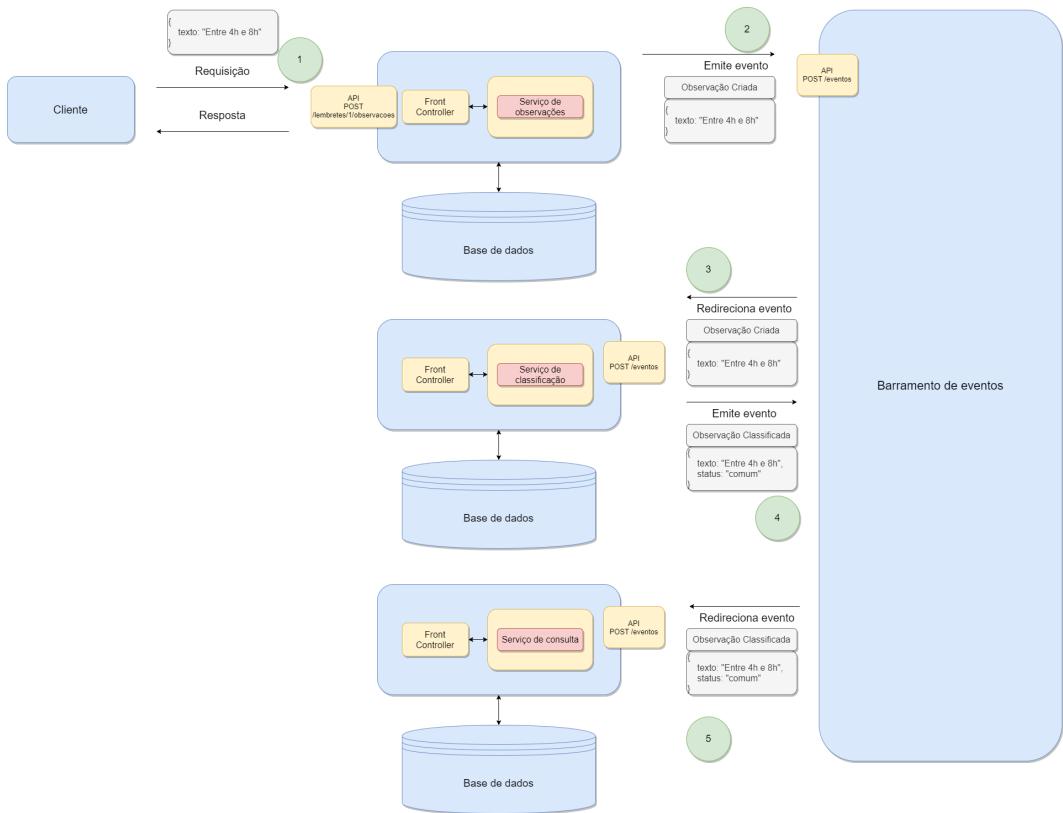


Embora essa implementação seja simples, ela tem algumas características que podem ser indesejáveis.

- A razão de ser do microsserviço de observações é fazer manipulações de acesso a dados envolvendo observações. A classificação tem uma regra um tanto específica que, inclusive, pode mudar com o tempo e se tornar mais complexa. Entregar essa nova responsabilidade ao microsserviço de observações pode comprometer a sua **alta coesão**, o que traz como consequência níveis mais baixos de manutenibilidade e reusabilidade.
- A sua implementação feita no microsserviço de observações pode comprometer uma das principais vantagens obtidas quando a arquitetura baseada em microsserviços é empregada: a possibilidade de **equipes diferentes** serem responsáveis por microsserviços diferentes e utilizarem tecnologias completamente diferentes em suas implementações.
- A classificação de uma observação não necessariamente é imediata. Ela não necessariamente ocorre assim que uma observação é inserida no sistema. É claro que estamos lidando com um exemplo didático, simples. No entanto, a classificação poderia ser um processo demorado que poderia, inclusive, requerer intervenção manual. Assim, em um determinado instante, pode ser de interesse manter a funcionalidade de classificação de observações em funcionamento mesmo se **eventualmente o microsserviço de observações estiver indisponível**.

4.3.40.2 Microsserviço próprio para a classificação Implementar a funcionalidade de classificação como um novo microsserviço parece natural. Uma possibilidade é ilustrada na Figura 4.3.20.

Figura 4.3.20

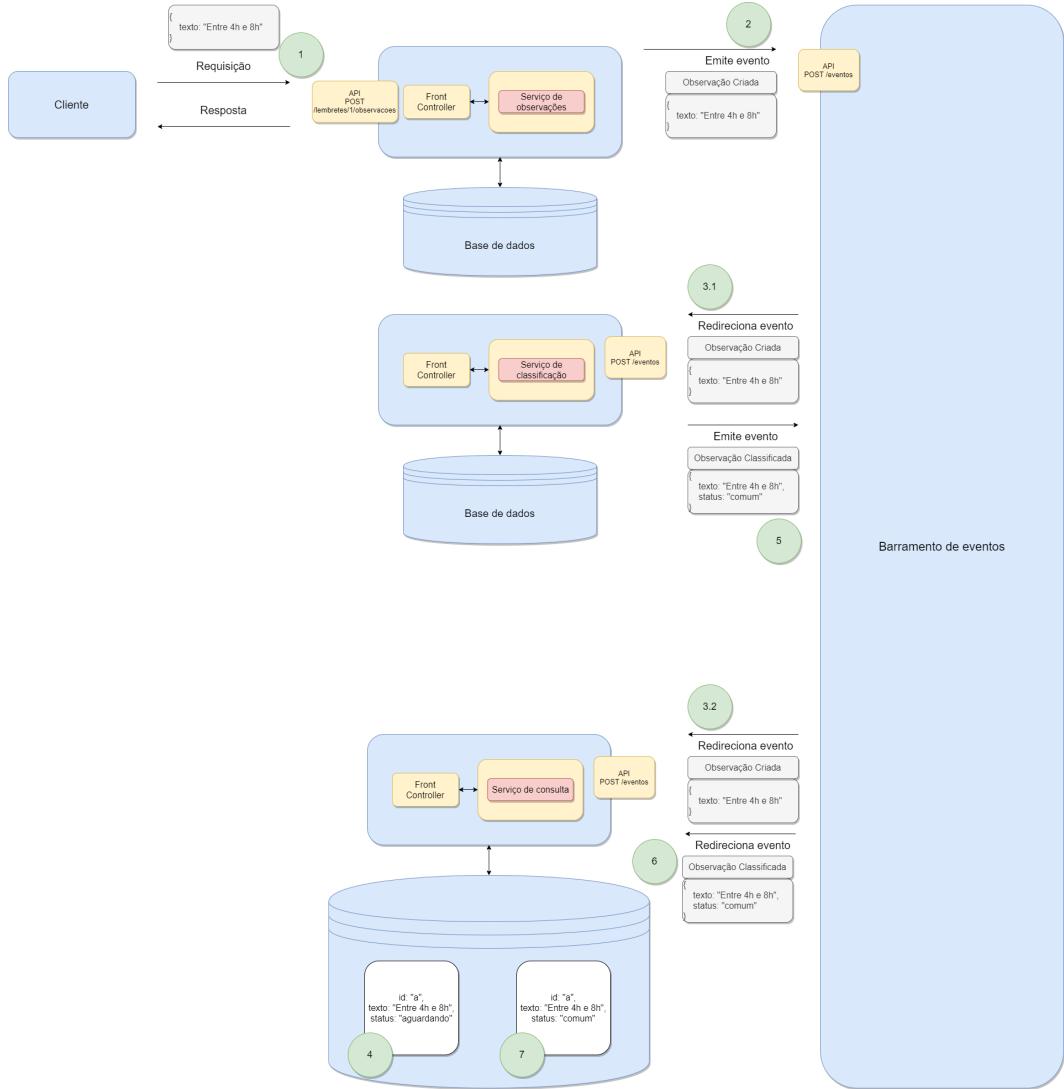


Essa alternativa ilustra uso apropriado da arquitetura baseada em microsserviços. Entretanto, dependendo da natureza da aplicação, ela pode trazer consequências também indesejáveis.

- Um usuário que faz a inserção de uma nova observação espera ser capaz de vê-la no sistema (na interface gráfica que está utilizando para acessar as funcionalidades) imediatamente. Ela precisa, pelo menos, ser apresentada com o status **sendo avaliada** ou algo parecido. Se o microsserviço de classificação demorar para realizar o seu trabalho, a base de dados do microsserviço de consulta ficará desatualizada e ela não poderá ser visualizada.

4.3.40.3 Microsserviço próprio para a classificação - atualização feita pelo microsserviço de consulta Outra possibilidade de implementação é ilustrada pela Figura 4.3.21.

Figura 4.3.21

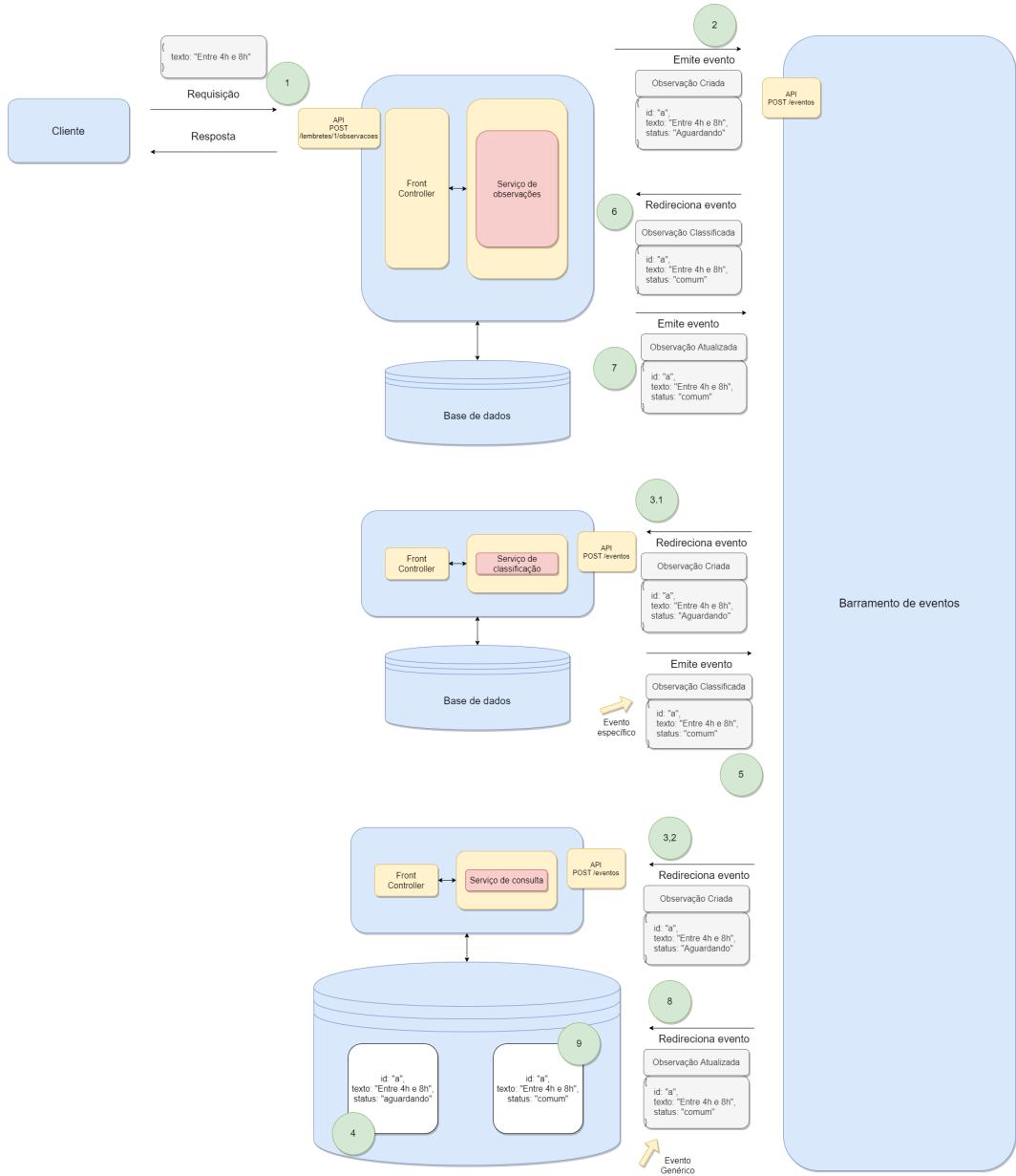


Com esta opção, o microsserviço de consulta passa a ter novas responsabilidades. Ele já é responsável por preparar os dados para disponibilização para aplicações clientes. Ele passa a ser responsável por interpretar outros tipos de eventos. Na Figura 4.3.21 ele está lidando somente com um novo tipo de evento. Entretanto, novos tipos de atualizações podem ser necessárias e, assim, a sua simplicidade e **alta coesão** seriam comprometidas.

4.3.40.4 Microsserviço próprio para a classificação - evento genérico de atualização A manipulação de eventos envolvendo observações cabe, afinal, ao microsserviço de observações. Embora não deva ser responsável pela classificação, uma vez que ela estiver pronta, ele deve ser o responsável por interpretá-la e atualizar a base de acordo. Além disso, ele deve gerar um **evento genérico de atualização** direcionado ao microsserviço de consulta. Ele vai simplesmente atualizar a base

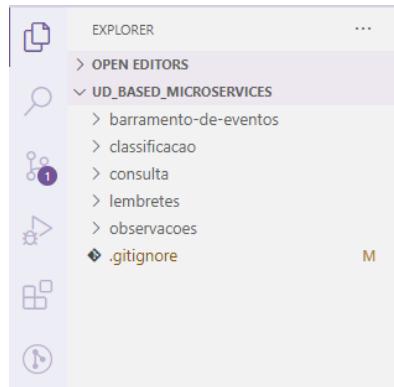
sem saber a que se refere aquela atualização. Sem, portanto, saber de detalhes específicos de observações. A Figura 4.3.22 ilustra essa estratégia.

Figura 4.3.22



4.3.41 O microsserviço de classificação Diante de todas as observações e vantagens e desvantagens abordadas, adotaremos esta estratégia. Crie uma pasta chamada *classificacao*, lado a lado com as demais, como mostra a Figura 4.3.23.

Figura 4.3.23



Ela irá abrigar os arquivos referentes à implementação do microsserviço de classificação. A seguir, execute

npm init -y

para criar um projeto. As dependências podem ser instaladas com

npm install express nodemon axios

Crie também um arquivo chamado *index.js* dentro da pasta *classificacao*. A seguir, no arquivo *package.json*, adicione o script para execução como fizemos com os demais microsserviços. Veja o Bloco de Código 4.3.22.

Bloco de Código 4.3.22

```
1  {
2      "name": "classificacao",
3      "version": "1.0.0",
4      "description": "",
5      "main": "index.js",
6      "scripts": {
7          "test": "echo \\"Error: no test specified\\" && exit 1",
8          "start": "nodemon index.js"
9      },
10     "keywords": [],
11     "author": "",
12     "license": "ISC",
13     "dependencies": {
14         "axios": "^0.21.1",
15         "express": "^4.17.1",
16         "nodemon": "^2.0.7"
17     }
18 }
```

A princípio, o microsserviço de classificação irá aguardar por eventos do tipo observação criada. O Bloco de Código 4.3.23 mostra seu código inicial, que deve ficar no arquivo *index.js*.

Bloco de Código 4.3.23

```
1  const express = require("express");
2  const app = express();
3  app.use(express.json());
4
5  app.post('/eventos', (req, res) =>{
6
7  });
8
9  app.listen(7000, () => console.log ("Classificação. Porta
10    7000));
```

Execute o novo microsserviço com

npm start

4.3.42 Microsserviço de observação registra status para cada nova observação Passamos agora a fazer os ajustes necessários para que a solução fique condizente com a arquitetura retratada pela Figura 4.3.22. Quando o microsserviço

de observações recebe uma nova requisição para criação de nova observação, ele a armazena na base com id e texto. A partir de agora, ele irá armazenar o **status** inicial, que será **aguardando** para todas as novas observações. Além disso, ao emitir um evento de observação criada, ele deve incluir o status. Veja o Bloco de Código 4.3.24.

Bloco de Código 4.3.24

```

1  app.put("/lembretes/:id/observacoes", async (req, res) => {
2      const idObs = uuidv4();
3      const { texto } = req.body;
4      //req.params dá acesso à lista de parâmetros da URL
5      const observacoesDoLembrete =
6          observacoesPorLembreteId[req.params.id] || [];
7      observacoesDoLembrete.push({ id: idObs, texto, status:
8          'aguardando' });
9      observacoesPorLembreteId[req.params.id] =
10         observacoesDoLembrete;
11     await axios.post("http://localhost:10000/eventos", {
12         tipo: "ObservacaoCriada",
13         dados: {
14             id: idObs,
15             texto,
16             lembreteId: req.params.id,
17             status: 'aguardando'
18         },
19     });
20     res.status(201).send(observacoesDoLembrete);
21 });

```

4.3.43 Barramento de eventos entrega eventos ao microsserviço de classificação Quando o barramento de eventos recebe um evento de observação criada, ele precisa redirecioná-lo para o microsserviço de classificação. O Bloco de Código 4.3.25 destaca esse ajuste.

Bloco de Código 4.3.25

```
1 app.post("/eventos", (req, res) => {
2     const evento = req.body;
3     //envia o evento para o microsserviço de lembretes
4     axios.post("http://localhost:4000/eventos", evento);
5     //envia o evento para o microsserviço de observações
6     axios.post("http://localhost:5000/eventos", evento);
7     //envia o evento para o microsserviço de consulta
8     axios.post("http://localhost:6000/eventos", evento);
9     //envia o evento para o microsserviço de classificação
10    axios.post("http://localhost:7000/eventos", evento);
11    res.status(200).send({ msg: "ok" });
12});
```

4.3.44 Microsserviço de classificação trata eventos recebidos O microsserviço de classificação, por sua vez, recebe um evento do tipo **ObservacaoCriada** e emite um outro evento do tipo **ObservacaoClassificada**, como no Bloco de Código 4.3.26.

Bloco de Código 4.3.26

```
1  const express = require("express");
2  const axios = require("axios");
3  const app = express();
4  app.use(express.json());
5  const palavraChave = "importante";
6  const funcoes = {
7      ObservacaoCriada: (observacao) => {
8          observacao.status =
9              observacao.texto.includes(palavraChave)
10             ? "importante"
11             : "comum";
12         axios.post("http://localhost:10000/eventos", {
13             tipo: "ObservacaoClassificada",
14             dados: observacao,
15         });
16     },
17     app.post("/eventos", (req, res) => {
18         funcoes[req.body.tipo](req.body.dados);
19         res.status(200).send({ msg: "ok" });
20     });
21
22     app.listen(7000, () => console.log("Classificação. Porta
23         7000"));

```

4.3.45 Microsserviço de observações lida com eventos emitidos pelo microsserviço de classificação O responsável por manipular eventos do tipo **ObservacaoClassificada** é o microsserviço de observações. Quando recebe um desses, ele atualiza a sua base e emite um evento genérico, do tipo **ObservacaoAtualizada**. Veja o Bloco de Código 4.3.27.

Bloco de Código 4.3.27

```

1   . . .
2   const observacoesPorLembreteId = {};
3   . . .
4   const funcoes = {
5     ObservacaoClassificada: (observacao) => {
6       const observacoes =
7         observacoesPorLembreteId[observacao.lembreteId];
8       const obsParaAtualizar = observacoes.find(o => o.id ===
9         observacao.id)
10      obsParaAtualizar.status = observacao.status;
11      axios.post('http://localhost:10000/eventos', {
12        tipo: "ObservacaoAtualizada",
13        dados: {
14          id: observacao.id,
15          texto: observacao.texto,
16          lembreteId: observacao.lembreteId,
17          status: observacao.status
18        }
19      });
20    }
21  };
22  . . .
23  app.post("/eventos", (req, res) => {
24    funcoes[req.body.tipo](req.body.dados);
25    res.status(200).send({ msg: "ok" });
26  );

```

4.3.46 Microsserviço de consulta lida com eventos de observações atualizadas após classificação O evento do tipo **ObservacaoAtualizada** tem como destino o microsserviço de consulta. Ele busca a observação pelo id e substitui o objeto existente por aquele incluído no evento. É fundamental que essa atualização seja “burra”. Não cabe ao microsserviço de consulta saber, por exemplo, que essa é uma atualização envolvendo o status da observação recebida. Veja o Bloco de Código 4.3.28.

Bloco de Código 4.3.28

```

1  const funcoes = {
2      LembreteCriado: (lembrete) => {
3          baseConsulta[lembrete.contador] = lembrete;
4      },
5      ObservacaoCriada: (observacao) => {
6          const observacoes =
7              baseConsulta[observacao.lembreteId]["observacoes"] ||
8                  [];
9          observacoes.push(observacao);
10         baseConsulta[observacao.lembreteId]["observacoes"] =
11             observacoes;
12     },
13     ObservacaoAtualizada: (observacao) => {
14         const observacoes =
15             baseConsulta[observacao.lembreteId]["observacoes"];
16         const indice = observacoes.findIndex((o) => o.id ===
17             observacao.id);
18         observacoes[indice] = observacao;
19     },
20 };

```

4.3.47 Microsserviços descartam eventos que não lhes são de interesse

Neste momento, alguns dos microsserviços estão gerando erros em tempo de execução, especialmente aqueles que fazem o tratamento dos eventos recebidos utilizando um mapa de funções. Ocorre que o barramento de eventos faz um simples **broadcast** a cada vez que recebe um evento. Isso quer dizer que cada microsserviço recebe todo tipo de evento. Obviamente, eles não estão preparados para lidar com todos os tipos de eventos. O microsserviço de consulta, por exemplo, não está interessado no evento de tipo **ObservacaoClassificada**. O microsserviço de classificação, por sua vez, não está interessado no evento de tipo **ObservacaoAtualizada**. Assim, precisamos tratar os casos em que os microsserviços recebem eventos que não lhes são de interesse, caso contrário, o acesso ao mapa de funções que cada um possui causará erros em tempo de execução. O que faremos é extremamente simples. Utilizaremos uma estrutura **try/catch** para, como o nome sugere “tentar” acessar o mapa de funções utilizando o tipo do evento recebido como chave. Caso o acesso ocorra com sucesso, o evento recebido é de interesse e o tratamento será realizado por uma das funções do mapa. A execução do bloco **try** termina com sucesso nesse caso. E o bloco **catch** é ignorado. Caso contrário, o evento recebido não é de interesse e haverá um erro, já que não existirá função para seu tratamento no mapa. A execução do **try** é interrompida e desviada para o bloco **catch**. O bloco **catch** apenas ignora o fluxo de execução recebido. Assim, eventos que não são de interesse são apenas ignorados. Começamos

ajustando o microsserviço de observações. Veja o Bloco de Código 4.3.29.

Bloco de Código 4.3.29

```
1 . . .
2 app.post("/eventos", (req, res) => {
3   try{
4     funcoes[req.body.tipo](req.body.dados);
5   }
6   catch (err){}
7   res.status(200).send({ msg: "ok" });
8 });


```

O microsserviço de consulta também precisa de tratamento, como mostra o Bloco de Código 4.3.30.

Bloco de Código 4.3.30

```
1 . . .
2 app.post("/eventos", (req, res) => {
3   try {
4     funcoes[req.body.tipo](req.body.dados);
5   } catch (err) {}
6   res.status(200).send({ msg: "ok" });
7 });
8 . . .


```

O microsserviço de classificação também requer ajustes, de maneira análoga aos demais. Veja o Bloco de Código 4.3.31.

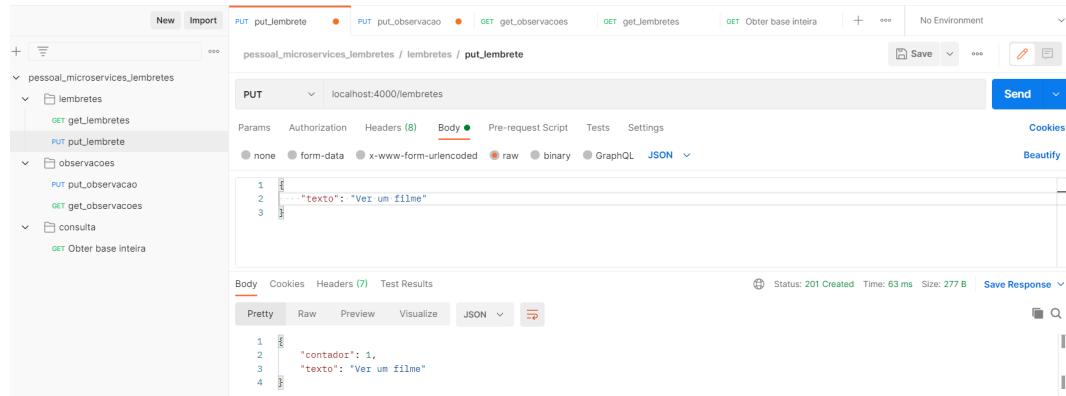
Bloco de Código 4.3.31

```
1 . . .
2 app.post("/eventos", (req, res) => {
3   try {
4     funcoes[req.body.tipo](req.body.dados);
5   } catch (err) {}
6   res.status(200).send({ msg: "ok" });
7 });
8 . . .


```

4.3.48 Testando a solução completa após descartes de eventos Para testar o funcionamento do sistema, pode ser uma boa ideia reiniciar todos os microsserviços. Assim garantimos que suas bases de dados não tem dados antigos que poderiam estar incondizentes com os novos detalhes de implementação. Feito isso, comece fazendo a inserção de um novo lembrete, como na Figura 4.3.24.

Figura 4.3.24



Anote o valor devolvido como identificador do lembrete inserido. Ele será usado para a inserção de novas observações. Faça as inserções de observações ilustradas nas figuras 4.3.25 e 4.3.26. Observe os resultados esperados. Quando a primeira inserção acontece, o microsserviço de observações nos devolve a base inteira imediatamente. Neste instante, a observação ainda não está classificada. Entretanto, ele já emitiu um evento de criação de observação que disparou todo o fluxo de classificação e atualização da base de consulta e de sua própria base. Repare no resultado esperado depois de fazer a segunda inserção de observação.

Figura 4.3.25

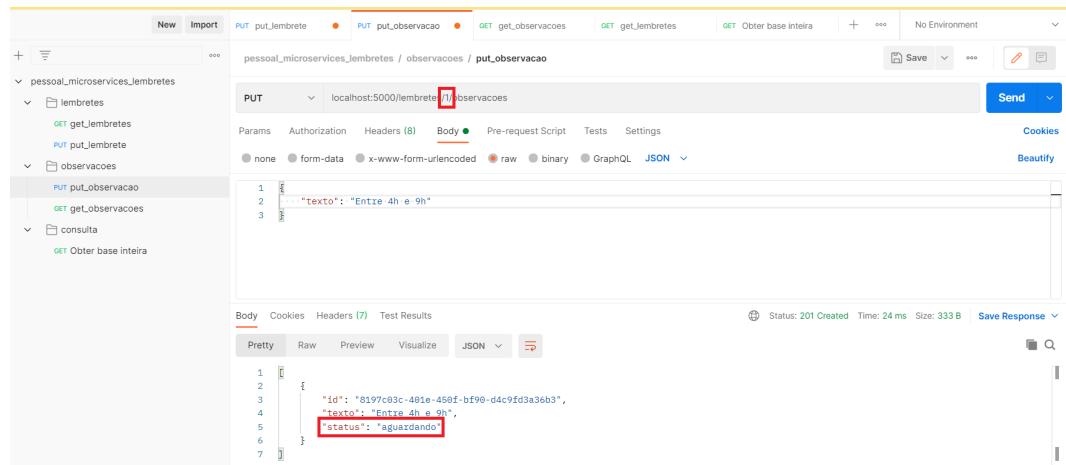
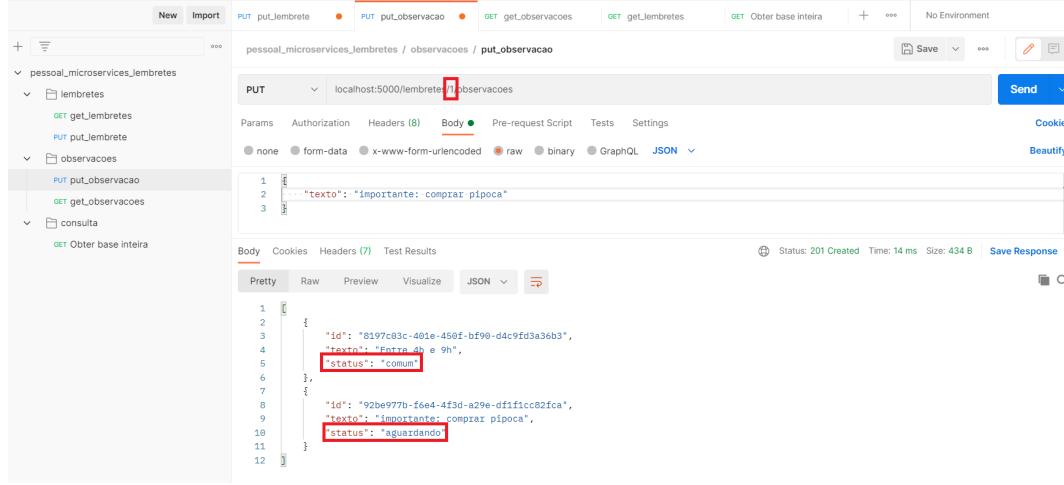
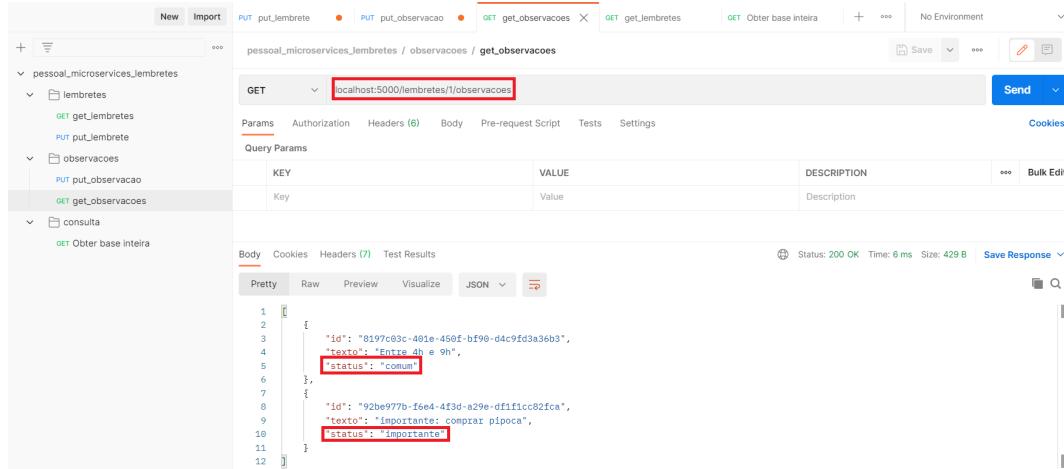


Figura 4.3.26



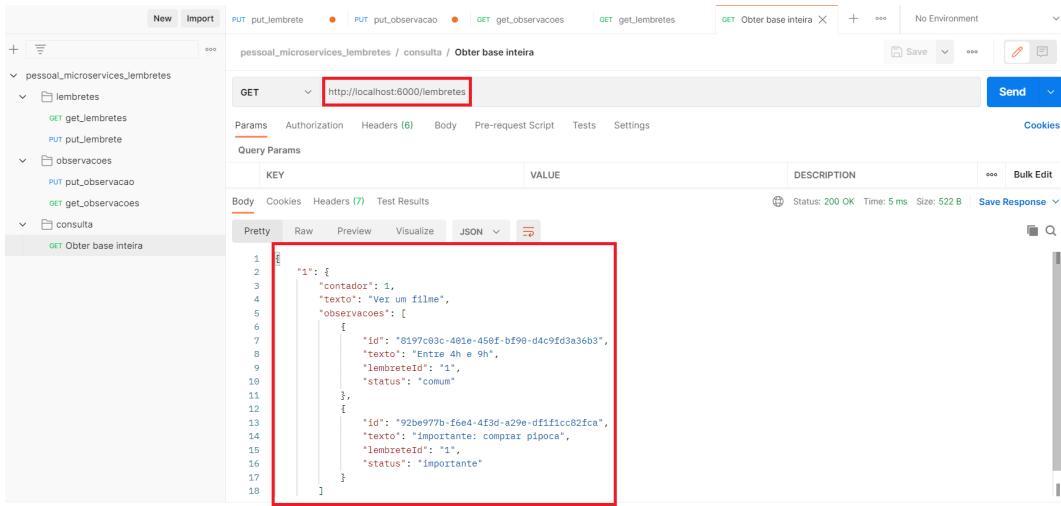
Envie uma requisição GET para o microsserviço de observações e verifique que todas as observações existentes em sua base já estão classificadas, como ilustra a Figura 4.3.27.

Figura 4.3.27



Finalmente, envie uma requisição GET para o microsserviço de consulta a fim de obter uma cópia da base inteira, incluindo lembretes e observações. Veja a Figura 4.3.28.

Figura 4.3.28



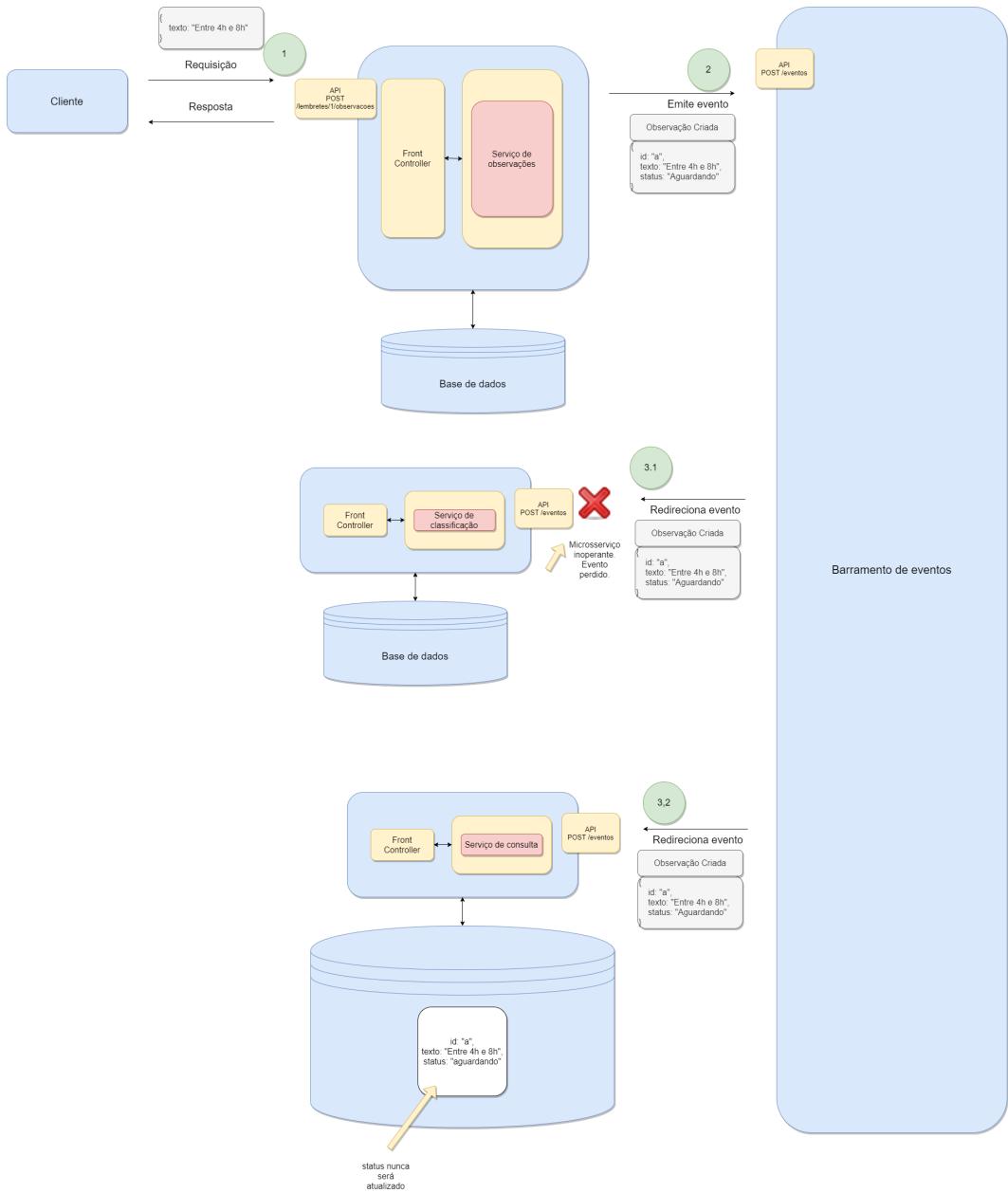
```

1: [
2:   {
3:     "id": "8397c03c-481e-458f-b198-d4c9fd3a36b3",
4:     "texto": "Ver um filme",
5:     "observacoes": [
6:       {
7:         "id": "8397c03c-481e-458f-b198-d4c9fd3a36b3",
8:         "texto": "Entre 4h e 9h",
9:         "lembreteId": "1",
10:        "status": "comum"
11:      },
12:      {
13:        "id": "92be977b-f6ed-4f3d-a29e-dff1ficc82fca",
14:        "texto": "Importante: comprar pipoca",
15:        "lembreteId": "1",
16:        "status": "importante"
17:      }
18:    ]
19:  }
20: ]

```

4.3.49 Como lidar com eventos perdidos? O sistema funciona adequadamente desde que todos os microserviços estejam em pleno funcionamento. O que ocorre com uma observação inserida em algum momento em que o microserviço de classificação não está em funcionamento? Veja a Figura 4.3.29.

Figura 4.3.29

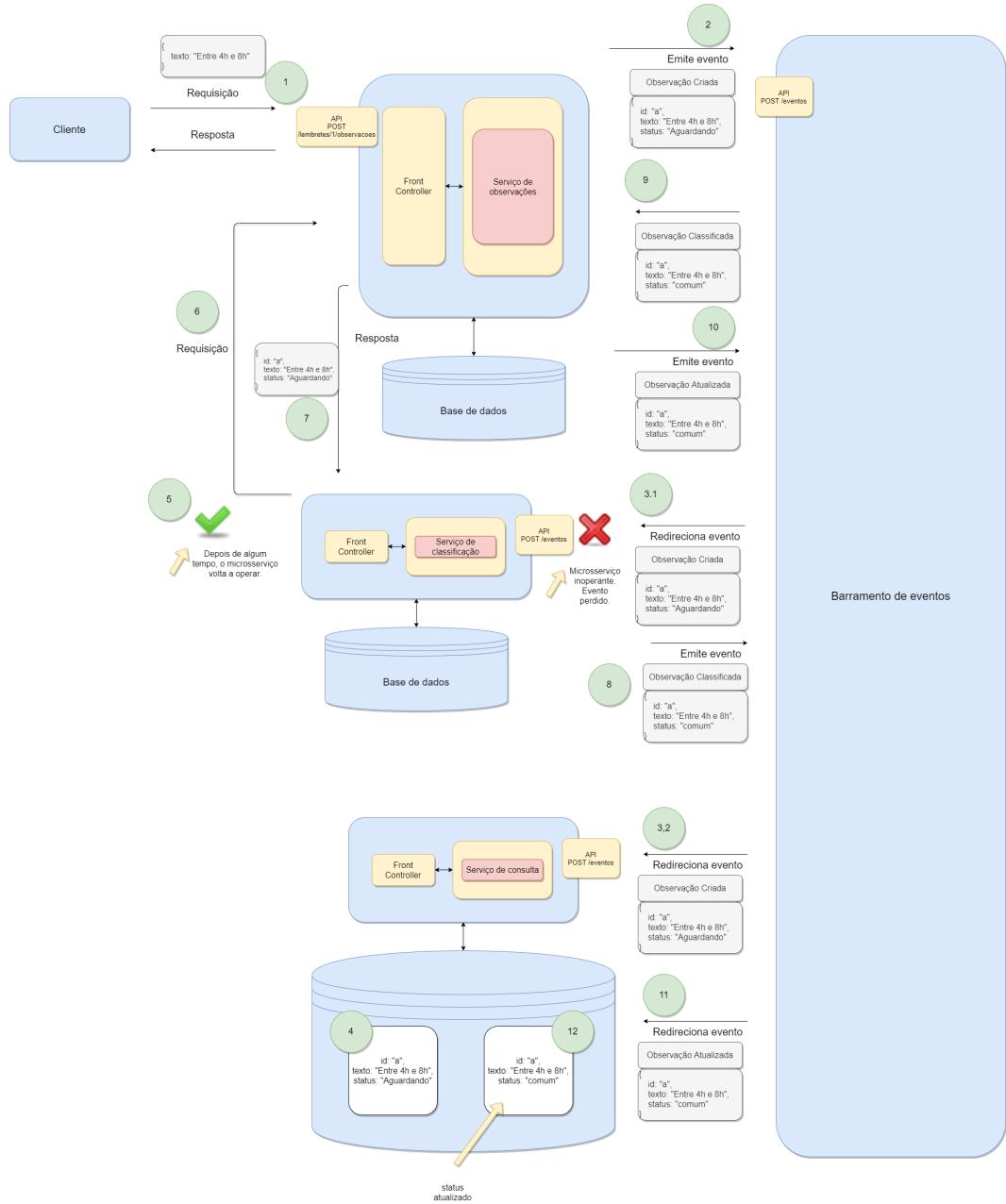


Uma vez que um evento tenha sido perdido, os microsserviços que tinham interesse por ele nunca mais poderão vê-lo novamente. Como ilustra a Figura 4.3.29, um exemplo de consequência são as observações que nunca são classificadas caso tenham sido inseridas em algum momento em que o microsserviço de classificação estava inoperante.

4.3.49.1 Eventos perdidos - Requisição síncrona Há algumas possibilidades para resolver esse problema. Uma delas consiste em fazer com que o microsserviço de classificação, uma vez que volte a operar, peça uma cópia das bases de lembretes

e observações para seus respectivos microsserviços. Essa possibilidade é ilustrada na Figura 4.3.30.

Figura 4.3.30



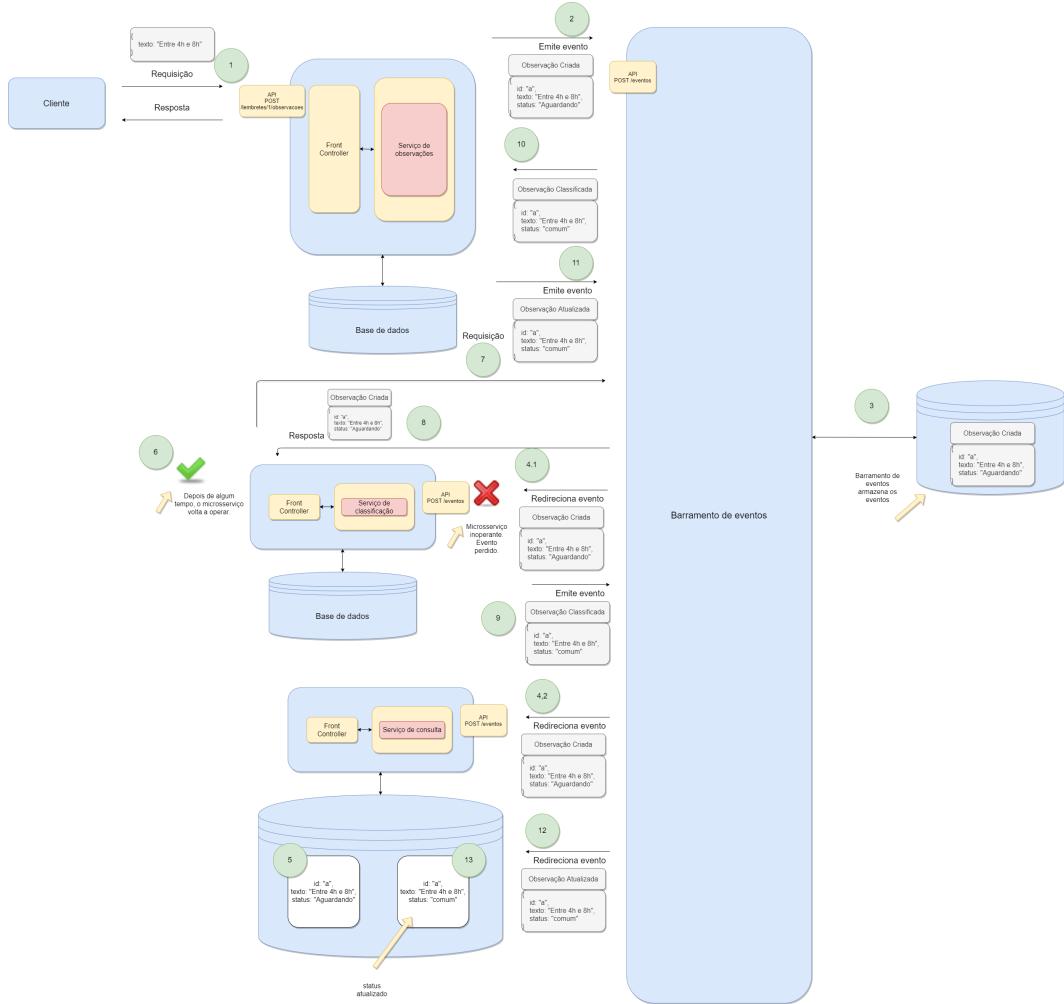
Embora resolva o problema, esta estratégia traz o uso da comunicação síncrona entre microsserviços, algo que temos tentado evitar.

4.3.49.2 Eventos perdidos - Armazenamento de eventos no barramento de eventos

Outra possibilidade é fazer com que o barramento de eventos armazene

os eventos que não conseguiram entregar aos destinatários. Neste cenário, caso o microserviço de classificação fique inoperante por algum tempo e perca alguns eventos, quando voltar a operar ele pode solicitar ao barramento os eventos que deixou de receber. Veja a Figura 4.3.31.

Figura 4.3.31



4.3.50 Barramento de eventos armazena eventos Adotaremos a estratégia em que o barramento de eventos armazena os eventos recebidos. No Bloco de Código 4.3.32, definimos uma base de dados (em memória volátil) para o barramento de eventos e fazemos com que ele armazene cada evento recebido. Além disso, adicionamos um novo endpoint que viabiliza a obtenção da base.

Bloco de Código 4.3.32

```

1   . . .
2   const eventos = []
3
4   app.post("/eventos", (req, res) => {
5       const evento = req.body;
6       eventos.push(evento)
7       //envia o evento para o microsserviço de lembretes
8       axios.post("http://localhost:4000/eventos", evento);
9       //envia o evento para o microsserviço de observações
10      axios.post("http://localhost:5000/eventos", evento);
11      //envia o evento para o microsserviço de consulta
12      axios.post("http://localhost:6000/eventos", evento);
13      //envia o evento para o microsserviço de classificação
14      axios.post("http://localhost:7000/eventos", evento);
15      res.status(200).send({ msg: "ok" });
16  });
17
18  app.get('/eventos', (req, res) => {
19      res.send(eventos)
20  })
21  . . .

```

4.3.51 Microsserviço de consulta solicita eventos potencialmente perdidos

Um dos microsserviços que está interessado em eventos potencialmente perdidos é o microsserviço de **consulta**. Se ele ficar inoperante em um período de tempo em que inserções de novos lembretes e/ou novas observações ocorrerem, a sua base de dados ficará desatualizada. Por isso, sempre que entrar em funcionamento, ele enviará um pedido ao barramento de eventos pelos eventos em que ele está interessado. Tal envio será feito por uma requisição HTTP, o que quer dizer que ele passa a precisar do pacote *axios*. Faça a sua instalação com

npm install axios

O Bloco de Código 4.3.33 mostra o microsserviço de consulta entrando em execução e fazendo o pedido ao barramento de eventos. A seguir, ele processa cada evento recebido, ignorando aqueles que não lhe são de interesse. O *axios* entrega os resultados de uma requisição GET em uma propriedade chamada **data**. Veja o Link 4.3.3.

Link 4.3.3
<https://github.com/axios/axios#response-schema>

Bloco de Código 4.3.33

```

1   . . .
2   const axios = require("axios");
3   . . .
4   //não esqueça do async
5   app.listen(6000, async () => {
6       console.log("Consultas. Porta 6000");
7       const resp = await
8           axios.get("http://localhost:10000/eventos");
9       //axios entrega os dados na propriedade data
10      resp.data.forEach((valor, indice, colecao) => {
11          try {
12              funcoes[valor.tipo](valor.dados);
13          } catch (err) {}
14      });
15  });

```

4.3.52 Testando a solução após implementação do tratamento de eventos perdidos

Realize um teste da seguinte forma.

- Certifique-se de que todos os microsserviços estão operando e com a base limpa.
- Faça a inserção de um novo lembrete.
- Consulte a base do microsserviço de consulta. Ela deve incluir o lembrete recém inserido.
- Interrompa o funcionamento do microsserviço de consulta.
- Faça a inserção de um novo lembrete.
- Coloque o microsserviço de consulta novamente em funcionamento.
- Consulte a base do microsserviço de consulta. A base deverá incluir os dois lembretes.

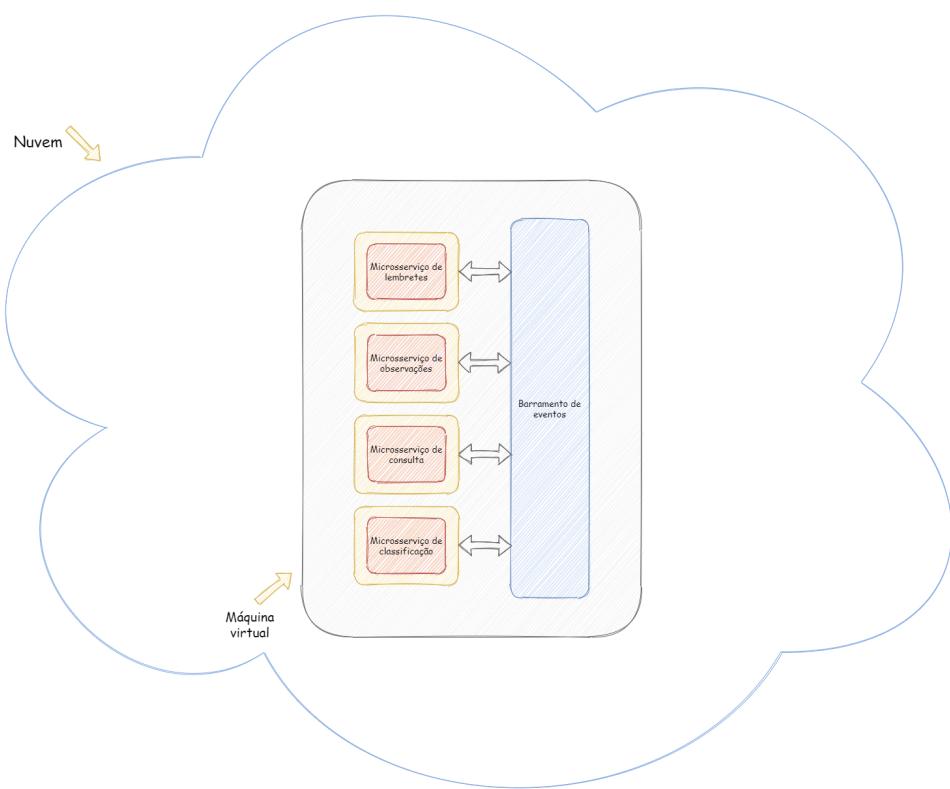
O tratamento é análogo para os demais microsserviços: todos os que tiverem interesse por determinados eventos precisam consultar o barramento de eventos quando entrarem em funcionamento.

4.3.53 Implantação

Há diferentes formas viáveis para a implantação da solução, cada qual com vantagens e desvantagens. Nesta seção avaliamos algumas possibilidades.

4.3.53.1 Implantação - Única máquina virtual - Única instância por microsserviço. Uma forma de implantação da solução consiste na alocação de uma máquina virtual em um serviço de computação em nuvem, como ilustra a Figura 4.3.32.

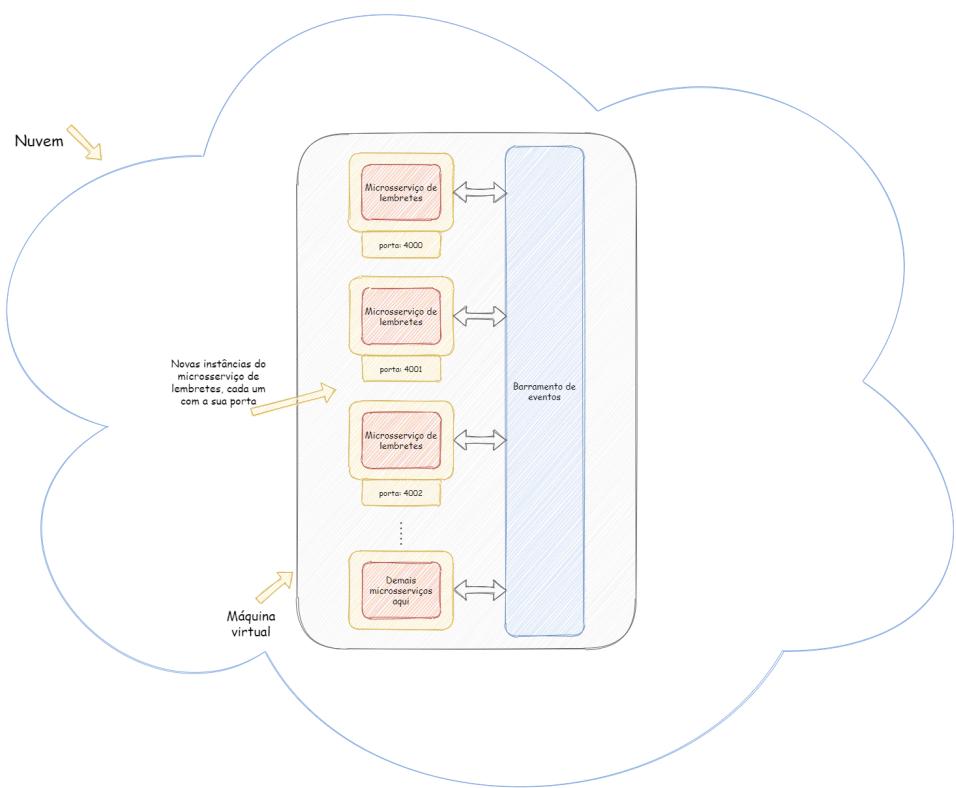
Figura 4.3.32



- Vantagens
 - **Simplicidade.** O modelo é muito fácil de se entender e o seu uso é extremamente simples.
 - **Custo.** A tendência é que essa solução seja relativamente barata.
- Desvantagens
 - **Único ponto de falha.** Esta desvantagem pode ser abstraída caso um serviço de computação em nuvem seja utilizado. Entretanto, se for necessário reiniciar a máquina virtual, por exemplo, todos os microsserviços se tornarão inoperantes durante aquele período.
 - **Escalabilidade não trivial.** O que fazer caso um dos microsserviços fique sobrecarregado?

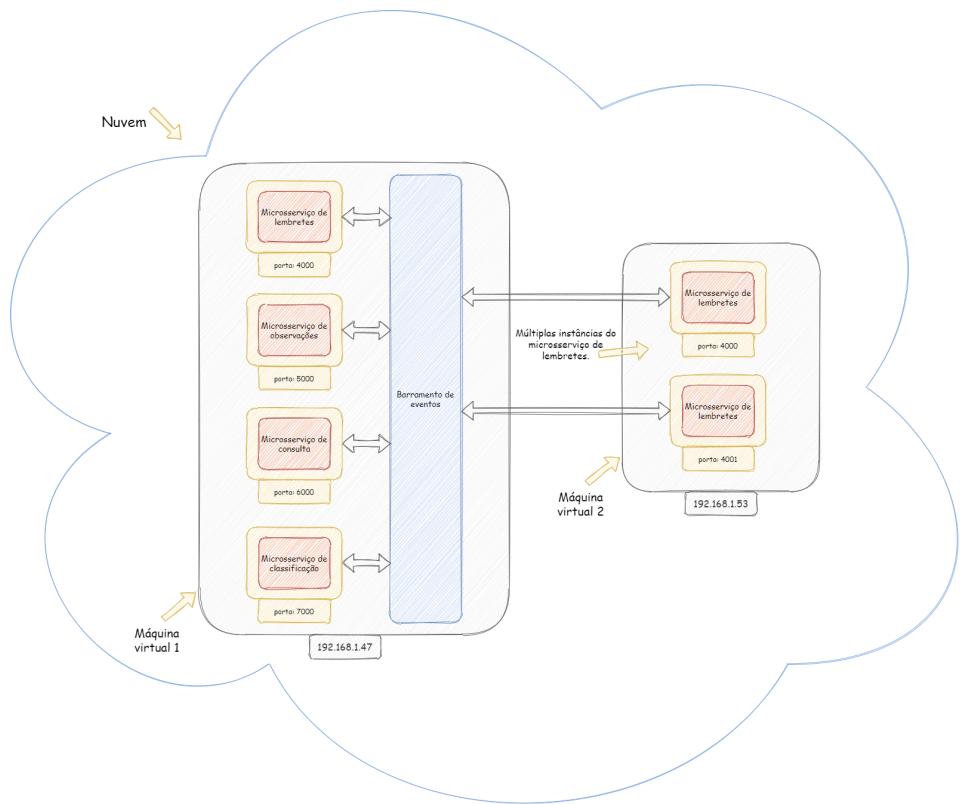
4.3.53.2 Implantação - Múltiplas máquinas virtuais - Potenciais múltiplas instâncias para cada microsserviço. Uma das desvantagens do modelo da Figura 4.3.32 está associado à escalabilidade. Esse problema pode ser resolvido colocando-se em funcionamento múltiplas instâncias dos microsserviços que tiverem maior carga de trabalho potencial. A Figura 4.3.33 ilustra essa possibilidade.

Figura 4.3.33



Embora o modelo da Figura 4.3.33 de certa forma resolva o problema de instâncias com alta demanda, ainda há o problema de todas elas estarem sob gerenciamento de uma única máquina virtual. Nada impede, entretanto, que múltiplas máquinas virtuais sejam alocadas, como na Figura 4.3.34.

Figura 4.3.34



Seja com uma única máquina virtual, seja utilizando diversas máquinas virtuais, esta estratégia tende a tornar a implementação do barramento de eventos mais complexa e acoplada a essa solução computacional específica. Ou seja, comprometemos o nível de reusabilidade do barramento de eventos. Inclusive, o problema pode se tornar maior caso o número de instâncias de cada microsserviço ou mesmo de máquinas virtuais possa variar ao longo do tempo. Sob forte demanda, pode ser interessante manter um número alto de máquinas virtuais alocadas. Quando a

demandar baixar, pode ser interessante reduzir o número de recursos alocados.

- Vantagens
 - **Escalabilidade.** O modelo pode se ajustar de acordo com a demanda, poupando recursos quando possível.
- Desvantagens
 - **Implementação do barramento de eventos.** A implementação do barramento de eventos torna-se não trivial e acoplada a essa solução específica. A cada instante, ele precisa decidir quais são as portas e quais são os endereços ip para os quais enviar os eventos. Não é razoável manter essa **estrutura de seleção** no barramento de eventos.

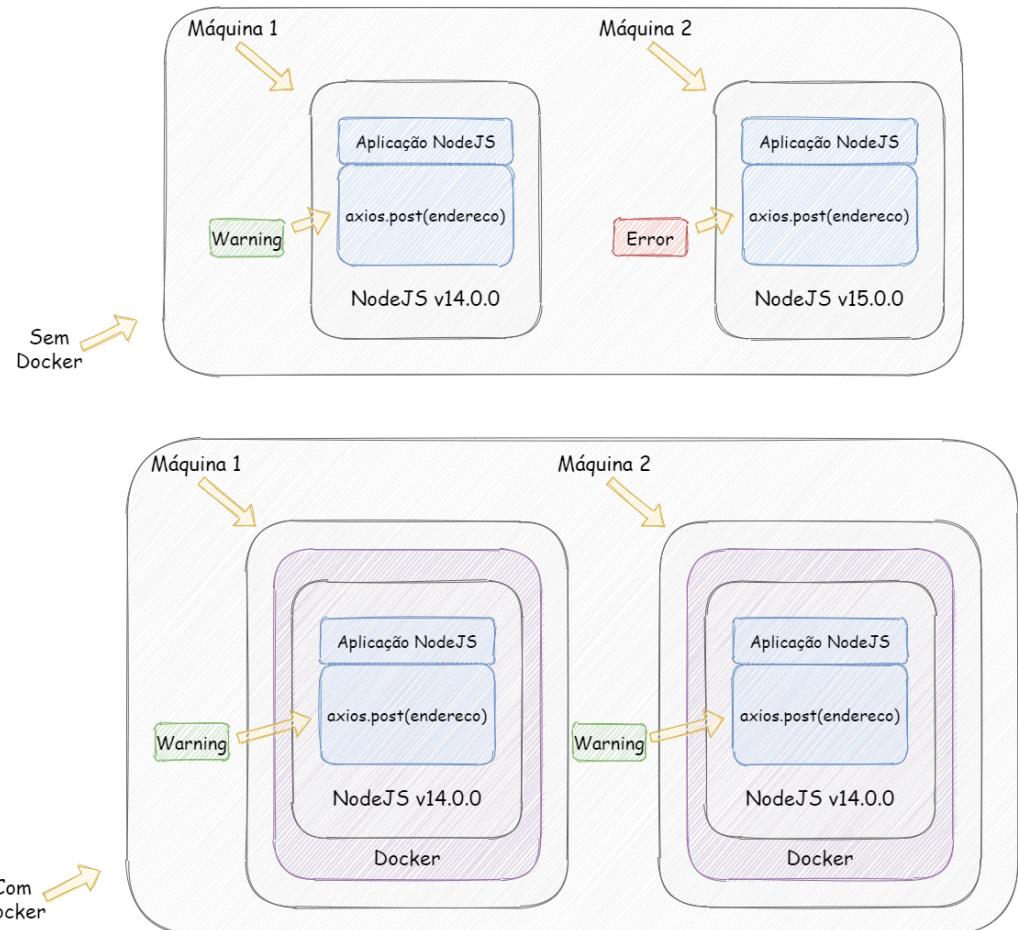
4.3.54   é uma ferramenta utilizada para criar e gerenciar **contêineres**.

DEFINIÇÃO

Um **contêiner** é uma unidade de software isolada com código a ser executado incluindo as suas dependências.

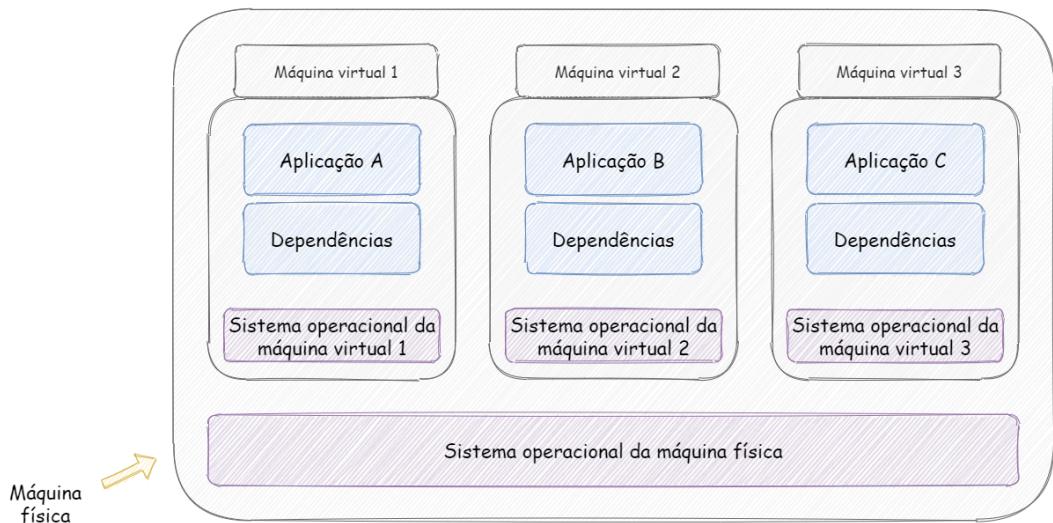
Quando utilizamos um contêiner, a expectativa é que ele execute **sem interfe- rência do ambiente externo**. Espera-se, também, que a sua existência não traga **impactos relevantes ao desempenho da aplicação**. Quando uma aplicação é distribuída utilizando-se um contêiner, temos a vantagem de garantir que as dependências de que necessita, cada uma em sua versão correta, estejam disponíveis. Veja a Figura 4.3.35.

Figura 4.3.35



O uso do  se assemelha ao uso de **máquinas virtuais**. Entretanto, há diferenças fundamentais que, em geral, o torna mais vantajoso. A Figura 4.3.36 ilustra o uso de máquinas virtuais.

Figura 4.3.36

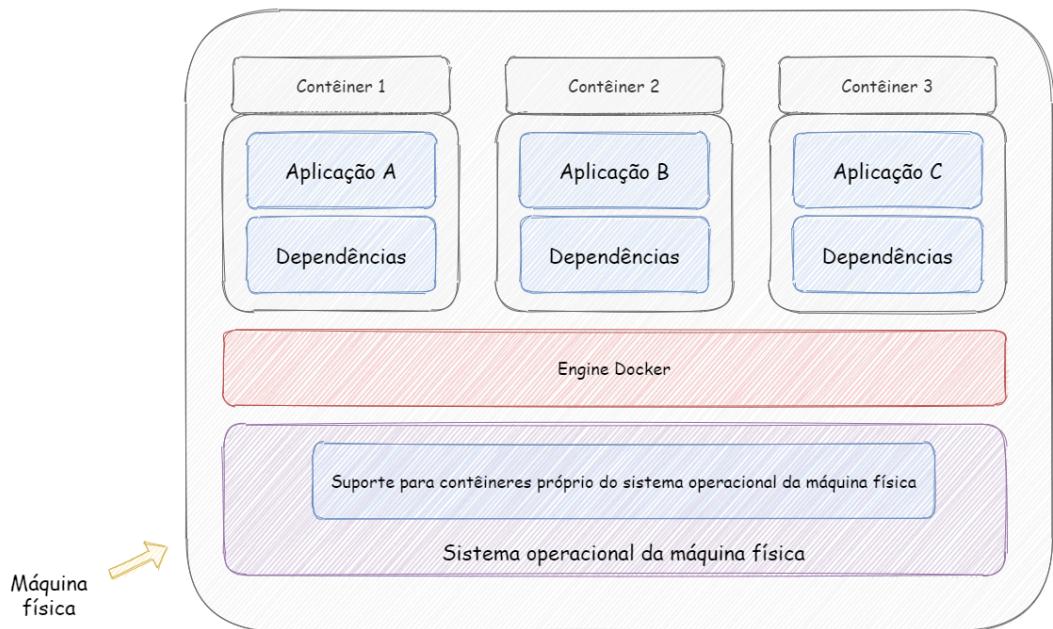


Do ponto de vista de isolamento e distribuição de aplicações com dependências e suas versões garantidas, as máquinas virtuais resolvem o problema tão bem quanto os contêineres. Entretanto, seu uso traz algumas desvantagens.

- Código duplicado, especialmente dos sistemas operacionais.
- Desperdício de espaço.
- Potencial perda de desempenho.

Usando o , o equivalente da Figura 4.3.36 é exibido na Figura 4.3.37.

Figura 4.3.37



Algumas considerações importantes observadas na Figura 4.3.37.

- O engine do é significativamente mais “leve” quando comparado a uma máquina virtual.
- Cada contêiner pode, de fato, ter partes de um sistema operacional nele presentes. Entretanto, nada comparado a um sistema operacional completo.

4.3.55 Obtendo o Docker Há versões do para **Linux**, **MacOS** e **Windows**. Visite o Link 4.3.4 e obtenha o **Docker Desktop** caso esteja utilizando Windows ou MacOS.

Link 4.3.4
<https://www.docker.com/products/docker-desktop>

Quando terminar, use

docker version

para testar a instalação.

É importante entender quais são as principais ferramentas envolvidas.

- **Docker Engine** viabiliza a execução de contêineres.
- **Docker CLI** é um cliente de linha de comando por meio do qual pode-se interagir com o Docker, como o nome sugere, pela linha de comando.
- **Docker Compose** é uma ferramenta que viabiliza a execução de aplicações que utilizam múltiplos contêineres. Basta um arquivo YAML em que os serviços são configurados para que, a seguir, com um único comando, eles possam ser executados.
- **Kubernetes³** é uma plataforma que gerencia serviços executados por contêineres. Ele é capaz de disponibilizar acesso a múltiplos serviços por meio de um único endereço e operar como um balanceador de carga. Além disso, também é capaz de reiniciar contêineres, substituir contêineres entre muitas outras funcionalidades.
- **Docker Desktop** é um pacote que inclui o **Docker Engine**, o **Docker CLI**, o **Docker Compose** e o **Kubernetes**.
- **Docker Hub** é um repositório de imagens .

Nota. No Windows, após a instalação do , é possível que apareça uma nova tela sugerindo a instalação do **Windows Subsystem for Linux (WSL) 2**. Trata-se de um Kernel Linux completo desenvolvido pela Microsoft. Com ele, é possível executar contêineres nativamente sem emulação. Também deixa de ser necessário manter scripts próprios para Windows e para Linux. É importante observar que ele está disponível apenas no **Windows 10, build 1903 em diante**. Veja mais no Link 4.3.5. Não deixe de seguir as instruções de instalação da Microsoft, disponíveis na página acessível por meio do Link 4.3.6.

Link 4.3.5

<https://docs.docker.com/docker-for-windows/wsl/>

Link 4.3.6

<https://docs.microsoft.com/en-us/windows/wsl/install-win10#step-1---enable-the-windows-subsystem-for-linux>

³Pronúncia e origem da palavra Kubernetes: <https://www.youtube.com/watch?v=uMA7qqXIXBk>

Nota. O Play with Docker (PWD) é um projeto que permite que usuários executem comandos Docker usando seu navegador. Visite o Link 4.3.7 para acessá-lo.

Link 4.3.7
<https://labs.play-with-docker.com>

4.3.56 Hello, ! Nesta seção, construiremos uma pequena aplicação com  com o intuito de entender as partes mais fundamentais do .

Crie uma nova pasta em seu workspace e execute

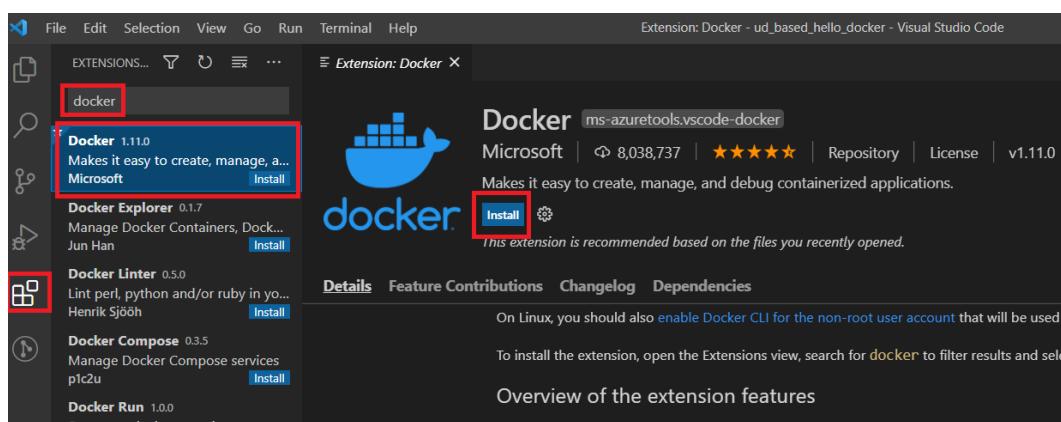
npm init -y

para criar um projeto . A seguir, execute

npm install express

para instalar o pacote . Caso utilize o VS Code, pode ser interessante instalar a **extensão Docker**. Ela é desenvolvida pela própria Microsoft. Para fazer a sua instalação, clique no ícone de extensões e busque por docker. A seguir, clique install. Veja a Figura 4.3.38.

Figura 4.3.38



Crie um arquivo chamado *index.js*. Seu conteúdo aparece no Bloco de Código 4.3.34.

Bloco de Código 4.3.34

```
1 const express = require("express");
2 const app = express();
3 app.use(express.json());
4
5 app.get("/hey-docker", (req, res) => {
6     res.status(200).json({
7         mensagem: "Hey, Docker!!",
8     });
9 });
10 app.listen(5200, () => console.log("up and running inside
11 docker"));
```

O próximo passo é criar um arquivo chamado *Dockerfile*, sem extensão alguma. Nele podemos descrever uma **imagem**  . Ou seja, os arquivos, ferramentas etc que desejamos que estejam disponíveis quando um **contêiner**  for colocado em execução em função dela. Pense em uma imagem  como uma classe, uma descrição. Por outro lado, pense em um contêiner Docker como um objeto construído a partir desta classe. Veja o conteúdo do arquivo *Dockerfile* no Bloco de Código 4.3.35.

Bloco de Código 4.3.35

```
1 #comentários são feitos com #
2 #desde que # seja o primeiro símbolo na linha
3 #essa linguagem não é case sensitive
4 #mas é boa prática usar maiúsculas para diferenciar comandos
   de argumentos
5 #queremos uma imagem com o node versão 14
6 FROM node:14

7
8 # um diretório no sistema de arquivos do contêiner para os
   comandos a seguir
9 WORKDIR /app

10
11 #copiamos o package.json para poder executar npm install
12 COPY package.json .

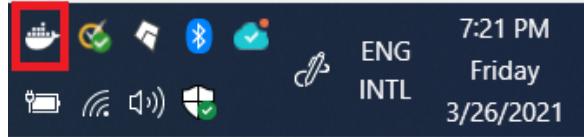
13
14 #executamos npm install
15
16 RUN npm install
17
18 #copiamos os demais arquivos
19 COPY . .
20
21 #tornamos o contêiner disponível na porta 5200
22 EXPOSE 5200
23
24 #colocamos o aplicativo em execução
25 CMD ["node", "index.js"]
```

Nota. node:14 é uma imagem disponível no *Docker Hub*. Veja o Link 4.3.8
Link 4.3.8

https://hub.docker.com/_/node

Nota. Para usar os comandos a seguir, certifique-se de que o  está em execução. Ele costuma exibir seu ícone na “system tray” do sistema operacional, como mostra a Figura 4.3.39.

Figura 4.3.39



A seguir, use

docker build .

para construir a imagem. O símbolo “.” indica o diretório em que se encontra o arquivo *Dockerfile*. Uma vez construída a imagem, você verá um feedback textual parecido com aquele exibido pela Figura 4.3.40.

Figura 4.3.40

```
[+] Building 1.8s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] transferring dockerfile: 562B
=> [internal] load .dockerignore
=> [internal] transferring context: 2B
=> [internal] load metadata for docker.io/library/node:14
=> [internal] load build context
=> [internal] transferring context: 21.45kB
=> [1/5] FROM docker.io/library/node:14@sha256:fe842f5b828c121514d62cbe0ace0927aec4f3130297180c3343e54e7ae97362
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY package.json .
=> CACHED [4/5] RUN npm install
=> CACHED [5/5] COPY .
=> exporting to image
=> => exporting layers
=> => writing image sha256:ea71b1d1dcdb9a23356e4cf49d36dc1ad4f37a16fa8f0ff4ac860170c1f53377
```

A última linha, destacada na Figura 4.3.40, mostra o identificador da imagem gerada. Neste exemplo, ele é igual a

ea71b1d1dcdb9a23356e4cf49d36dc1ad4f37a16fa8f0ff4ac860170c1f53377

Ele pode ser usado para colocar a imagem em execução. Para isso, use

docker run -p 5200:5200 ea71b1d1dcdb

Basta usar os 12 primeiros caracteres. Especificamos duas portas: primeiro

dizemos qual a porta do computador host para a qual as requisições serão enviadas. A seguir, especificamos qual a porta do contêiner para a qual elas devem ser direcionadas. Faça um teste usando um cliente HTTP (como o seu navegador ou o Postman) visitando

http://localhost:5200/hey-docker

Cada contêiner colocado em execução recebe um nome automático. Use

docker ps

para verificar o nome do seu. Veja a Figura 4.3.41.

Figura 4.3.41

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
d96c092ff53c	ea71b1d1dcdb	"docker-entrypoint.s..."	17 minutes ago	Up 17 minutes	0.0.0.0:5200->5200/tcp	clever_turing

Para encerrar a sua execução, use

docker stop nome_do_conteiner

4.3.57 Comandos

O  possui diversos comando que podem ser úteis.

- **docker build -t tag_para_o_conteiner .** Cria uma imagem e associa a ela a **tag** `tag_para_o_conteiner`. O “.” ao final indica o diretório em que se encontra o arquivo *Dockerfile*.
- **docker run [id ou tag]** Um contêiner pode ser colocado em execução por meio de seu id e por meio de sua tag.
- **docker run -it [id ou tag][novo_comando]** Coloca um contêiner em execução substituindo o comando padrão por novo_comando.
- **docker ps** Traz informações sobre todos os contêineres ativos no momento.
- **docker exec -it id comando** Executa um comando em um contêiner que já está em execução. Exemplo: `docker exec -it id sh` coloca um “shell” em execução dentro do contêiner.
- **docker logs id** Exibe os logs do contêiner cujo id foi especificado.
- **docker help** Exibe os comandos Docker disponíveis.
- **docker comando --help** Exibe as opções do comando especificado. Exemplo: `docker build --help`.
- **docker images** Lista informações sobre as imagens existentes.

4.3.58 Implantando o microsserviço de lembretes

Para colocar o microsserviço de lembretes em execução, crie um arquivo *Dockerfile* - sem extensão - no seu diretório raiz (mesmo diretório em que se encontra o arquivo *package.json*). Veja o seu conteúdo no Bloco de Código 4.3.36.

Nota. Os microsserviços possuem o texto “localhost” fixo no código. Quando utilizada por uma aplicação executada por um contêiner, essa constante se refere ao próprio contêiner. Isso quer dizer que os microsserviços não poderão se comunicar. É possível substituir o texto “localhost” pelo ip da máquina host para que eles possam se comunicar.

Bloco de Código 4.3.36

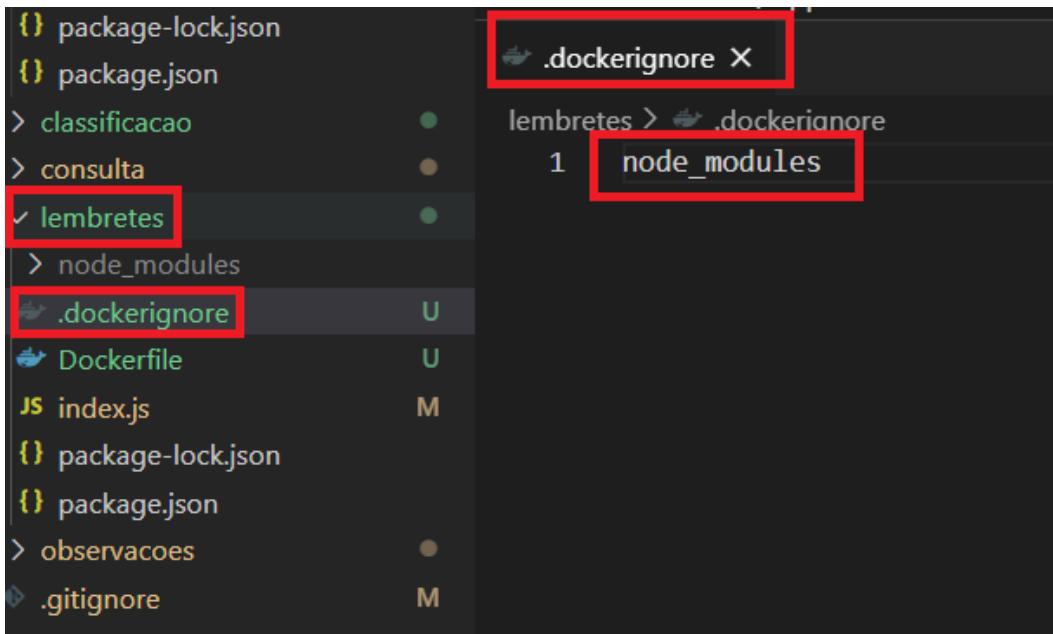
```

1  #alpine: imagem Linux "pequena"
2  FROM node:alpine
3
4  # um diretório no sistema de arquivos do contêiner para os
   comandos a seguir
5  WORKDIR /app
6
7  #copia o arquivo package.json para poder executar npm
   install
8  COPY package.json .
9
10 #instala as dependências
11 RUN npm install
12
13 #copia todo o conteúdo .local para a imagem
14 COPY . .
15
16 #executa quando o contêiner entrar em execução
17 CMD ["npm", "start"]

```

O comando `COPY` está copiando todo o conteúdo do diretório local para o sistema de arquivos da imagem. Entretanto, a pasta `node_modules` não é de interesse já que o comando `npm install` é executado quando a imagem é criada. Além disso, pode ser que existam dependências locais necessárias somente em tempo de desenvolvimento que não são necessárias em tempo de execução. Por isso, vamos criar um arquivo chamado `.dockerignore` e especificar que `node_modules` não deve ser copiada. Ele precisa ficar no diretório em que o comando `build` será executado. Neste caso, no mesmo diretório em que se encontra o arquivo `Dockerfile`. Veja a Figura 4.3.42.

Figura 4.3.42



O conteúdo do arquivo *.dockerignore* é

node_modules

ou seja, o nome da pasta que não deve ser copiada para a imagem. Construa a imagem com

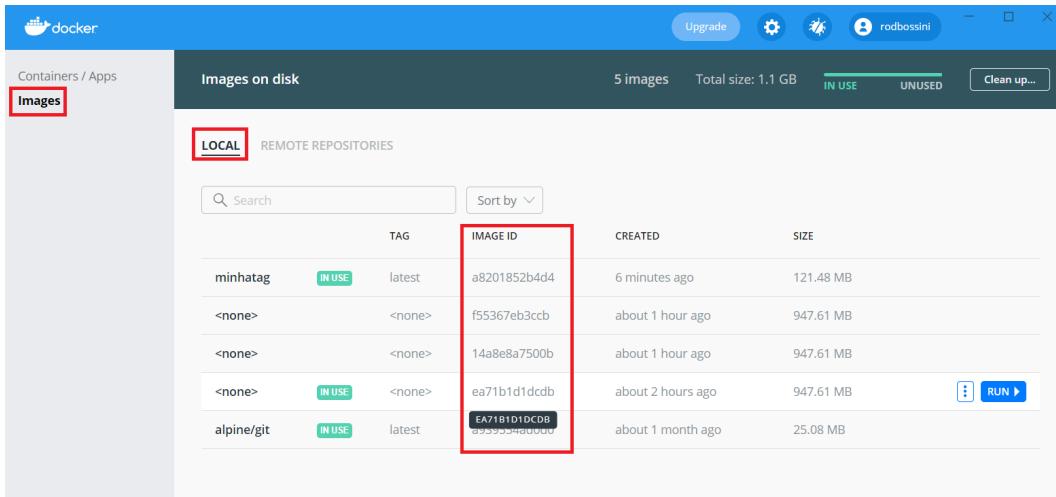
docker build .

Se desejar, use

docker build -t mss-lembretes .

para associar a tag **mss-lembretes** à imagem. Lembre-se de pegar o seu id assim que a execução do comando terminar. Também é possível pegar o id na interface gráfica do Docker, que pode ser aberta clicando-se no ícone na “system tray” do sistema operacional. Veja a Figura 4.3.43.

Figura 4.3.43



A seguir, coloque um contêiner em execução com

docker run -p 4000:4000 id_da_imagem

4.3.59 Testando o microsserviço de lembretes com Neste instante, já é possível enviar requisições ao microsserviço de lembretes por meio de **http://localhost:4000/lembretes**. Entretanto, note que os demais microsserviços não estão em execução. Isso quer dizer que um erro será gerado. Faça uma requisição com o Postman e use

docker logs id_do_conteiner

para verificar as mensagens de erro. Você também pode usar

docker ps

para pegar o id de cada contêiner em execução e então usar

docker logs id_do_conteiner

para cada um deles.

4.3.60 Implantando os demais microsserviços com Repita o processo para colocar os demais microsserviços em execução usando o . Os arquivos *Dockerfile* e *.dockerignore* são iguais. Basta copiá-los para as pastas de cada microsserviço. A seguir, use

docker build -t mss-observacoes .

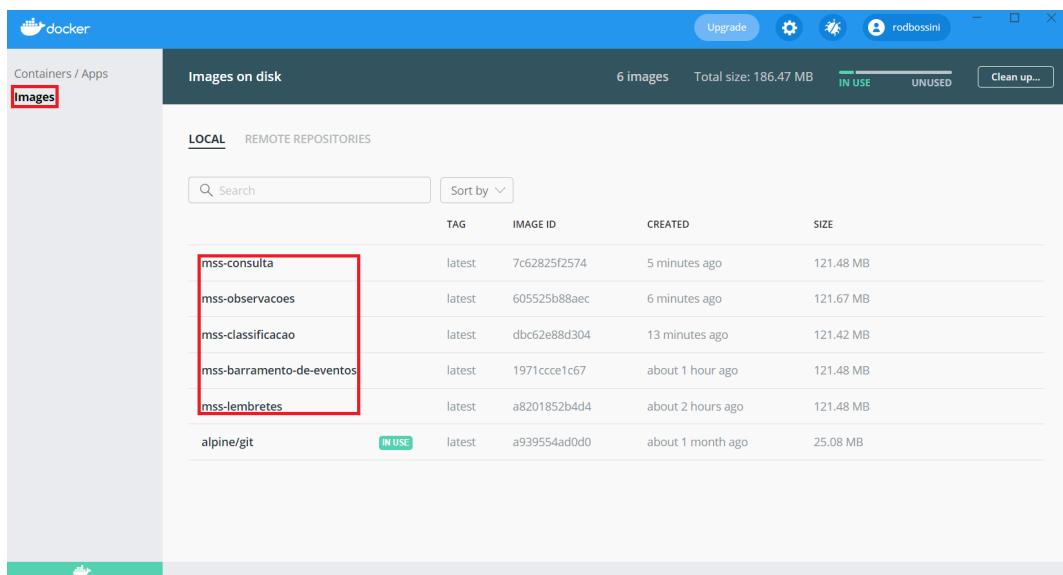
docker build -t mss-consulta .

docker build -t mss-classificacao .

docker build -t mss-barramento-de-eventos .

Assim você terá construído uma imagem para cada microsserviço. Na interface gráfica do  , o resultado deve ser parecido com aquele exibido pela Figura 4.3.44.

Figura 4.3.44



4.3.61 Testando os demais microsserviços com Use

```
//importante subir o barramento antes dos demais
//pois ele é consultado por eles
docker run -p 10000:10000 mss-barramento-de-eventos
```

docker run -p 4000:4000 mss-lembretes

docker run -p 5000:5000 mss-observacoes

docker run -p 6000:6000 mss-consulta

docker run -p 7000:7000 mss-classificacao

para colocar todos os microsserviços em execução, cada qual em um contêiner separado. Abra o Postman e realize alguns testes.

4.3.62  **kubernetes** O uso de contêineres traz as diversas vantagens mencionadas. Entretanto, gerenciar a sua execução, em particular em um ambiente distribuído em que a demanda por processamento varia significativamente, pode se tornar bastante trabalhoso. É aí que entra o uso do  **kubernetes**. Segundo a documentação oficial, que pode ser acessada por meio do Link 4.3.9, o  **kubernetes** é uma plataforma para o gerenciamento de serviços que facilita a configuração declarativa e a automação.

Link 4.3.9

<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

Nota. O nome  **kubernetes** tem origem grega e significa algo como **timoneiro, homem do leme, tripulante responsável pela navegação**. A ideia é defini-lo como o **responsável pela condução de uma embarcação de contêineres**.

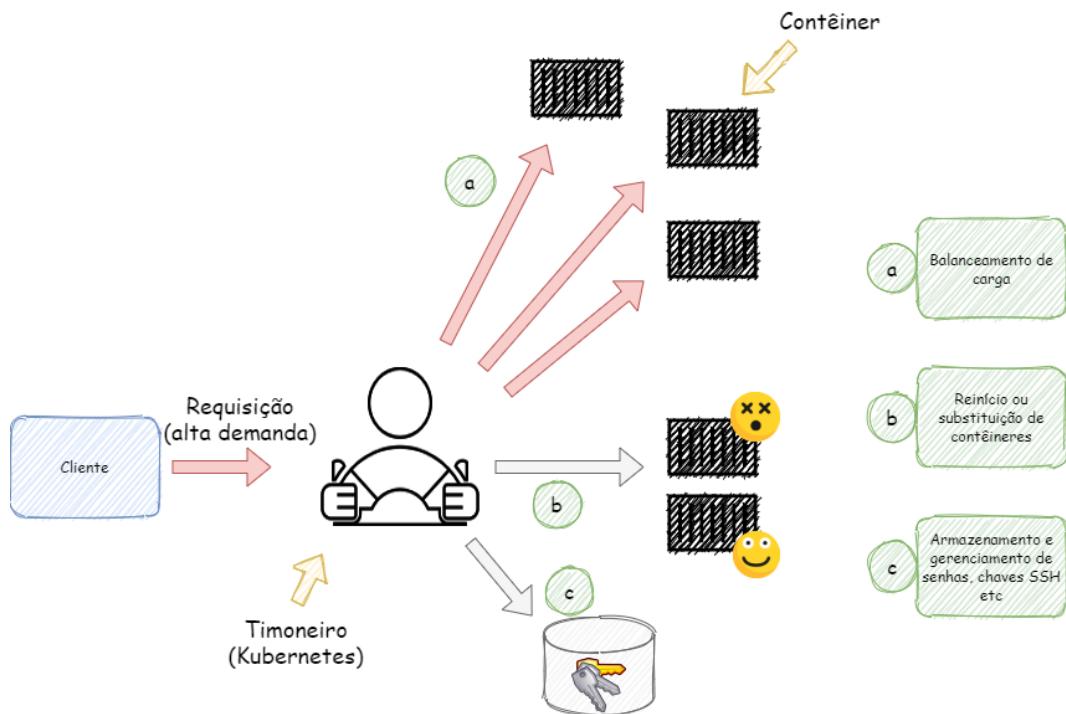
Nota. O  **kubernetes** foi criado pelo Google. Hoje ele é mantido pela **Cloud Native Computing Foundation**. Acesse a sua página oficial por meio do Link 4.3.10.

Link 4.3.10

<https://www.cncf.io/>

A Figura 4.3.45 ilustra algumas das funcionalidades providas pelo  **kubernetes**.

Figura 4.3.45



Quando colocamos o **kubernetes** em execução, obtemos um **cluster**. Um cluster é um conjunto de **máquinas trabalhadoras** - do inglês *worker machines* - que executam aplicações em contêineres. Todo cluster possui, pelo menos, uma máquina trabalhadora.

4.3.63 Arquitetura do **kubernetes**

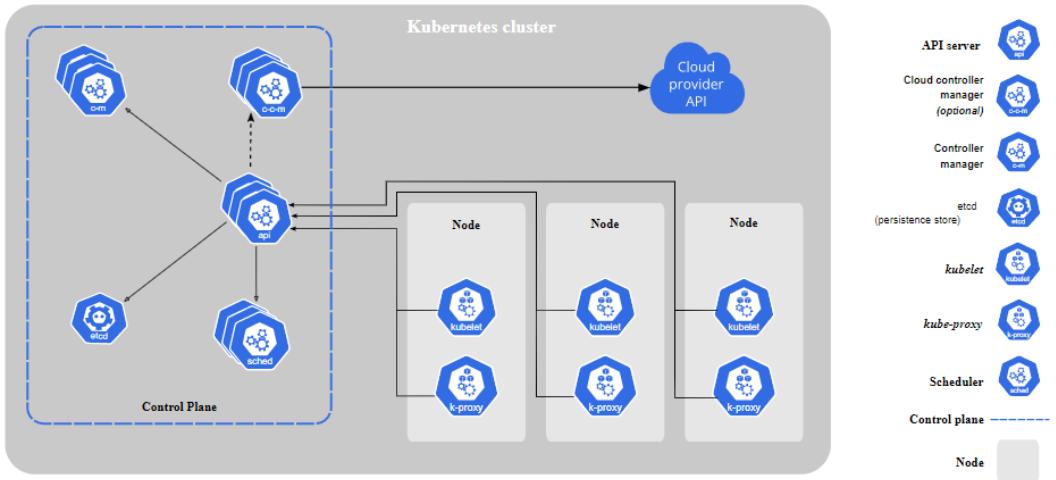
A arquitetura do **kubernetes** é composta por

- Uma **máquina trabalhadora** que executa**Pods**. Máquinas trabalhadoras são também chamadas de **nós**. Elas podem ser máquinas físicas ou virtuais.
- Um **Pod** é um conjunto de contêineres em execução.
- O **control plane** é a camada de orquestração que expõe uma API por meio da qual é possível implantar e gerenciar contêineres. Em ambiente de produção, a sua execução comumente é realizada por vários computadores, o que traz tolerância a falhas e alta disponibilidade.

Veja a Figura 4.3.46.

Figura 4.3.46:

<https://kubernetes.io/docs/concepts/overview/components/>



4.3.64 Detalhes sobre o control plane

O control plane é constituído por

- **kube-apiserver** Desempenha o papel de front end do control-plane expondo a API do **kubernetes**.
- **etcd** Mecanismo de armazenamento baseado em pares chave/valor. Usado para armazenar dados referentes ao funcionamento do cluster.
- **kube-scheduler** Responsável por detectar novos Pods para os quais um nó ainda não foi alocado e providenciar a alocação.
- **kube-controller-manager** Executa **controllers**. Um controller verifica constantemente o estado do cluster e executa ações necessárias para alterar seu estado atual, levando-o para o estado desejado. Um **exemplo** de controller é o **Node controller**. Ele é responsável por informar quando um nó se torna inoperante.
- **cloud-controller-manager** É um tipo de control plane que viabiliza a conexão entre o cluster do **kubernetes** e um provedor de computação em nuvem. Ele separa os componentes que interagem com o provedor de nuvem daqueles que interagem somente com o cluster local.

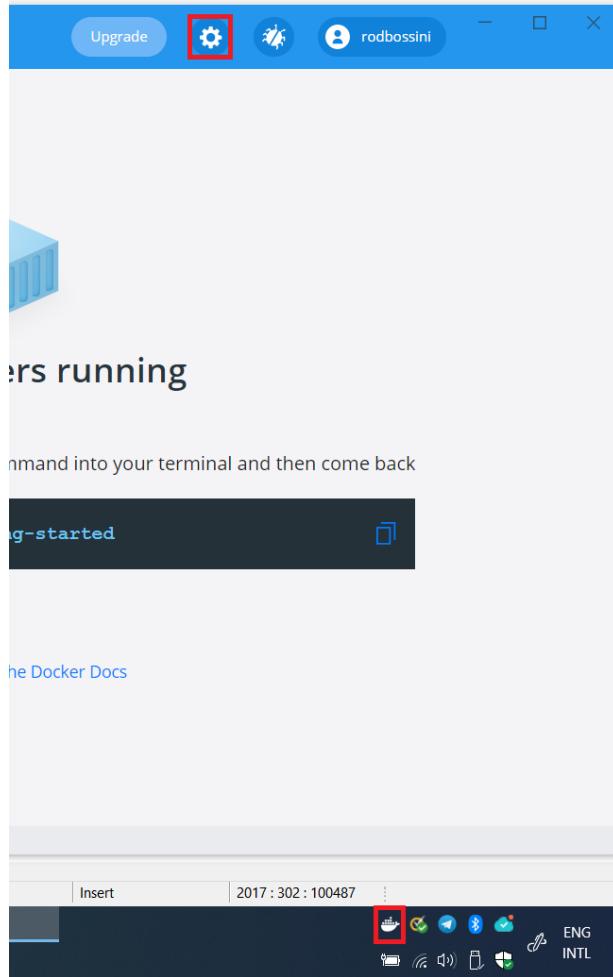
4.3.65 Detalhes sobre os nós Um nó ou máquina virtual, por sua vez, possui os seguintes componentes.

- **kubelet** Responsável por garantir que os contêineres estão em execução em um Pod, de acordo com especificações obtidas em **PodSpecs**. Um kubelet não se preocupa com contêineres em execução que eventualmente não tenham sido criados pelo  **kubernetes**.
- **kube-proxy** Implementa a ideia de **Service** do  **kubernetes**. Trata-se de uma maneira de expor um conjunto de Pods como um serviço em rede.
- **Container runtime** É o software responsável por executar os contêineres. Hoje o  **docker** é certamente o mais utilizado. Entretanto, qualquer um que implemente a especificação **CRI**⁴ pode ser utilizado.

4.3.66 Instalação do  kubernetes : Windows A instalação do  **kubernetes** varia em função do sistema operacional utilizado. Para o **Windows**, basta instalar o **Docker Desktop**. Ele inclui uma implementação do  **kubernetes** que pode ser executada localmente. É preciso habilitar o uso do  **kubernetes** por meio da interface gráfica do  **docker**. Clique no ícone que fica na “system tray” do sistema operacional e então, clicar no botão “Settings, como na Figura 4.3.47.

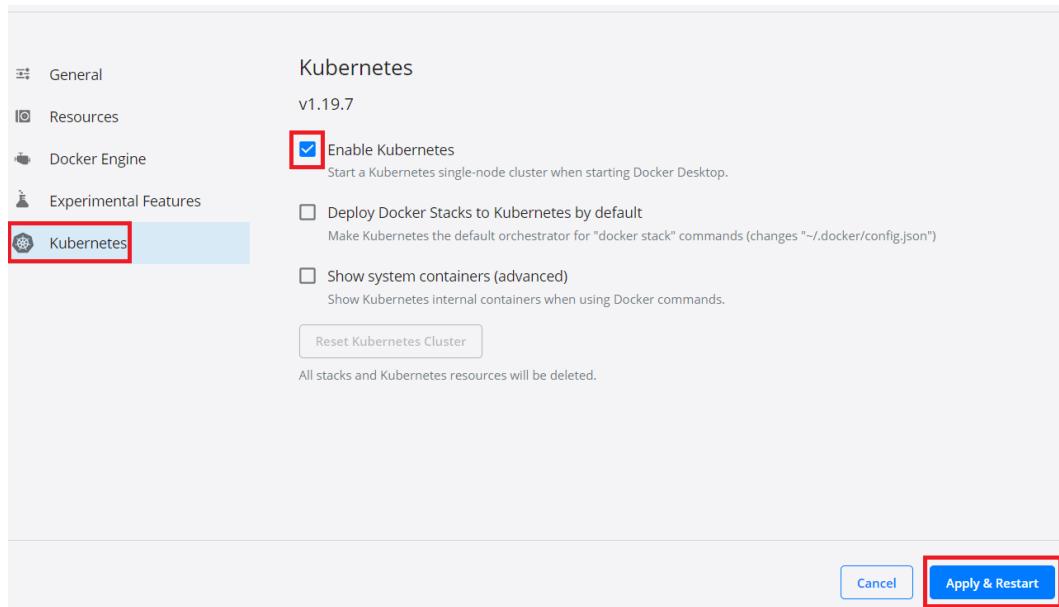
⁴<https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md>

Figura 4.3.47



A seguir, clique na opção **Kubernetes**, marque a caixa **Enable Kubernetes** e clique **Apply & Restart**, como na Figura 4.3.48.

Figura 4.3.48



O download da implementação do **kubernetes** será iniciado e o Docker Desktop será reiniciado quando a instalação estiver concluída.

4.3.67 Instalação do kubernetes : Linux & MacOS Caso esteja utilizando **Linux** ou **MacOS**, uma opção é fazer a instalação do **minikube**. O procedimento é muito simples. Há pacotes oficiais apropriados para os dois sistemas operacionais. Siga as instruções disponíveis no Link 4.3.11.

Link 4.3.11
<https://minikube.sigs.k8s.io/docs/start/>

4.3.68 Hello, kubernetes ! Nesta seção utilizaremos diversos comandos do **kubernetes** a fim de ilustrar o seu uso básico.

4.3.68.1 Checando versões Use

kubectl version

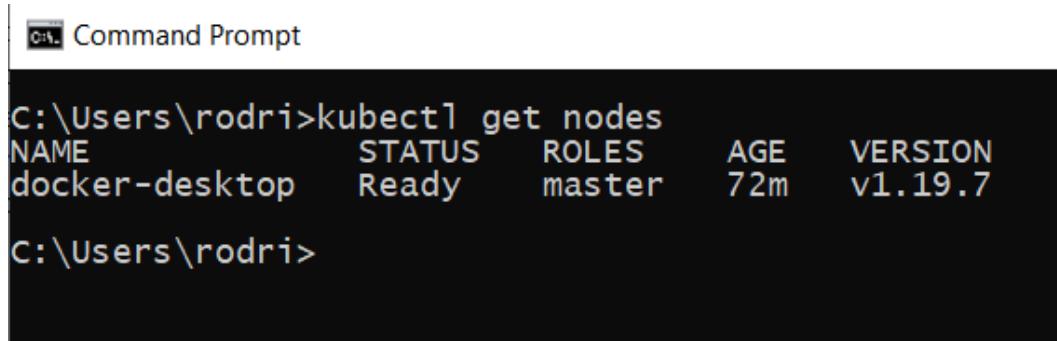
para verificar a versão do cliente e do servidor. O cliente é o próprio **kubectl**. Ele é um cliente de linha de comando que permite acesso a clusters **kubernetes**. O servidor, por outro lado, é o cluster **kubernetes**.

4.3.68.2 Visualizando máquinas trabalhadoras (nós) Execute

kubectl get nodes

para exibir os nós (máquinas trabalhadoras) disponíveis no cluster. Caso seja a sua primeira instalação do  **kubernetes**, provavelmente você verá um único nó, como mostra a Figura 4.3.49.

Figura 4.3.49



```
C:\Users\rodri>kubectl get nodes
NAME           STATUS    ROLES      AGE     VERSION
docker-desktop   Ready     master    72m     v1.19.7
C:\Users\rodri>
```

4.3.68.3 Forma geral de uso do **kubectl**

Digite apenas

kubectl

para ver as formas como o **kubectl** pode ser utilizado. A forma geral é

kubectl ação recurso

Para obter ajuda sobre um comando específico, use

kubectl get nodes --help

4.3.68.4 Criando um Deployment

A fim de implantar uma aplicação usando o  **kubernetes**, criamos uma **configuração de Deployment** que o **instrui** sobre **como criar e atualizar instâncias da aplicação**. Uma vez criado um *Deployment*, o *control plane* produz instâncias da aplicação que serão executadas em diferentes *nós*. Um *controller* de *Deployment* monitora cada instância. Se algum *nó* ficar inoperante e comprometer o funcionamento de uma instância da aplicação, esse *controller* se encarrega de colocar em execução uma nova instância dela em outro *nó* disponível. Use

```
kubectl create deployment meu-primeiro-deployment  
--image=gcr.io/google-samples/kubernetes-bootcamp:v1
```

para criar seu primeiro *Deployment*.

Nota. gcr.io/google-samples/kubernetes-bootcamp:v1 é uma imagem que possui um servidor web simples que simplesmente devolve a requisição que recebeu.

A seguir, verifique quais *Deployments* você possui com

```
kubectl get deployments
```

O resultado deve ser parecido com aquele exibido pela Figura 4.3.50.

Figura 4.3.50

```
C:\Users\rodri>kubectl get deployments  
NAME READY UP-TO-DATE AVAILABLE AGE  
meu-primeiro-deployment 1/1 1 1 7m9s
```

4.3.68.5 Proxy para acesso aos *Pods* A aplicação está em execução em um contêiner   . Cada contêiner sob controle do  executa dentro de um *Pod*. Por padrão, *Pods* executam em uma rede privada, isolada do mundo externo. O `kubectl` pode criar um **proxy** entre a rede interna do cluster e o mundo externo com

```
kubectl proxy
```

O resultado deve ser parecido com a Figura 4.3.51.

Figura 4.3.51

```
C:\Users\rodri>kubectl get deployments  
NAME READY UP-TO-DATE AVAILABLE AGE  
meu-primeiro-deployment 1/1 1 1 5h32m  
C:\Users\rodri>kubectl proxy  
Starting to serve on 127.0.0.1:8001
```



Abra o Postman e faça uma requisição na porta exibida pela Figura 4.3.51. A saída deve ser parecida com a Figura 4.3.52.

Figura 4.3.52

The screenshot shows the Postman application interface. At the top, there is a red box around the 'GET' method and another red box around the URL 'localhost:8001'. Below the header, there are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Headers (6)' tab is currently selected. Under 'Query Params', there is a table with one row containing 'Key' and 'Value'. The 'Body' tab is selected, showing a JSON response with a list of API endpoints. The JSON code is as follows:

```
1
2     "paths": [
3         "/api",
4         "/api/v1",
5         "/apis",
6         "/apis/",
7         "/apis/admissionregistration.k8s.io",
8         "/apis/admissionregistration.k8s.io/v1",
9         "/apis/admissionregistration.k8s.io/v1beta1",
10        "/apis/apiextensions.k8s.io",
11        "/apis/apiextensions.k8s.io/v1",
12        "/apis/apiextensions.k8s.io/v1beta1",
13        "/apis/apiregistration.k8s.io",
14        "/apis/apiregistration.k8s.io/v1",
15        "/apis/apiregistration.k8s.io/v1beta1",
16        "/apis/apps",
17        "/apis/apps/v1",
18        "/apis/authentication.k8s.io",
19        "/apis/authentication.k8s.io/v1",
20        "/apis/authentication.k8s.io/v1beta1",
21        "/apis/authorization.k8s.io",
22        "/apis/authorization.k8s.io/v1",
23        "/apis/authorization.k8s.io/v1beta1",
24        "/apis/autoscaling",
25        "/apis/autoscaling/v1",
26        "/apis/autoscaling/v2beta1",
27        "/apis/autoscaling/v2beta2",
28        "/apis/batch",
29        "/apis/batch/v1",
30        "/apis/batch/v1beta1",
31        "/apis/certificates.k8s.io",
```

Cada linha do resultado é um endpoint para o qual requisições podem ser

direcionadas. O  **kubernetes** gera um endpoint para cada Pod com base em seu nome.

4.3.68.6 Nomes de *Pods* com Go templates Vamos extrair o nome do Pod existente e guardar em uma variável de ambiente para uso futuro. O comando `kubectl get pods` traz uma quantidade bastante grande de informações sobre os pods. Use

`kubectl get pods -o json (ou yaml, se preferir)`

para visualizar todos os campos disponíveis. A saída deve ser parecida com o que exibe a Figura 4.3.53.

Figura 4.3.53

```
{
  "apiVersion": "v1",
  "items": [
    {
      "apiVersion": "v1",
      "kind": "Pod",
      "metadata": {
        "creationTimestamp": "2021-04-17T06:48:20Z",
        "generateName": "meu-primeiro-deployment-7cd85b47b7-",
        "labels": {
          "app": "meu-primeiro-deployment",
          "pod-template-hash": "7cd85b47b7"
        },
        "managedFields": [
          {
            "apiVersion": "v1",
            "fieldsType": "FieldsV1",
            "fieldsV1": {
              "f:metadata": {
                "f:generateName": {},
                "f:labels": {
                  ".": {},
                  "f:app": {},
                  "f:pod-template-hash": {}
                }
              },
              "f:ownerReferences": {
                ".": {},
                "k:{\"uid\":\"bb1ff9a3-bfb1-4d5d-b86c-5c312dc7e1f3\)": {
                  ".": {},
                  "f:apiVersion": {},
                  "f:blockOwnerDeletion": {},
                  "f:controller": {},
                  "f:kind": {},
                  "f:name": {},
                  "f:uid": {}
                }
              }
            }
          },
          "f:spec": {
            "f:containers": {
              "k:{\"name\": \"kubernetes-bootcamp\)": {
                ".": {},
                "f:image": {},
                "f:imagePullPolicy": {},
                "f:name": {}
              }
            }
          }
        ]
      }
    }
  ]
}
```

Estude a estrutura do JSON e verifique que ele possui uma coleção associada à chave `items`. Cada item na lista é um *Pod*. Cada um deles possui um objeto JSON associado a uma chave chamada `metadata`. Entre muitas chaves, o objeto em questão possui uma chamada `name`. Ela está associada ao nome de cada *Pod*. Para extrair essa e outras informações de interesse em um formato específico, podemos utilizar um **Go template**. Veja a sua documentação oficial no Link 4.3.12.

Link 4.3.12
<https://golang.org/pkg/text/template/>

Antes de prosseguir, vejamos alguns exemplos de uso de *Go templates* envolvendo a saída do `kubectl`.

- `kubectl get pods -o go-template='Hello, Go templates!'` - Ignora o que o `kubectl` produz e apenas mostra o texto especificado.
- `kubectl get pods -o go-template='{{.apiVersion}}'` - extrai o valor associado à chave `apiVersion`, existente na saída produzida pelo `kubectl`. O caractere `.` que precede o nome da propriedade é fundamental. Se ele for omitido, `apiVersion` será considerada uma tentativa de chamada de função. Note que `{{}}` é usado para avaliar expressões.
- `kubectl get pods -o go-template='{{.items}}'` - extraí os dados de todos os itens de uma só vez.
- `kubectl get pods -o go-template='{{range .items}}{{.metadata}}{{end}}'` - usa a função `range` para iterar sobre a coleção `items`. Exibe o objeto associado à chave `metadata` de cada Pod.
- `kubectl get pods -o go-template='{{range .items}}{{.metadata.name}}{{end}}'` - Exibe o nome de cada Pod. `{{end}}` encerra a repetição criada por `range`.

Como temos um único *Pod* no momento, vamos usar

```
kubectl get pods -o go-template= '{{range
.items}}{{.metadata.name}}{{end}}
set POD_NAME=nome_obtido_aqui //Windows
export POD_NAME=nome_obtido_aqui //Unix-like
echo %POD_NAME% //Windows, para testar
echo $POD_NAME //Unix-like, para testar
```

para guardar o seu nome em uma variável de ambiente.

4.3.68.7 Enviando requisições a um *Pod* Execute

```
curl  
http://localhost:8001/api/v1/namespaces/default/pods/%POD_-  
NAME% //Windows  
curl  
http://localhost:8001/api/v1/namespaces/default/pods/$POD_-  
NAME //Unix-like
```

para obter detalhes do *Pod*. Também é possível fazer essa requisição usando o Postman ou o navegador.

4.3.68.8 Logs de um *Pod* Use

```
kubectl logs %POD_NAME% //Windows  
kubectl logs $POD_NAME //Unix-like
```

para visualizar os logs do seu *Pod*.

4.3.68.9 Executando comandos “dentro” de um contêiner Podemos executar comandos “dentro” de um contêiner com

```
//Lista conteúdo no diretório atual  
kubectl exec %POD_NAME% -- ls //Windows  
kubectl exec $POD_NAME -- ls //Unix-like  
// Exibe as variáveis de ambiente  
kubectl exec %POD_NAME% -- env //Windows  
kubectl exec $POD_NAME -- env //Unix-like  
// Obtém a data  
kubectl exec %POD_NAME% -- date //Windows  
kubectl exec $POD_NAME -- date //Unix-like
```

Nota. Um *Pod* executa múltiplos contêineres. Entretanto, não há especificação sobre qual contêiner utilizar para cada comando dos exemplos. Isso ocorre pois o *Pod* que estamos usando possui um único contêiner em execução.

4.3.68.10 A aplicação implantada - Código-fonte e teste

A aplicação que implantamos está definida em um arquivo chamado *server.js*. Para visualizar o seu conteúdo e testá-la a partir do próprio contêiner em que se encontra, vamos abrir um terminal vinculado ao contêiner. faça isso com

```
kubectl exec -ti %POD_NAME% -- bash //Windows  
kubectl exec -ti $POD_NAME -- bash //Unix-like
```

Nota. As flags `i` e `t` usadas pelo comando `exec` fazem com que

- `i` - a entrada padrão do computador que estamos usando seja direcionada para o contêiner.
- `t` - o terminal que estamos abrindo no contêiner seja um `tty`. Assim, ele pode receber e enviar mensagens tal qual um **Teletypewriter**.

Resumidamente, em conjunto, essas flags viabilizam a comunicação entre o host e o contêiner por meio do terminal.

Nota. Veja mais sobre o conceito de `tty` no Link 4.3.13.

Link 4.3.13
<http://www.linusakesson.net/programming/tty/>

Para visualizar o conteúdo do arquivo `server.js`, no terminal vinculado ao contêiner, use

```
cat server.js
```

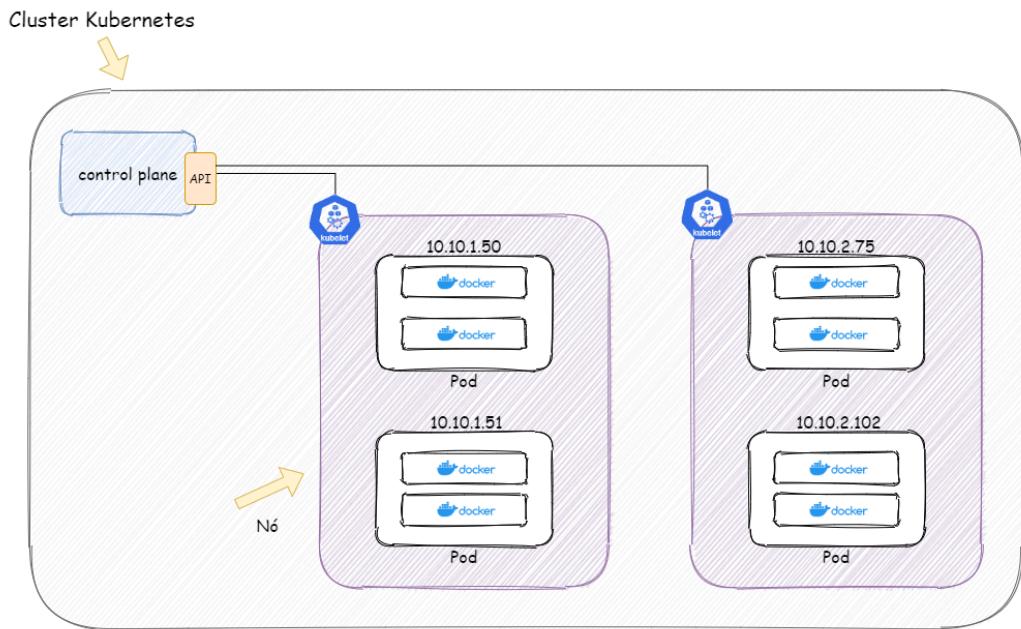
Verifique que o servidor está em funcionamento com

```
curl localhost:8080
```

4.3.68.11 Serviços

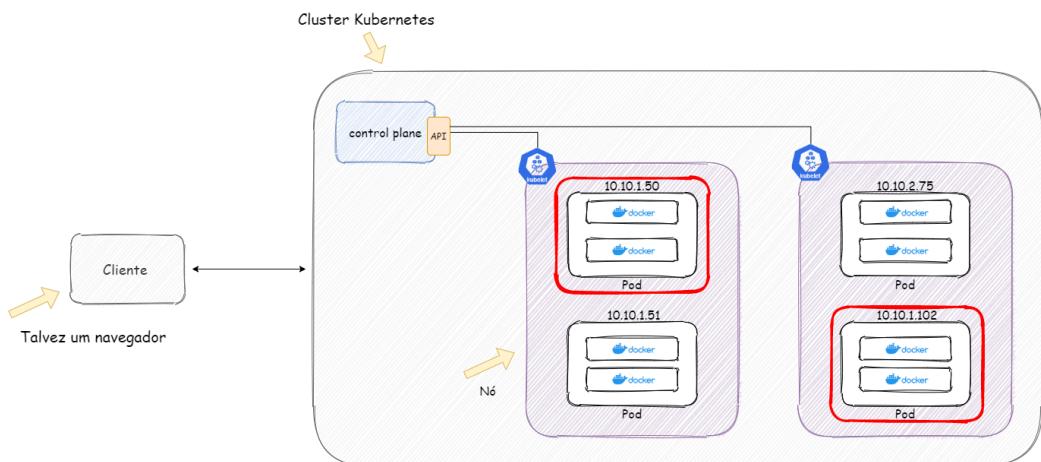
Considere a arquitetura retratada pela Figura 4.3.54.

Figura 4.3.54



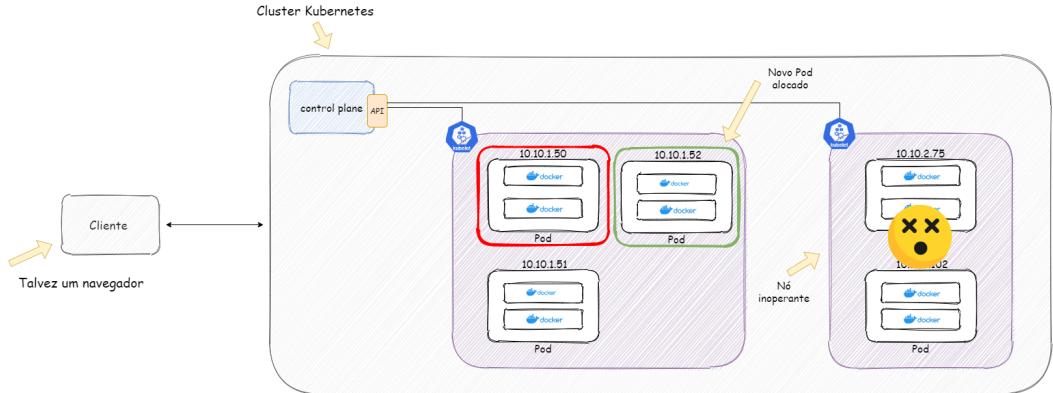
Ela destaca um detalhe muito importante. *Pods* possuem IP próprio, mesmo aqueles executando no mesmo *nó*. Agora, suponha que uma aplicação deseja utilizar uma funcionalidade disponibilizada por contêineres executando no cluster. É possível que mais de um *Pod* esteja envolvido no atendimento a requisições feitas por ela. Eles são destacados na Figura 4.3.55.

Figura 4.3.55



Quando um *nó* se torna inoperante, cabe ao  **kubernetes** colocar em execução novas instâncias dos *Pods* que estavam sob sua responsabilidade. Veja a Figura 4.3.56.

Figura 4.3.56



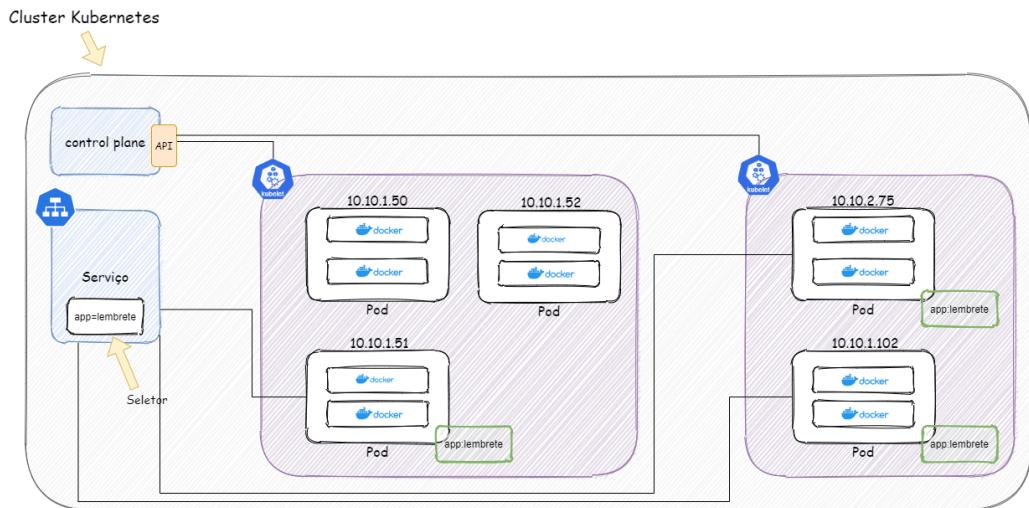
É claro que esse tipo de alteração deve ser transparente para a aplicação cliente. Ela sequer deve saber que está sendo atendida por múltiplos *Pods* no cluster. É aí que entra o conceito de **serviço**. Um *serviço* é um agrupamento lógico de *Pods*. Quando um serviço é criado, ele pode ter um **tipo**. Cada tipo influencia a forma como o serviço é exposto para acesso a seus *Pods*.

Nota. Um serviço fornece uma forma simples para acesso a todos os seus **Pods**, abstraindo detalhes como Ips e portas diferentes.

- **ClusterIP** - O serviço é exposto somente dentro do próprio cluster.
- **NodePort** - Expõe o serviço externamente ao cluster usando a mesma porta de cada *nó* especificado. O padrão para acesso é *NodeIp:NodePort*.
- **LoadBalancer** - Cria um balanceador de carga e atribui a ele um único IP externo e fixo.

Um serviço agrupa **Pods** por meio de **rótulos e seletores**. Aplicamos rótulos a *Pods* que desejamos que façam parte daquele serviço e, na definição do serviço, especificamos aquele rótulo escolhido como seu seletor. Veja a Figura 4.3.57.

Figura 4.3.57



Comece verificando os serviços existentes com

kubectl get services

A saída deve conter um único serviço, como mostra a Figura 4.3.58. Ele é criado por padrão pelo **kubernetes**.

Figura 4.3.58

```
C:\Users\rodri>kubectl get services
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP      8d
```

Crie um serviço com

**kubectl expose deployment/meu-primeiro-deployment
--type="NodePort" --port 8080**

Execute

kubectl get services

novamente. Para verificar a porta do nó - campo **NodePort** - que foi exposta, use

kubectl describe services/meu-primeiro-deployment

Vamos armazená-la em uma variável de ambiente para uso futuro. Para obtê-la, use

```
kubectl get services/meu-primeiro-deployment -o  
go-template="{{(index .spec.ports 0).nodePort}}"  
set NODE_PORT=porta_obtida //Windows  
export NODE_PORT=porta_obtida //Unix-like  
echo %NODE_PORT% //Windows, para testar  
echo $NODE_PORT //Unix-like, para testar
```

Obtenha o endereço IP da máquina com

```
ipconfig //Windows  
ifconfig //Unix-like
```

Nota. Caso esteja utilizando o  `minikube`, use `minikube ip` para obter o endereço do seu cluster.

Faça um teste com

```
curl ip_da_sua_maquina:%NODE_PORT% //Windows  
curl ip_da_sua_maquina:$NODE_PORT //Unix-like
```

4.3.68.12 Manipulação de rótulos Nosso *Pod* possui um rótulo que foi atribuído automaticamente quando criamos o *deployment*. Use

```
kubectl describe deployment
```

para visualizá-lo. Veja a Figura 4.3.59.

Figura 4.3.59

```
C:\Users\rodri>kubectl describe deployment
Name:               meu-primeiro-deployment
Namespace:          default
CreationTimestamp:  Sat, 17 Apr 2021 21:42:41 -0300
Labels:             app=meu-primeiro-deployment
Annotations:        deployment.kubernetes.io/revision: 1
Selector:           app=meu-primeiro-deployment
Replicas:           1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=meu-primeiro-deployment
  Containers:
    kubernetes-bootcamp:
      Image:      gcr.io/google-samples/kubernetes-bootcamp:v1
      Port:       <none>
      Host Port: <none>
      Environment: <none>
      Mounts:    <none>
      Volumes:   <none>
  Conditions:
    Type     Status  Reason
    ----     ----   -----
    Available  True    MinimumReplicasAvailable
    Progressing  True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   meu-primeiro-deployment-7cd85b47b7 (1/1 replicas created)
Events:
  Type     Reason          Age   From            Message
  ----     -----          ---   ---            -----
```

Podemos fazer uma busca por *Pods* e por *serviços* filtrando por rótulo. Para isso, use

```
kubectl get pods -l chave=valor
kubectl get services -l chave=valor
```

Armazene o nome do *Pod* obtido com

```
kubectl get pods -o go-template --template "{{range
  .items}}{{.metadata.name}}{{end}}
set POD_NAME=nome_aqui //Windows
export POD_NAME=nome_aqui //Unix-like
```

Aplique um rótulo ao *Pod* com

```
kubectl label pod %POD_NAME% versao=v1 //Windows
kubectl label pod $POD_NAME //Unix-like
```

Verifique que o novo rótulo foi aplicado com

```
kubectl describe pods %POD_NAME%
```

A Figura 4.3.60 exibe parte do resultado esperado.

Figura 4.3.60

```
C:\Users\rodri>kubectl describe pods %POD_NAME%
Name:      meu-primeiro-deployment-7cd85b47b7-rvpq4
Namespace:  default
Priority:   0
Node:       docker-desktop/192.168.65.4
Start Time: Sat, 17 Apr 2021 21:42:41 -0300
Labels:     app=meu-primeiro-deployment
            pod-template-hash=7cd85b47b7
            versao=v1
Annotations: <none>
Status:     Running
IP:        10.1.0.63
 IPs:
   IP:      10.1.0.63
Controlled By: ReplicaSet/meu-primeiro-deployment-7cd85b47b7
Containers:
```

A busca pode ser realizada usando qualquer rótulo existente no objeto de interesse. Use, por exemplo

```
kubectl get pods -l versao=v1
```

4.3.68.13 Removendo serviços Uma vez que não seja necessária a exposição causada por um serviço, ele pode ser removido. A sua remoção não implica na remoção dos *Pods* associados a ele. A remoção pode ser feita com

```
kubectl delete service -l app=meu-primeiro-deployment
```

Verifique que o serviço já não está disponível com

```
kubectl get services
```

Verifique também que não é mais possível realizar requisições usando a porta que havia sido exposta com

```
curl ip_da_sua_maquina:%NODE_PORT% //Windows
curl ip_da_sua_maquina:$NODE_PORT //Unix-like
```

Observe, entretanto, que a aplicação permanece executando dentro do *Pod*. Use

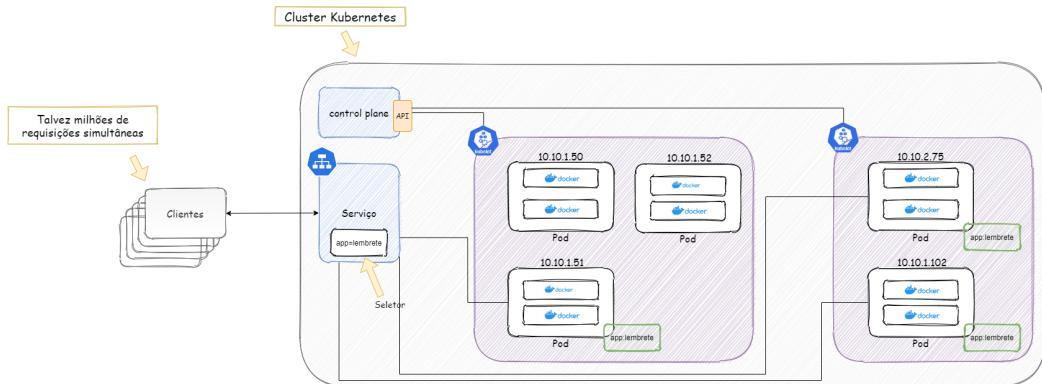
```
kubectl exec -ti %POD_NAME% -- curl localhost:8080  
Windows
```

```
kubectl exec -ti $POD_NAME -- curl localhost:8080  
Unix-like
```

para verificar. Ocorre que o deployment está sob gerência do  **kubernetes** e ele garante uma instância da aplicação em execução. Para removê-la, seria necessário remover também o *deployment*.

4.3.68.14 Escalabilidade Quando a demanda a funcionalidades de sua aplicação aumenta, é de interesse que novos recursos sejam alocados para mantê-la operando e com tempo de resposta aceitável. No  **kubernetes**, a **escalabilidade** é obtida aumentando-se o número de **rélicas** especificadas no *deployment*. Quando requisições são atendidas por múltiplas instâncias da aplicação, é preciso distribuir a carga de trabalho de alguma forma. Os serviços possuem um **balanceador de carga** embutido que se encarrega de fazê-lo. Serviços também se encarregam de **verificar esporadicamente se cada um de seus Pods está operando**, garantindo que requisições sejam atendidas somente por aqueles que estejam. Veja a Figura 4.3.61.

Figura 4.3.61



Obtenha a sua lista de *deployments* com

```
kubectl get deployments
```

Cada *deployment* possui um **ReplicaSet**. O propósito de um *ReplicaSet* é viabilizar a replicação de *Pods*. Um *ReplicaSet* possui alguns campos, tais como

- um *seletor* usado para especificar quais *Pods* fazem parte dele
- um número de rélicas que indica quantos *Pods* devem ser mantidos em execução.

Use

kubectl get rs

para verificar seus *ReplicaSets*. As colunas **DESIRED** e **CURRENT** indicam quantas réplicas da aplicação são desejadas e quantas estão atualmente em funcionamento, respectivamente. Veja a Figura 4.3.62.

Figura 4.3.62

```
C:\Users\rodri>kubectl get rs
NAME                DESIRED   CURRENT   READY   AGE
meu-primeiro-deployment-7cd85b47b7    1         1         1      88m
```

Podemos aumentar o número de réplicas desejadas com

kubectl scale deployments/meu-primeiro-deployment --replicas=4

Verifique novamente a sua lista de *deployments* com

kubectl get deployments

Verifique também a sua lista de *Pods* com

```
kubectl get pods //simples
kubectl get pods -o wide //mais detalhes
```

A Figura 4.3.63 destaca que cada *Pod* tem seu próprio endereço IP.

Figura 4.3.63

```
C:\Users\rodri>kubectl get pods -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP           NODE   NOMINATED
DE   READINESS GATES
meu-primeiro-deployment-7cd85b47b7-b8fl1 1/1     Running   0          20m   10.1.0.65   docker-desktop   <none>
<none>
meu-primeiro-deployment-7cd85b47b7-jphsz 1/1     Running   0          20m   10.1.0.64   docker-desktop   <none>
<none>
meu-primeiro-deployment-7cd85b47b7-rvpq4 1/1     Running   0          113m  10.1.0.63   docker-desktop   <none>
<none>
meu-primeiro-deployment-7cd85b47b7-srtj6  1/1     Running   0          20m   10.1.0.66   docker-desktop   <none>
<none>
```

A descrição de objetos do  **kubernetes**, em geral, inclui os eventos que os envolveram ao longo do tempo. Ajustar o número de réplicas como fizemos é um evento que teve impacto no *deployment*. Visualize isso com

kubectl describe deployments/meu-primeiro-deployment

A Figura 4.3.64 destaca a lista de eventos.

Figura 4.3.64

```
C:\Users\rodri>kubectl describe deployments/meu-primeiro-deployment
Name:               meu-primeiro-deployment
Namespace:          default
CreationTimestamp:  Sat, 17 Apr 2021 21:42:41 -0300
Labels:             app=meu-primeiro-deployment
Annotations:        deployment.kubernetes.io/revision: 1
Selector:           app=meu-primeiro-deployment
Replicas:           4 desired | 4 updated | 4 total | 4 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=meu-primeiro-deployment
  Containers:
    kubernetes-bootcamp:
      Image:   gcr.io/google-samples/kubernetes-bootcamp:v1
      Port:    <none>
      Host Port: <none>
      Environment: <none>
      Mounts:  <none>
      Volumes: <none>
  Conditions:
    Type     Status  Reason
    ----  -----
    Progressing  True   NewReplicaSetAvailable
    Available   True   MinimumReplicasAvailable
OldReplicaSets: <none>
NewReplicaSet:  meu-primeiro-deployment-7cd85b47b7 (4/4 replicas created)
Events:
  Type     Reason            Age   From           Message
  Normal   ScalingReplicaSet  20h   deployment-controller  Scaled up replica set meu-primeiro-deployment-7cd85b47b7 to 4
```

4.3.68.15 Testando o balanceador de carga Quando uma requisição é enviada ao serviço, ele deve usar o seu balanceador de carga para distribuir a carga de trabalho entre os *Pods* que gerencia. Como removemos o serviço anterior, será necessário criar um novo para testar o balanceador de carga. Faça isso com

```
kubectl expose deployment/meu-primeiro-deployment
--type="NodePort" --port 8080
```

A seguir, obtenha a porta exposta com

```
kubectl get service meu-primeiro-deployment -o
go-template="{{(index .spec.ports 0).nodePort}}"
set NODE_PORT=porta //Windows
export NODE_PORT=porta //Unix-like
```

Execute

```
curl ip_da_sua_maquina:%NODE_PORT% //Windows
curl ip_da_sua_maquina:$NODE_PORT //Unix-like
```

diversas vezes e verifique o resultado. A Figura 4.3.65 destaca o nome do *nó* responsável pelo atendimento a cada requisição.

Figura 4.3.65

```
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-rvpq4 | v=1
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-srtj6 | v=1
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-srtj6 | v=1
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-b8f11 | v=1
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-b8f11 | v=1
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-b8f11 | v=1
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-b8f11 | v=1
C:\Users\rodri>curl 192.168.1.161:%NODE_PORT%
Hello Kubernetes bootcamp! | Running on: meu-primeiro-deployment-7cd85b47b7-jphsz | v=1
```

Para reduzir o número de réplicas, basta usar

kubectl scale deployments/meu-primeiro-deployment --replicas=2

E para testar, use

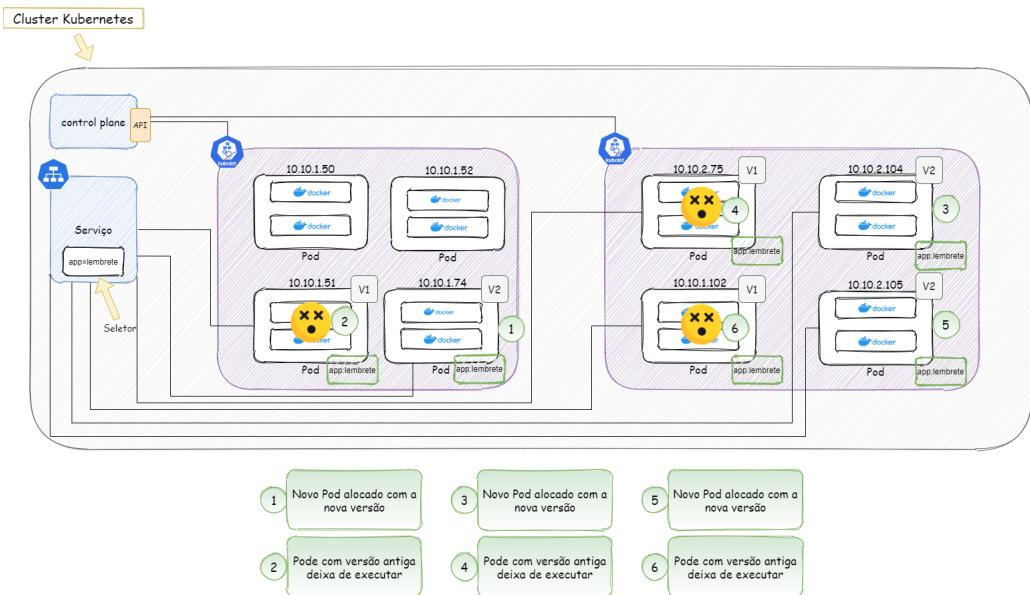
kubectl get deployments //simples
kubectl get deployments -o wide //Mais detalhes

Volte a utilizar quatro réplicas para os testes a seguir, com

kubectl scale deployments/meu-primeiro-deployment --replicas=4

4.3.68.16 Implantando novas versões da aplicação Nos dias atuais, é muito comum que desenvolvedores entreguem muitos **pequenos updates em um único dia** e que **muitos deles sejam implantados no mesmo dia**. Trata-se de uma prática comum de **DevOps**. Idealmente, a atualização acontece **sem que a aplicação passe por um período de indisponibilidade**. No  **kubernetes**, isso é obtido por meio de **Rolling Updates**. Um *rolling update* funciona substituindo gradualmente cada *Pod* existente por um novo. Enquanto um **Pod** é substituído, os demais permanecem atendendo as requisições dos clientes. Veja a Figura 4.3.66.

Figura 4.3.66



Há algumas considerações importantes ainda sobre os *rolling updates*.

- Por padrão, no máximo **um** *Pod* pode ficar indisponível durante um *rolling update*.
- Por padrão, no máximo **um** *Pod* pode ser criado por vez durante um **rolling update**.
- Ambos valores citados podem ser reconfigurados.
- O  **kubernetes** aplica uma versão (um identificador) automaticamente a cada atualização realizada.
- Uma vez que uma atualização tenha sido realizada, é possível voltar para qualquer uma que tenha sido implantada com sucesso no passado.

Para testar o funcionamento dos *rolling updates*, comece usando

kubectl get deployments

para visualizar os *deployments* que possui no momento. A seguir, visualize os seus *Pods* com

kubectl get pods

Visualize a imagem que está sendo executada pelos *Pods* com

kubectl describe pods

Para atualizar a imagem sendo executada pelos *Pods*, use

```
kubectl set image deployments/meu-primeiro-deployment
kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2
```

Nota. **kubernetes-bootcamp** é o identificador do contêiner existente na imagem que implantamos desde o início.

Logo depois, execute

kubectl get pods

diversas vezes. Deverá ser possível ver informações sobre a criação e remoção de *Pods*, como ilustra a Figura 4.3.67.

Figura 4.3.67

NAME	READY	STATUS	RESTARTS	AGE
meu-primeiro-deployment-69d7bc8bcc-jv8sd	1/1	Running	0	67s
meu-primeiro-deployment-69d7bc8bcc-qsxpz	1/1	Running	0	61s
meu-primeiro-deployment-856fdfb4d-vhrjq	0/1	ContainerCreating	0	6s
C:\Users\rodri>kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
meu-primeiro-deployment-69d7bc8bcc-jv8sd	1/1	Running	0	71s
meu-primeiro-deployment-69d7bc8bcc-qsxpz	1/1	Running	0	65s
meu-primeiro-deployment-856fdfb4d-vhrjq	0/1	ContainerCreating	0	10s
C:\Users\rodri>kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
meu-primeiro-deployment-69d7bc8bcc-jv8sd	1/1	Running	0	72s
meu-primeiro-deployment-69d7bc8bcc-qsxpz	1/1	Running	0	66s
meu-primeiro-deployment-856fdfb4d-vhrjq	0/1	ContainerCreating	0	11s
C:\Users\rodri>kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
meu-primeiro-deployment-69d7bc8bcc-jv8sd	1/1	Running	0	73s
meu-primeiro-deployment-69d7bc8bcc-qsxpz	1/1	Running	0	67s
meu-primeiro-deployment-856fdfb4d-vhrjq	0/1	ContainerCreating	0	12s
C:\Users\rodri>kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
meu-primeiro-deployment-69d7bc8bcc-jv8sd	1/1	Running	0	73s
meu-primeiro-deployment-69d7bc8bcc-qsxpz	1/1	Terminating	0	67s
meu-primeiro-deployment-856fdfb4d-vhrjq	1/1	Running	0	12s
meu-primeiro-deployment-856fdfb4d-zlmb2	0/1	ContainerCreating	0	0s
C:\Users\rodri>kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
meu-primeiro-deployment-69d7bc8bcc-jv8sd	1/1	Terminating	0	74s
meu-primeiro-deployment-69d7bc8bcc-qsxpz	1/1	Terminating	0	68s
meu-primeiro-deployment-856fdfb4d-vhrjq	1/1	Running	0	13s
meu-primeiro-deployment-856fdfb4d-zlmb2	1/1	Running	0	1s
C:\Users\rodri>kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
meu-primeiro-deployment-69d7bc8bcc-jv8sd	1/1	Terminating	0	75s
meu-primeiro-deployment-69d7bc8bcc-qsxpz	1/1	Terminating	0	69s
meu-primeiro-deployment-856fdfb4d-vhrjq	1/1	Running	0	14s
meu-primeiro-deployment-856fdfb4d-zlmb2	1/1	Running	0	2s

Para testar a atualização, use

```
curl ip_da_sua_maquina:%NODE_PORT% //Windows
curl ip_da_sua_maquina:$NODE_PORT //Unix-like
```

A Figura 4.3.68 destaca múltiplos *Pods* atendendo requisições. Perceba que todos eles estão usando a versão mais recente.

Figura 4.3.68

C:\Users\rodri>curl localhost:%NODE_PORT%	Hello Kubernetes bootcamp! Running on: meu-primeiro-deployment-69d7bc8bcc-ktvsc v=2
C:\Users\rodri>curl localhost:%NODE_PORT%	Hello Kubernetes bootcamp! Running on: meu-primeiro-deployment-69d7bc8bcc-th24d v=2
C:\Users\rodri>curl localhost:%NODE_PORT%	Hello Kubernetes bootcamp! Running on: meu-primeiro-deployment-69d7bc8bcc-ktvsc v=2
C:\Users\rodri>curl localhost:%NODE_PORT%	Hello Kubernetes bootcamp! Running on: meu-primeiro-deployment-69d7bc8bcc-ktvsc v=2

Também é possível verificar o status da atualização com

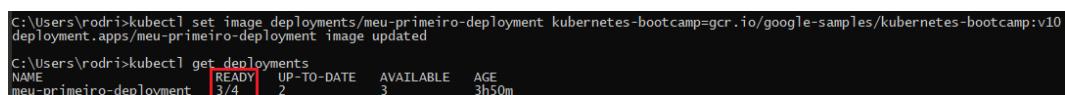
kubectl rollout status deployments/meu-primeiro-deployment

A seguir, vamos fazer uma atualização utilizando uma **imagem inexistente**. Assim, será necessário voltar à versão anterior. Para atualizar os *Pods* com a nova imagem, use

kubectl set image deployments/meu-primeiro-deployment kubernetes-bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10

Observe, na Figura 4.3.69, que o número de *Pods* prontos não é o total. Um deles ficou comprometido com a tentativa de atualização utilizando uma imagem inexistente.

Figura 4.3.69



C:\Users\rodri>kubectl set image deployments/meu-primeiro-deployment kubernetes-bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10
deployment.apps/meu-primeiro-deployment image updated
C:\Users\rodri>kubectl get deployments
NAME READY UP-TO-DATE AVAILABLE AGE
meu-primeiro-deployment 3/4 2 3 3h50m

Execute

kubectl get pods
kubectl describe pods

para visualizar mais informações sobre os *Pods* e entender o que houve. Podemos desfazer a última atualização realizada com

kubectl rollout undo deployments/meu-primeiro-deployment

4.3.69 Definindo objetos do  kubernetes com arquivos de configuração
Cada objeto ou recurso como *Pods* e *Deployments* que criamos até então pode também ser criado por meio da especificação de **arquivos de configuração**. Essa é a forma recomendada de fazê-lo por diferentes razões.

- Uma vez escritos, eles podem ser armazenados no sistema de controle de versão junto com o código-fonte. Conforme o sistema evolui, aquilo que é alocado para seu funcionamento também muda. Por isso, para cada versão, é importante saber quais objetos precisam ser alocados.
- Eles documentam aquilo que está sendo alocado no  **kubernetes** para o funcionamento da aplicação.

4.3.69.1 Arquivo de configuração para um *Pod* - Microsserviço de lembretes Vamos escrever um arquivo de configuração em que especificamos um *Pod* para execução do microsserviço de lembretes.

Abra um terminal vinculado ao diretório em que se encontra o *Dockerfile* do seu microsserviço de lembretes. Vamos criar uma nova imagem com uma nova tag com

```
docker build -t rodbossini/lembretes:0.0.1 .
```

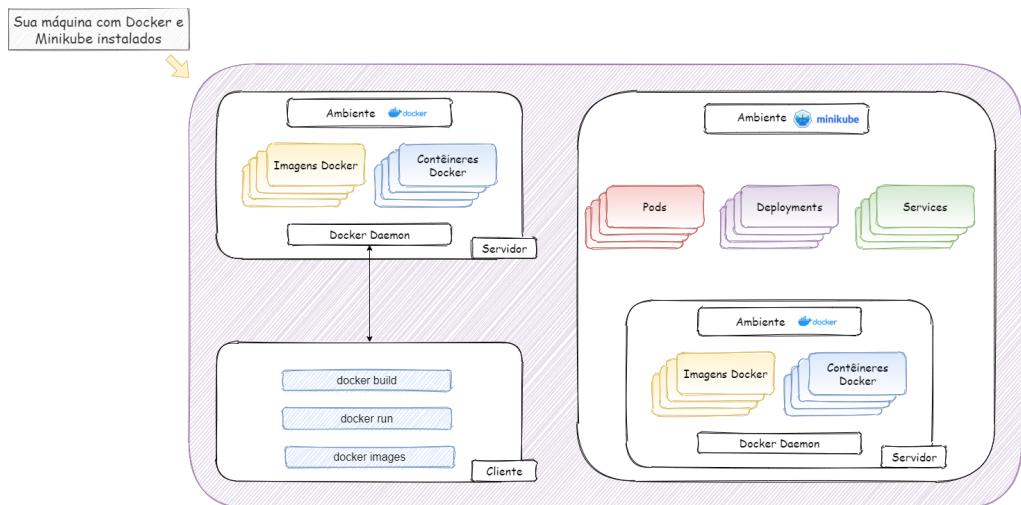
Nota. Use o padrão seu_id_no_docker_hub/nome_da_imagem:versao. Caso não tenha um id no Docker Hub, crie um no Link 4.3.14. É grátis.

Link 4.3.14
<https://hub.docker.com/>

Nota. O simples fato de criar uma imagem com *tag* no padrão seu_id_no_docker_hub/nome_da_imagem:versao não faz com que ela seja armazenada no seu Docker Hub. Em breve utilizaremos um comando  docker para fazer isso. Neste momento, a sua imagem está armazenada localmente.

Há uma informação muito importante para usuários  minikube. Por padrão, o  minikube possui o seu próprio ambiente  docker, como mostra a Figura 4.3.70.

Figura 4.3.70



Entretanto, o cliente  , por padrão, se comunica com o 

daemon da máquina host, como ilustra a Figura 4.3.70. O  daemon é um processo servidor responsável por atender às requisições feitas pelo cliente . As imagens, contêineres etc criados ficam disponíveis na máquina host.

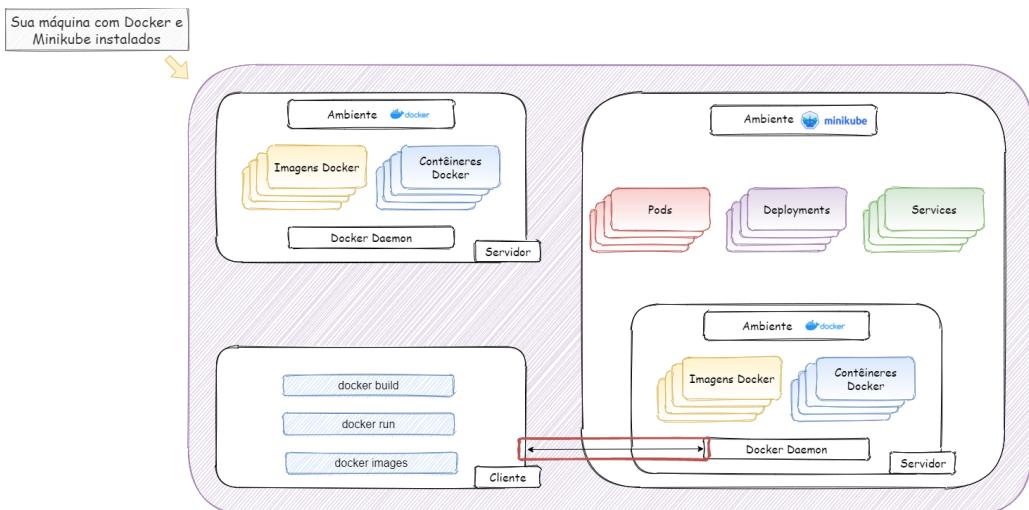
Quando o  tenta fazer uso de uma imagem por meio de seu rótulo, ele a procura localmente, em seu próprio ambiente , e não na máquina host.

Desta forma, caso a imagem tenha sido criada na máquina host, o  irá reportar um erro do tipo **ErrImagePull**. Para resolver esse problema, podemos instruir o cliente  a enviar requisições ao  daemon que executa no ambiente  do .

```
eval $(minikube docker-env) //Unix-like
minikube docker-env | Invoke-Expression //Powershell, Windows 10
```

O efeito é destacado na Figura 4.3.71.

Figura 4.3.71



Para fazer com que o cliente  passe a se comunicar novamente com o  daemon da máquina host, use

```
eval $(minikube docker-env -u) //Unix-like
minikube docker-env -u | Invoke-Expression //Powershell, Windows
10
```

Visite o Link 4.3.15 para mais detalhes.

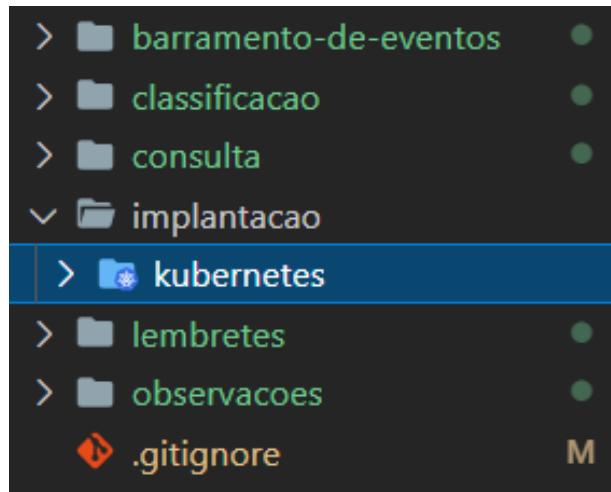
Link 4.3.15

<https://minikube.sigs.k8s.io/docs/handbook/pushing/>

Prosseguindo, crie uma pasta chamada *implantacao* e, dentro dela, uma outra

chamada *kubernetes*, como ilustra a Figura 4.3.72.

Figura 4.3.72



Na pasta *kubernetes*, crie um arquivo chamado *lembretes.yaml*. Seu conteúdo aparece no Bloco de Código 4.3.37. Ele especifica

- **apiVersion** - Qual versão da API do **kubernetes** está sendo usada para criar esse objeto. Ocorre que podemos especificar objetos customizados além daqueles disponíveis por padrão. Quando usamos **v1**, estamos dizendo que queremos criar objetos a partir da coleção padrão do **kubernetes**. Quando criamos objetos customizados, eles fazem parte de uma nova versão que criamos e que pode ser especificada aqui.
- **kind** - Qual o tipo do objeto.
- **metadata** - Dados sobre o objeto, como o seu nome e um identificador.
- **spec** - Descrição do estado desejado para o objeto.

Nota. Comentários em arquivos *.yaml* são definidos com o símbolo *#*.

Nota. Há diversos validadores de sintaxe yaml disponíveis on-line. Um deles pode ser acessado por meio do Link 4.3.16.

Link 4.3.16
<https://codebeautify.org/yaml-validator>

Bloco de Código 4.3.37

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4      name: lembretes
5  spec:
6      containers:
7          - name: lembretes
8              image: rodbossini/lembretes:0.0.1
9              resources:
10                 limits:
11                     memory: 256Mi
12                     cpu: 1
```

Nota. O *Pod* que estamos criando executará somente um contêiner. É aceitável e até desejável que tenham nomes iguais. No Bloco de Código 4.3.37 o nome de ambos é `lembretes`.

A seguir, a partir de um terminal vinculado à pasta em que se encontra o arquivo `lembretes.yaml`, use

```
kubectl apply -f lembretes.yaml
```

para criar o *Pod*. Verifique a sua existência com

```
kubectl get pods/lembretes
kubectl describe pods/lembretes
```

Nota. Caso deseje, um objeto do  **kubernetes** pode ser removido usando o mesmo arquivo usado na sua criação: `kubectl delete -f arquivo.yaml`.

4.3.69.2 Arquivo de configuração para um *Deployment* Como vimos, *deployments* podem ser utilizados para tarefas importantes envolvendo replicação de *Pods*, atualização de versão de imagem em uso nos *Pods* e assim por diante. Eles também podem ser criados com arquivos de configuração. Nesta seção iremos definir um *deployment* usando um arquivo de configuração. Os *Pods* a que ele estará associado também serão definidos assim, por isso, comece **removendo o arquivo `lembretes.yaml`**. Ele não causa nenhum dano mas não será mais usado. A seguir, crie um arquivo chamado `lembretes-deployment.yaml`. Seu conteúdo aparece no

Bloco de Código 4.3.38.

Bloco de Código 4.3.38

```
1 #deployments vem de apps/v1
2 apiVersion: apps/v1
3 #tipo
4 kind: Deployment
5 metadata:
6     #nome do deployment
7     name: lembretes-deployment
8 spec:
9     #quantas cópias
10    replicas: 1
11    #para especificar o rótulo
12    selector:
13        matchLabels:
14            #rótulo, app não tem nada de especial, pode ser qq coisa
15            #Deployment vai selecionar todo Pod que tiver esse rótulo
16            app: lembretes
17    #modelo que vai ser usado para construção dos Pods
18    template:
19        metadata:
20            labels:
21                #os Pods terão esse rótulo, assim,
22                #serão selecionados por esse deployment
23                app:lembretes
24    spec:
25        containers:
26            - name: lembretes
27                image: rodbossini/lembretes:0.0.1
28                resources:
29                    limits:
30                        memory: 256Mi
31                        cpu: 1
```

Crie o *deployment* com

kubectl apply -f lembretes-deployment.yaml

Verifique a existência de seu *deployment* com

kubectl get deployments
kubectl describe deployments

Verifique também que há um *Pod* em execução. Seu nome deve ser igual ao nome do *deployment* seguido de um código gerado pelo  **kubernetes**. É interessante observar que um *deployment* é uma configuração usada pelo  **kubernetes** para decidir quais *Pods* devem estar em execução. Se removermos o *Pod*, ele será recriado! Você pode testar isso com

```
//use o nome do pod que foi gerado na sua máquina
kubectl delete pod lembretes-deployment-76767c559c-x9m7g
//aguarde alguns segundos
kubectl get pods
```

Você deverá ver um novo Pod cujo nome contém o nome do *deployment*. Note que o código em seu nome é diferente do anterior.

4.3.70 Atualizando a imagem usada por um *deployment* Uma imagem sendo utilizada por um *deployment* pode ser atualizada de formas diferentes. Uma delas consiste em

- Fazer as atualizações desejadas no código-fonte da aplicação.
- Gerar uma nova imagem  **docker**.
- Atualizar o arquivo de configuração em que o *deployment* foi especificado para que ele use a nova versão.
- Usar `kubectl apply` para informar o  **kubernetes** sobre a atualização.

Para testar essa possibilidade, abra o arquivo *index.js* do microsserviço de lembretes. Adicione a linha destacada no Bloco de Código 4.3.39.

Bloco de Código 4.3.39

```
1   . . .
2   app.listen(4000, () => {
3     console.log('Nova versão')
4     console.log("Lembretes. Porta 4000");
5   );
```

A seguir, em um terminal vinculado ao diretório em que se encontram os arquivos do microsserviço de lembretes, execute

```
docker build -t rodbossini/lembretes:0.0.2 .
```

para gerar a nova imagem. Atualize o arquivo que descreve o *deployment* como mostra o Bloco de Código 4.3.40.

Bloco de Código 4.3.40

```
1 #deployments vem de apps/v1
2 apiVersion: apps/v1
3 #tipo
4 kind: Deployment
5 metadata:
6   #nome do deployment
7   name: lembretes-deployment
8 spec:
9   #quantas cópias
10  replicas: 1
11  #para especificar o rótulo
12  selector:
13    matchLabels:
14      #rótulo, app não tem nada de especial, pode ser qq coisa
15      #Deployment vai selecionar todo Pod que tiver esse rótulo
16      app: lembretes
17  #modelo que vai ser usado para construção dos Pods
18 template:
19   metadata:
20     labels:
21       #os Pods terão esse rótulo, assim,
22       #serão selecionados por esse deployment
23       app: lembretes
24   spec:
25     containers:
26       - name: lembretes
27         image: rodbossini/lembretes:0.0.2
```

Passe a utilizar o novo *deployment* com

```
kubectl apply -f lembretes-deployment.yaml
```

Ao executar

```
kubectl get pods
```

você deverá visualizar um novo *Pod* com tempo de vida de apenas alguns segundos. Copie o nome dele e use

```
kubectl logs lembretes-deployment-85968d5ff6-n2w9f//Use o nome
do seu pod aqui
```

para visualizar seu log. O método que utilizamos pode se tornar mais difícil ao longo do tempo, especialmente quando os arquivos `.yaml` se tornarem maiores e mais complexos. Usando este método, toda vez que houver alguma alteração na aplicação, teremos de abrir o arquivo `.yaml` correspondente e atualizar a versão, como fizemos atualizando de `0.0.1` para `0.0.2`. Em outras palavras, violamos o **princípio aberto/fechado**. Além disso, podemos digitar o número errado da versão. Uma **segunda possibilidade** de implantação consiste nos seguintes passos.

- Deixar de especificar a versão explicitamente, o que instrui o  **kubernetes** a utilizar a última versão disponível. Podemos simplesmente não especificar coisa alguma ou escrever a palavra **latest** no lugar da versão. Dá na mesma.
- Atualizar o código-fonte da aplicação conforme desejado.
- Gerar uma nova versão da imagem com o  **Docker**.
- Enviar a imagem para o **Docker Hub**.
- Aplicar a nova versão com `kubectl rollout`.

Comece atualizando o arquivo `lembretes-deployment.yaml` como destacado no Bloco de Código 4.3.41.

Bloco de Código 4.3.41

```

1  #deployments vem de apps/v1
2  apiVersion: apps/v1
3  #tipo
4  kind: Deployment
5  metadata:
6      #nome do deployment
7      name: lembretes-deployment
8  spec:
9      #quantas cópias
10     replicas: 1
11     #para especificar o rótulo
12     selector:
13         matchLabels:
14             #rótulo, app não tem nada de especial, pode ser qq coisa
15             #Deployment vai selecionar todo Pod que tiver esse rótulo
16             app: lembretes
17     #modelo que vai ser usado para construção dos Pods
18     template:
19         metadata:
20             labels:
21                 #os Pods terão esse rótulo, assim,
22                 #serão selecionados por esse deployment
23                 app: lembretes
24         spec:
25             containers:
26                 - name: lembretes
27                     image: rodbossini/lembretes

```

A seguir, atualize o arquivo *index.js* do microsserviço de lembretes como destacado no Bloco de Código 4.3.42.

Bloco de Código 4.3.42

```

1  . . .
2  app.listen(4000, () => {
3      console.log("Nova versão");
4      console.log("Agora usando o Docker Hub");
5      console.log("Lembretes. Porta 4000");
6  });

```

O próximo passo é gerar uma nova imagem com o . Com um terminal vinculado à pasta em que se encontra o arquivo *index.js* do microsserviço de lembretes, use

```
docker build -t rodbossini/lembretes .
```

Execute

```
docker images
```

e, como destaca a Figura 4.3.73, verifique que a *tag* da nova imagem é **latest**.

Figura 4.3.73

REPOSITORY	IMAGE ID	CREATED	TAG	SIZE
rodbossini/lembretes			latest	

Para enviar a imagem para o **Docker Hub**, faça login na sua conta com

```
docker login
```

A seguir, execute

```
docker push rodbossini/lembretes
```

para enviar a imagem para um repositório público do seu Docker Hub.

Nota. Depois de fazer um `docker push`, visite a página de seu Docker Hub. A imagem deve estar disponível lá. Por padrão, ela será armazenada em um repositório público. Contas grátis têm direito a um único repositório privado.

Nota. Caso uma imagem não exista localmente, ambos  **kubernetes** e  **minikube** irão tentar encontrá-la no ambiente  **docker** padrão (Docker Hub).

Para atualizar a imagem utilizada no *deployment*, use

```
kubectl rollout restart deployment lembretes-deployment
```

Verifique novamente a sua lista de *deployments* com

```
kubectl get deployments
```

Verifique também a sua lista de *Pods* com

kubectl get pods

Como destaca a Figura 4.3.74, deve haver um novo *Pod* com poucos segundos de tempo de vida.

Figura 4.3.74

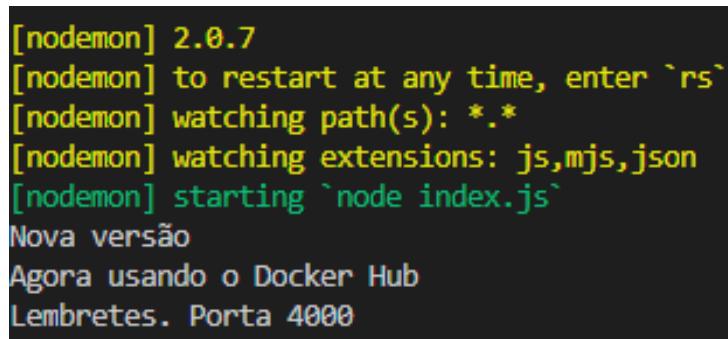
NAME	READY	STATUS	RESTARTS	AGE
lembretes-deployment-6557d596fd-9rp7q	1/1	Running	0	7s
lembretes-deployment-75b97c596f-dd7tv	0/1	Terminating	0	4m55s

Anote o nome do novo *Pod* e use

//use o nome do seu pod
kubectl logs lembretes-deployment-6557d596fd-9rp7q

para visualizar os logs do contêiner executando no novo *Pod*. O resultado deve ser parecido com aquele exibido pela Figura 4.3.75.

Figura 4.3.75



A terminal window displaying the output of a Node.js application using nodemon. The logs show the application starting up, watching files, and running on port 4000.

```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Nova versão
Agora usando o Docker Hub
Lembretes. Porta 4000
```

Nota. Pode ser necessário reiniciar o  kubernetes e remover as imagens armazenadas localmente. No Windows, você pode reiniciar o  kubernetes nas configurações do Docker Desktop. Se estiver usando o  minikube , use minikube delete, apague as imagens com docker image rm id_da_imagem e, a seguir, use minikube start.

Nota. É possível instruir o  **kubernetes** e o  **minikube** para que utilizem somente imagens locais. Veja o exemplo no Bloco de Código 4.3.43.

Bloco de Código 4.3.43

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: lembretes
5  spec:
6    containers:
7      - name: lembretes
8        image: rodbossini/lembretes
9        imagePullPolicy: Never
10   resources:
11     limits:
12       memory: 256Mi
13       cpu: 1
```

4.3.71 Serviço para acessar o microsserviço de lembretes Nesta seção, vamos criar um *serviço* que irá viabilizar o acesso à aplicação que implantamos. Assim como os demais objetos, serviços também podem ser criados por meio de arquivos de configuração. Comece criando um arquivo chamado *lembretes-service.yaml* na pasta *implantacao/kubernetes*. Seu conteúdo aparece no Bloco de Código 4.3.44.

Bloco de Código 4.3.44

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4      name: lembretes-service
5  spec:
6      #Ip externo, acessível fora do cluster
7      type: NodePort
8      selector:
9          #todo Pod que tiver essa tag
10         #fará parte desse serviço
11         app: lembretes
12     ports:
13         - name: lembretes
14             protocol: TCP
15             #o nó recebe requisições nessa porta
16             port: 4000
17             # e direciona para essa porta do Pod
18             targetPort: 4000

```

Para criar o serviço use,

kubectl apply -f lembretes-service.yaml

Verifique a sua existência com

**kubectl get services
kubectl describe service lembretes-service**

A Figura 4.3.76 destaca a porta aleatória que foi associada ao serviço. Esta é a porta que deve ser usada externamente. O cliente envia requisições para esta porta. Internamente, o *serviço* as redireciona para a porta 4000 do *Nó* que, por sua vez, as entrega ao contêiner em sua porta 4000.

Figura 4.3.76

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	61m
lembretes-service	NodePort	10.108.158.187	<none>	4000:32143/TCP	29s

Use o Postman para fazer uma requisição em `http://localhost:sua_porta/lembretes`.

Nota. Caso esteja utilizando o  **minikube**, lembre-se de obter o seu IP com **minikube ip**.

4.3.72 Comunicação interna entre os microsserviços Até então, os microsserviços utilizam o endereço IP da máquina host para se comunicarem. Isso já não será possível uma vez que sejam executados por instâncias de *Pods* do  **kubernetes**. Para viabilizar a comunicação interna entre microsserviços executando em uma instância do  **kubernetes**, usamos **serviços do tipo Cluster IP**.

4.3.72.1 Deployment e serviço de tipo Cluster IP para o barramento de eventos e para o microsserviço de lembretes Começamos colocando o barramento de eventos em funcionamento e viabilizando a sua comunicação com o microsserviço de lembretes. Isso será feito da seguinte forma.

- Construir uma imagem  **docker** para o barramento de eventos.
- Enviar a imagem para o Docker Hub.
- Criar um *Deployment* para o barramento de eventos.
- Criar um serviço do tipo *Cluster IP* para o barramento de eventos e para o microsserviço de lembretes.

Para criar a imagem  **docker** para o barramento de eventos, vá até o diretório em que se encontra o respectivo arquivo *Dockerfile* e use

```
docker build -t rodbossini/barramento-de-eventos .
```

Envie a imagem para o Docker Hub com

```
docker push rodbossini/barramento-de-eventos
```

Repita o procedimento para a imagem  **docker** do microsserviço de lembretes. Crie uma nova com

```
docker build -t rodbossini/lembretes .
```

e envie para o Docker Hub com

```
docker push rodbossini/lembretes
```

Crie um arquivo chamado *barramento-de-eventos-deployment.yaml* na pasta *implantacao/kubernetes*. Seu conteúdo aparece no Bloco de Código 4.3.45.

Bloco de Código 4.3.45

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: barramento-de-eventos-deployment
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: barramento-de-eventos
10   template:
11     metadata:
12       labels:
13         app: barramento-de-eventos
14   spec:
15     containers:
16       - name: barramento-de-eventos
17         image: rodbossini/barramento-de-eventos
```

Para criar o novo *Deployment*, use

```
kubectl apply -f barramento-de-eventos-deployment.yaml
```

Execute

```
kubectl get pods
```

para verificar se há um novo *Pod* em funcionamento. O prefixo de seu nome deve ser **barramento-de-eventos**. Seu tempo de vida deve ser de alguns segundos. O serviço do tipo *Cluster IP* para disponibilização interna do barramento de eventos também será criado no arquivo *barramento-de-eventos-deployment.yaml*. Veja o Bloco de Código 4.3.46.

Bloco de Código 4.3.46

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4      name: barramento-de-eventos-deployment
5  spec:
6      replicas: 1
7      selector:
8          matchLabels:
9              app: barramento-de-eventos
10     template:
11         metadata:
12             labels:
13                 app: barramento-de-eventos
14     spec:
15         containers:
16             - name: barramento-de-eventos
17                 image: rodbossini/barramento-de-eventos
18     ---
19     apiVersion: v1
20     kind: Service
21     metadata:
22         name: barramento-de-eventos-service
23     spec:
24         selector:
25             app: barramento-de-eventos
26             #ip interno ao cluster
27             type: ClusterIP
28         ports:
29             - name: barramento-de-eventos
30                 protocol: TCP
31                 port: 10000
32                 targetPort: 10000
```

Para confirmar a criação do serviço, use

kubectl apply -f barramento-de-eventos-deployment.yaml

O novo serviço criado pode ser visualizado com

kubectl get services

O resultado deve ser parecido com aquele exibido pela Figura 4.3.77.

Figura 4.3.77

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
barramento-de-eventos-service	ClusterIP	10.111.2.239	<none>	10000/TCP
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP
lembretes-service	NodePort	10.104.55.15	<none>	4000:30686/TCP

A criação do serviço de tipo *Cluster IP* para o microsserviço de lembretes é semelhante. Abra o arquivo *lembretes-deployment.yaml* e adicione o conteúdo destacado no Bloco de Código 4.3.47.

Bloco de Código 4.3.47

```

1 #deployments vem de apps/v1
2 apiVersion: apps/v1
3 #tipo
4 kind: Deployment
5 metadata:
6   name: lembretes-deployment #nome do deployment
7 spec:
8   #quantas cópias
9   replicas: 1
10  #para especificar o rótulo
11  selector:
12    matchLabels:
13      #rótulo, app não tem nada de especial, pode ser qq coisa
14      #Deployment vai selecionar todo Pod que tiver esse rótulo
15      app: lembretes
16  #modelo que vai ser usado para construção dos Pods
17 template:
18   metadata:
19     labels:
20       #os Pods terão esse rótulo, assim,
21       #serão selecionados por esse deployment
22       app: lembretes
23 spec:
24   containers:
25     - name: lembretes
26       image: rodbossini/lembretes
27 ---
28 apiVersion: v1
29 kind: Service
30 metadata:
31   #nome diferente do outro de tipo NodePort
32   #que havíamos criado
33   name: lembretes-clusterip-service
34 spec:
35   selector:
36     app: lembretes
37   ports:
38     - name: lembretes
39       protocol: TCP
40       port: 4000
41       targetPort: 4000

```

O barramento de eventos tenta se comunicar com todos os microsserviços.

Vamos manter a sua comunicação somente com o microsserviço de lembretes para depois fazer os ajustes necessários. Veja o Bloco de Código 4.3.48.

Bloco de Código 4.3.48

```

1  const eventos = [];
2  app.post("/eventos", (req, res) => {
3      const evento = req.body;
4      eventos.push(evento);
5      //envia o evento para o microsserviço de lembretes
6      axios.post("http://localhost:4000/eventos", evento);
7      //envia o evento para o microsserviço de observações
8      //axios.post("http://localhost:5000/eventos", evento);
9      //envia o evento para o microsserviço de consulta
10     // axios.post("http://localhost:6000/eventos",
11         evento).catch((err) => {
12         // console.log("err", err);
13         // });
14         //envia o evento para o microsserviço de classificação
15         //axios.post("http://localhost:7000/eventos", evento);
16         res.status(200).send({ msg: "ok" });
17     });

```

Nossos microsserviços estão fazendo requisições direcionadas a **localhost** o que já não funciona já que eles estão sob coordenação dos serviços  **kubernetes** que criamos. É preciso substituir a constante **localhost** usando os nomes dos serviços criados. Os nomes dos serviços são aqueles especificados nos arquivos *.yaml*. Também é possível obtê-los com

kubectl get services

Veja o resultado na Figura 4.3.78.

Figura 4.3.78

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
barramento-de-eventos-service	ClusterIP	10.111.2.239	<none>	10000/TCP
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP
lembretes-clusterip-service	ClusterIP	10.111.22.17	<none>	4000/TCP
lembretes-service	NodePort	10.104.55.15	<none>	4000:30686/TCP

Comece ajustando o microsserviço de lembretes. Em seu arquivo *index.js*, substitua a constante **localhost** pelo nome do serviço  **kubernetes** que dá acesso ao barramento de eventos. Veja o Bloco de Código 4.3.49.

Bloco de Código 4.3.49

```

1 //era assim
2   //await axios.post("http://localhost:10000/eventos", {
3   //fica assim
4   await axios.post("http://barramento-de-eventos-service:1000_"
5     "0/eventos",
6   {
7     tipo: "LembreteCriado",
8     dados: {
9       contador,
10      texto,
11    },
12  });
13  res.status(201).send(lembretes[contador]);
14);

```

Faça o mesmo ajuste para o barramento de eventos. Faça com que ele use o nome do serviço  **kubernetes** que dá acesso interno ao microsserviço de lembretes, como no Bloco de Código 4.3.50.

Bloco de Código 4.3.50

```

1 app.post("/eventos", (req, res) => {
2   const evento = req.body;
3   eventos.push(evento);
4   //envia o evento para o microsserviço de lembretes
5   //era assim
6   //axios.post("http://localhost:4000/eventos", evento);
7   //fica assim
8   axios.post("http://lembretes-clusterip-service:4000/eventos_"
9     "",
10    evento);
11   //envia o evento para o microsserviço de observações
12   //axios.post("http://localhost:5000/eventos", evento);
13   //envia o evento para o microsserviço de consulta
14   // axios.post("http://localhost:6000/eventos",
15     evento).catch((err) => {
16     //  console.log("err", err);
17     // });
18   //envia o evento para o microsserviço de classificação
19   //axios.post("http://localhost:7000/eventos", evento);
20   res.status(200).send({ msg: "ok" });
21 });

```

Feitas as atualizações, precisamos gerar novas imagens  . Vá até a

pasta em que se encontra o arquivo *Dockerfile* do barramento de eventos e execute

```
docker build -t rodbossini/barramento-de-eventos .
```

para gerar a nova imagem. A seguir, use

```
docker push rodbossini/barramento-de-eventos
```

para enviá-la para o Docker Hub. Repita o procedimento para o microserviço de lembretes. Na pasta em que se encontra o seu arquivo *Dockerfile*, execute

```
docker build -t rodbossini/lembretes .
```

A seguir, envie a imagem para o Docker Hub com

```
docker push rodbossini/lembretes
```

A seguir, precisamos atualizar os *deployments*. Use

```
kubectl get deployments
```

caso precise se lembrar de seus nomes. Veja a Figura 4.3.79.

Figura 4.3.79

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
barramento-de-eventos-deployment	1/1	1	1	6d20h
lembretes-deployment	1/1	1	1	7d12h

A atualização dos *deployments* pode ser feita com

```
kubectl rollout restart deployment  
barramento-de-eventos-deployment
```

e

```
kubectl rollout restart deployment lembretes-deployment
```

Execute

```
kubectl get pods
```

e verifique os novos *Pods* criados. Eles devem ter poucos segundos de tempo de vida. É possível que você ainda veja os antigos sendo destruídos, caso execute este comando logo após a atualização dos *deployments*. Veja a Figura 4.3.80.

Figura 4.3.80

NAME	READY	STATUS	RESTARTS	AGE
barramento-de-eventos-deployment-6cb889765b-7xzjj	1/1	Running	0	23s
lembretes-deployment-6557d596fd-tzx55	0/1	Terminating	1	6d19h
lembretes-deployment-74f46c9b87-lzcp4	1/1	Running	0	8s

4.3.72.2 Testando a comunicação interna entre microsserviço de lembretes e barramento de eventos Neste instante já deve ser possível testar a comunicação interna entre o microsserviço de lembretes e o barramento de eventos. Para isso, precisamos utilizar um cliente HTTP (como o Postman) para enviar uma requisição ao cluster  **kubernetes**. A requisição partirá de um ambiente externo ao cluster, por isso precisamos usar uma porta que tenha sido aberta previamente. Lembre-se que isso é feito utilizando um serviço  **kubernetes** do tipo *NodePort* ou *LoadBalancer*. O serviço  **kubernetes** do tipo *NodePort* que criamos anteriormente pode ser visualizado com

kubectl get services

Veja o resultado na Figura 4.3.81. Note que é possível visualizar a porta que foi aberta para o ambiente externo.

Figura 4.3.81

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
barramento-de-eventos-service	ClusterIP	10.111.2.239	<none>	10000/TCP	6d19h
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	7d21h
lembretes-clusterip-service	ClusterIP	10.111.22.17	<none>	4000/TCP	6d19h
lembretes-service	NodePort	10.104.55.15	<none>	4000-30686/TCP	7d

O endereço IP a ser utilizado para fazer a requisição é aquele associado ao ambiente em que o cluster está em execução. Caso esteja utilizando o **Docker for Desktop**, basta utilizar a constante **localhost**. Usuários do  **minikube** podem obter o seu endereço ip com

minikube ip

Abra o Postman e faça uma requisição **PUT** como ilustra a Figura 4.3.82. Lembre-se de substituir **localhost** pelo ip apropriado, caso esteja utilizando o  **minikube**.

Figura 4.3.82

The screenshot shows the Postman interface. The URL is set to `localhost:30686/lembretes`. A red box highlights the `PUT` method. The `Body` tab is selected, and the dropdown shows `JSON`. The request body contains the following JSON:

```

1
2   "1": {
3     "contador": 1,
4     "texto": "Fazer café"
5   }
6

```

The response status is `200 OK`, and the response body is identical to the request body.

A seguir, faça também um **GET**, como na Figura 4.3.83.

Figura 4.3.83

The screenshot shows the Postman interface. The URL is set to `localhost:30686/lembretes`. A red box highlights the `GET` method. The `Body` tab is selected, and the dropdown shows `JSON`. The response status is `200 OK`, and the response body is identical to the request body.

Para ter certeza de que tudo deu certo, verifique os logs do *Pod* que executa o microsserviço de lembretes. Ele deve ter recebido um evento do barramento de eventos e exibido na saída padrão. Pegue o nome do *Pod* com

kubectl get pods

Veja a Figura 4.3.84.

Figura 4.3.84

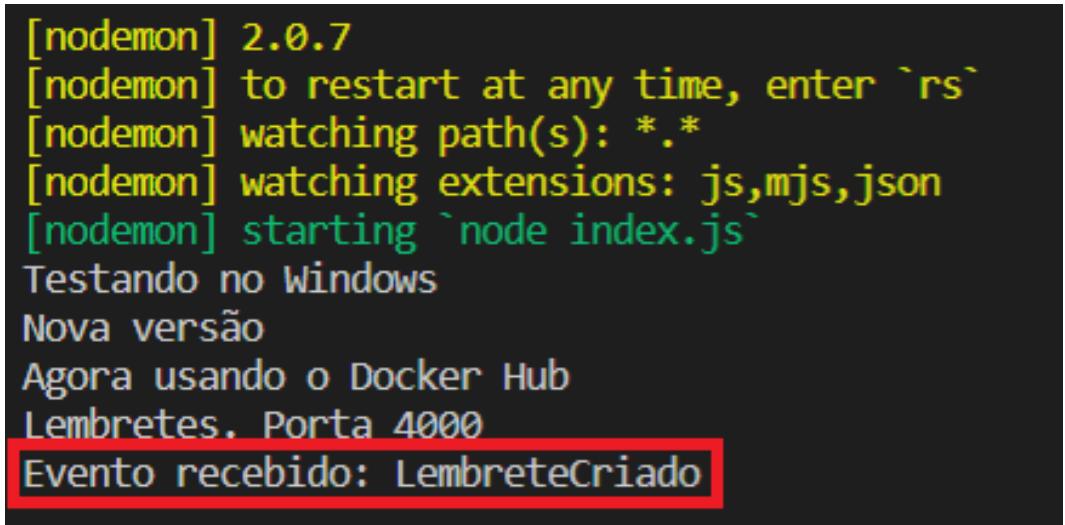
NAME	READY	STATUS	RESTARTS	AGE
barramento-de-eventos-deployment-6cb889765b-7xzjj	1/1	Running	0	28m
lembretes-deployment-78dc5fd895-lg9xr	1/1	Running	0	3m13s

A seguir, use

kubectl logs nome_do_seu_pod

para visualizar seus logs. A saída deve ser parecida com o que exibe a Figura 4.3.85.

Figura 4.3.85



```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Testando no Windows
Nova versão
Agora usando o Docker Hub
Lembretes. Porta 4000
Evento recebido: LembreteCriado
```

Caso a sua saída não exiba uma mensagem assim, pode ser que o seu microserviço de lembretes não esteja registrando cada evento que recebe. Você pode ajustar isso como mostra o Bloco de Código 4.3.51. Estamos no arquivo *index.js* do microsserviço de lembretes.

Bloco de Código 4.3.51

```
1   . . .
2   app.post("/eventos", (req, res) => {
3     console.log("Evento recebido: " + req.body.tipo);
4     res.status(200).send({ msg: "ok" });
5   );
6   .
```

Nota. Pode ser que um ou mais de seus microsserviços apresente alguma falha durante a execução, o que pode comprometer o funcionamento da solução como um todo. Nestes casos é recomendável verificar o log de cada *Pod* envolvido com `kubectl logs`. Se encontrar alguma mensagem de erro, investigue a sua causa e, se necessário, ajuste o código fonte. Depois disso, gere uma nova imagem  `docker` com `docker build`, envie a imagem para o Docker Hub com `docker push` e reinicie o deployment envolvido com `kubectl rollout restart deployment`.

4.3.72.3 Exercícios Crie novos serviços  `kubernetes` do tipo Cluster IP e coloque os demais microsserviços em funcionamento. Faça um teste completo e certifique-se de que a solução está completamente funcional.

Capítulo 5

Instalação do



O **node** é um ambiente que viabiliza, entre outras coisas, a execução de código Javascript do lado do servidor. A sua instalação traz também o **Node Package Manager (npm)**, um gerenciador de pacotes por meio do qual podemos fazer a instalação de pacotes disponíveis no ecossistema **node**. Há algumas formas para realizar a sua instalação.

5.1 Instalação com o instalador regular do NodeJS Uma maneira bastante simples para fazer a instalação do NodeJS é por meio do download do seu instalador, disponível no site oficial[2]. A instalação do NodeJS já implica na instalação do npm.

5.2 Instalação usando um gerenciador de versões Há ainda a possibilidade de fazer a instalação do **node** por meio de um **Node Version Manager (NVM)**, ou seja, um gerenciador de versões do **node**. Ele permite que tenhamos diversas versões do **node** instaladas e que façamos a alternância entre elas conforme desejado. Além disso, o funcionamento do **node** ocorre no diretório do usuário do sistema operacional, o que quer dizer que seu uso tende a evitar problemas de permissão para acesso a determinados diretórios. O instalador de um NVM pode ser obtido nos links 5.2.1 (Linux e Mac) e 5.2.2(Windows).

Link 5.2.1
<https://github.com/nvm-sh/nvm>

Link 5.2.2
<https://github.com/coreybutler/nvm-windows>

Uma vez instalado o NVM, para instalar o  , basta usar

nvm install versao-desejada

É interessante instalar a última versão LTS disponível na maior parte dos casos. As versões disponíveis para instalação podem ser listadas com

nvm ls-remote

no Linux e no Mac e com

nvm list available

no Windows. A última versão LTS disponível no momento em que esse documento foi escrito, era a 14.15.5. Para fazer a sua instalação, o comando é

nvm install 14.15.5

A seguir, para colocá-la em uso, use

nvm use 14.15.5

Você pode verificar se o  foi corretamente instalado com

node --version

ou com

node -v

Referências

- [1] L. Bass, P. Clements e R. Kazman. *Software Architecture in Practice*. 3^a ed. Addison-Wesley Professional, 2012.
- [2] Dahl, R. *Node.js*. Disponível em: <https://nodejs.org/en/>. Acesso em abril de 2021.
- [3] A. Fox e D. Patterson. *Engineering Software as a Service - An Agile Approach Using Cloud Computing*. 1^a ed. Strawberry Canyon LLC, 2018.
- [4] Martin Fowler. *Who Needs an Architect?* Disponível em: <https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf>. Acesso em fevereiro de 2021.
- [5] TJ Holowaychuk. *Express - Node.js web application framework*. Disponível em: <https://expressjs.com>. Acesso em abril de 2021.