Universidade Federal do Rio Grande do Norte
Arturo Fonseca de Souza, Vinícius Ernani de Carvalho
Relatório de desenvolvimento: comparando desempenho entre arquiteturas MVC e reativa.

1 INTRODUÇÃO

O campo de construção de aplicações web sempre foi marcado pelo rápido desenvolvimento tecnológico e pela rápida adaptação das tecnologias às novas tendências tecnológicas. Ainda dentro desse campo, há alguns anos desenvolveu-se o que hoje é conhecido como Arquitetura Reativa.

Não só uma solução arquitetural que aprimora as aplicações e as torna mais responsivas, os paradigmas encontrados dentro da Arquitetura Reativa a tornam também mais moderna e atraente do que as soluções arquiteturais convencionais para Softwares que vigoravam na década passada, como o padrão MVC.

Tal arquitetura é moderna porque apresenta a aplicação de grandes avanços tecnológicos na própria maneira de pensar os Softwares, como a busca por uma aplicação responsiva e resiliente, e é também atraente pois usa muito bem tecnologias que estão voltando a ter fama e espaço dentro do desenvolvimento de soluções tecnológicas, como a programação funcional.

Outra característica interessante dos sistemas que implementam uma arquitetura reativa é sua flexibilidade. São sistemas que reagem melhor às mudanças, tanto de ambiente (como, por exemplo, o caso de uma aplicação de compras em período de promoção), tanto de escopo de projeto, por naturalmente serem divididos em serviços desacoplados.

Dessa forma, tendo em vista o conjunto de vantagens que um sistema reativo pode trazer, o intuito desse trabalho é de realizar um teste prático entre dois sistemas semelhantes construídos com arquiteturas diferentes.

O primeiro dos sistemas foi construído utilizando o padrão MVC, enquanto o segundo sistema apresenta uma arquitetura reativa. Ambos os sistemas foram construídos em Java junto ao framework Spring, que foi a tecnologia escolhida para realizar este experimento.

2 A APLICAÇÃO

2.1 Sobre o ReservaGo

O sistema proposto para realizar essa análise foi o ReservaGo, uma aplicação que concentra hotéis, casas, apartamentos e espaços passíveis de reserva e os lista para usuários interessados em alugar tais localidades.

O ReservaGo é constituído por três microsserviços que comunicam-se através de requisições HTTP. O primeiro dos três serviços é o serviço de admin, que realiza o armazenamento de todos os dados que serão utilizados na aplicação. O segundo serviço é

o serviço de checkout, responsável por prover todos os dados que serão visualizados ou consultados pelos usuários na hora de efetivar o aluguel de uma localidade. O terceiro e último serviço é o serviço payment, responsável por validar o pagamento dos clientes e gerar as transações que servirão para histórico e consulta futura no banco de dados.

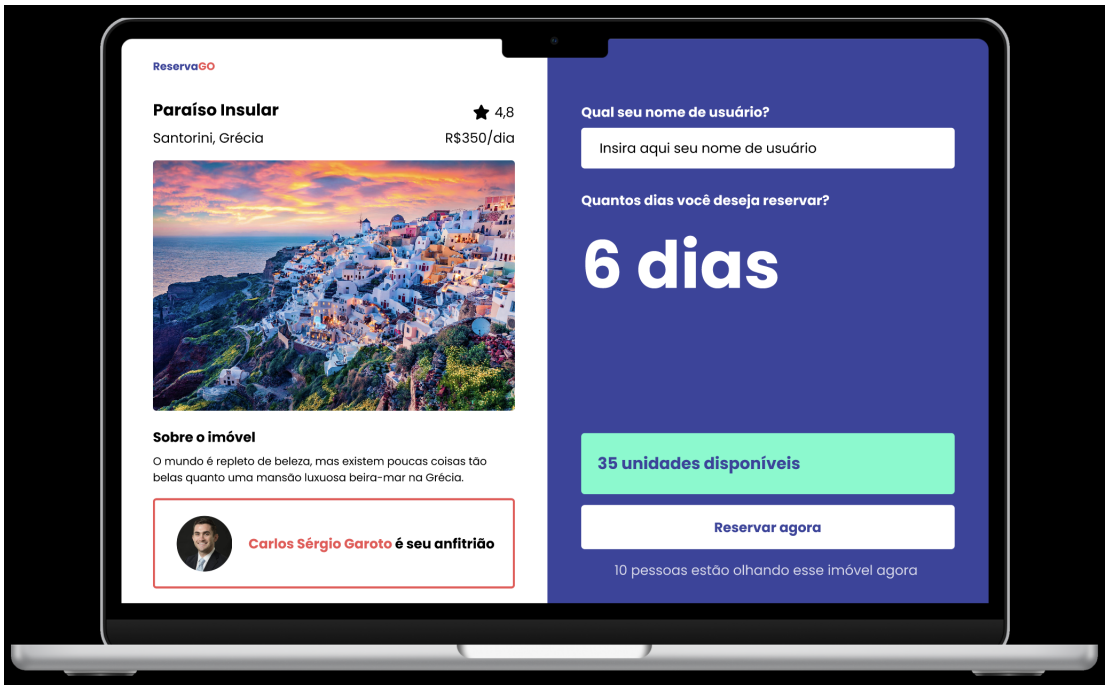

Figura 1 – Visual do cliente da aplicação ReservaGo

2.2 Diagrama de Serviços

Por se tratar de uma aplicação grande, com múltiplos serviços rodando ao mesmo tempo, é interessante ter uma visão geral de como é realizada a composição da aplicação.

Primeiramente, é importante compreender que todas as chamadas à aplicação saem de um meio cliente. O cliente, hoje, é uma instância do Postman (ou Insomnia), mas há também uma aplicação cliente que é apta a realizar chamadas diretas à API.

Tendo isso em mente, o projeto se organiza da seguinte forma:

- Existe um serviço de configuração que informa os dados de funcionamento para todos os outros serviços dentro do ReservaGo.
- Um serviço de descoberta é utilizado para dinamicamente encontrar e disponibilizar as aplicações através da rede.
- Um serviço de balanceamento de carga é utilizado para trafegar entre as diferentes instâncias da aplicação registradas no serviço de descoberta.

- Um serviço de gateway é responsável por gerir o encaminhamento das requisições realizadas à ele para outros serviços dentro da aplicação, permitindo a divisão de chamadas entre diferentes serviços.
- Um banco de cache é utilizado para agilizar a resposta a requisições já conhecidas ou frequentes.
- Três serviços são responsáveis pelo funcionamento das regras de negócio da aplicação (Admin, Checkout, Payment).
- Cada um dos serviços anteriores tem uma instância de um banco de dados não relacional rodando dentro de um container.

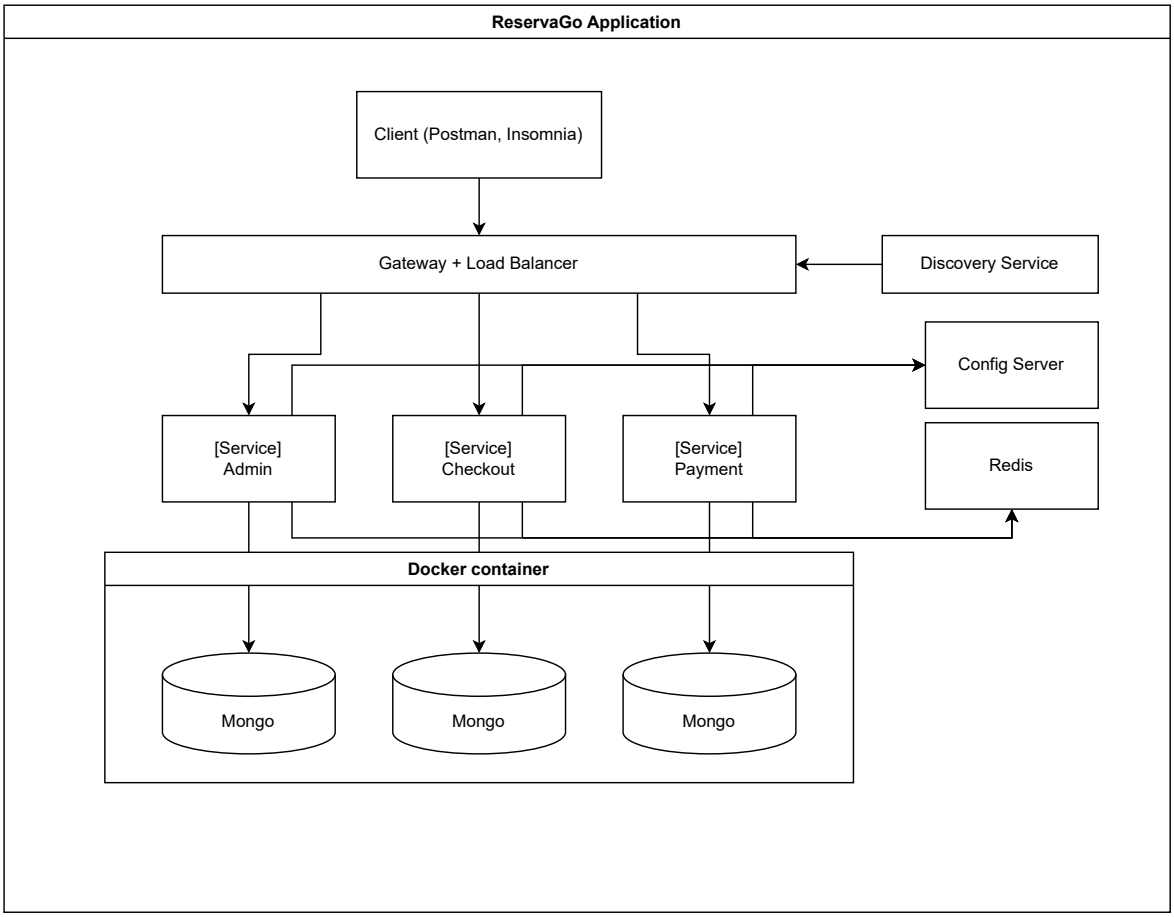

Figura 2 – Diagrama de Serviços da aplicação.

3 METODOLOGIA

Buscando realizar uma boa avaliação das duas arquiteturas, será analisado o comportamento da taxa de vazão por segundo (throughput per second) ao passo em que

aumenta-se a quantidade de usuários que tentam realizar o fluxo do sistema, um processo conhecido como Teste de Carga.

Como ponto principal de foco, analisaremos como o sistema se comporta quando centenas ou milhares de usuários tentam realizar o checkout (uma ação que consulta os dois outros sistemas) ao mesmo tempo.

Para obter-se tal métrica, utilizaremos uma ferramenta chamada JMeter, que possui capacidade não só de realizar tais testes como também de gerar relatórios precisos e de maneira automática, evitando possíveis erros humanos no processo de observação das requisições.

É importante denotar, também, que este teste foi realizado em um Asus TUF Gaming B450M-PRO II com o seguinte Hardware: processador AMD Ryzen 5 3400G 4-core, memória RAM de 16GB DDR4, com armazenamento de 1000GB.

4 SISTEMA COM ARQUITETURA MVC (NÃO REATIVA)

4.1 Sobre o Sistema

Os detalhes de desenvolvimento do sistema não reativo podem ser visualizados no próprio repositório da aplicação, hospedado no Github.

4.2 Desempenho nos testes

4.2.1 Testes de verificação

O teste de verificação faz uma chamada GET à API de checkout através do endpoint `checkout/verify/id`, onde id é o ID da Localidade que queremos verificar. Esse endpoint é utilizado para verificar a disponibilidade de uma localidade para aluguel.

Tabela 1 – Desempenho do teste de aluguel no MVC.

Número de usuários	Vazão por Segundo (throughput/s)
100	20,0/s
200	40,2/s
500	100/s
1000	200,0/s
2000	239,8/s
4000	258,5/s
6000	285,0/s

4.2.2 Teste de aluguel

No teste de aluguel, nós chamamos o endpoint `/checkout/verify/1` passando um payload genérico. Esse endpoint é utilizado para realizar o aluguel de uma localidade.

Tabela 2 – Desempenho do teste de aluguel no MVC.

Número de usuários	Vazão por Segundo (throughput/s)
100	20,2/s
200	40,2/s
500	99,9/s
1000	200,0/s
2000	239,8/s
4000	258,6/s
6000	285,0/s

4.3 Total dos testes

Considera-se o total dos testes o valor de vazão dos dois testes somados.

Tabela 3 – Desempenho total da aplicação MVC.

Número de usuários	Vazão por Segundo (throughput/s)
100	20,4/s
200	80,0/s
500	199,4/s
1000	398,6/s
2000	478,9/s
4000	516,5/s
6000	569,4/s

Além da tabela com os valores, o gráfico de linhas abaixo apresenta a progressão da vazão de maneira visual, deixando explícito o momento de joelho e o momento em que o sistema atingiu sua carga máxima com o aumento da base de usuários de teste.

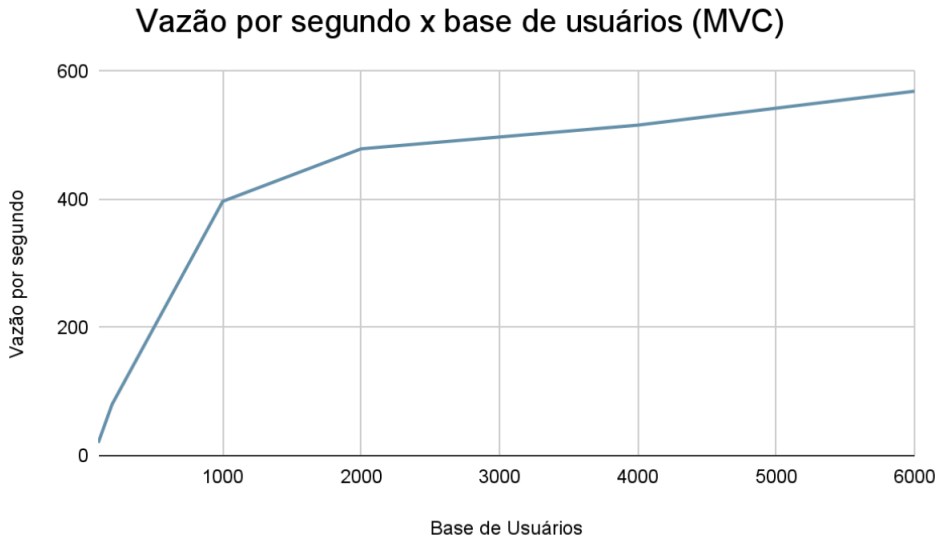

Figura 3 – Gráfico do Sistema MVC

5 SISTEMA COM ARQUITETURA FLUX (REATIVA)

5.1 Sobre o sistema

Os detalhes de desenvolvimento do sistema reativo podem ser visualizados no próprio repositório da aplicação, hospedado no Github.

5.2 Desempenho nos testes

5.2.1 Teste de verificação

Esse teste é semelhante ao teste realizado na seção 4.2.1 deste presente relatório.

Tabela 4 – Desempenho do teste de verificação no Flux.

Número de usuários	Vazão por Segundo (throughput/s)
100	20,1/s
200	39,9/s
500	74,3/s
1000	89,4/s
2000	98,7/s
4000	96,5/s
6000	88,8/s

5.2.2 Teste de aluguel

Este teste é semelhante ao teste realizado na seção 4.2.2 deste presente relatório.

Tabela 5 – Desempenho do teste de aluguel no Flux.

Número de usuários	Vazão por Segundo (throughput/s)
100	22,1/s
200	40,1/s
500	70,8/s
1000	75,6/s
2000	73,0/s
4000	66,8/s
6000	64,0/s

5.3 Total dos testes

Este teste é semelhante ao teste realizado na seção 4.3 deste presente relatório.

O gráfico de linhas do sistema exibe bem o momento de carga máxima do sistema, caracterizado pela queda de throughput independente do aumento do número de usuários.

Tabela 6 – Desempenho total da aplicação Flux utilizando Virtual Threads.

Número de usuários	Vazão por Segundo (throughput/s)
100	39,7/s
200	79,1/s
500	140,2/s
1000	150,6/s
2000	145,7/s
4000	133,4/s
6000	127,8/s

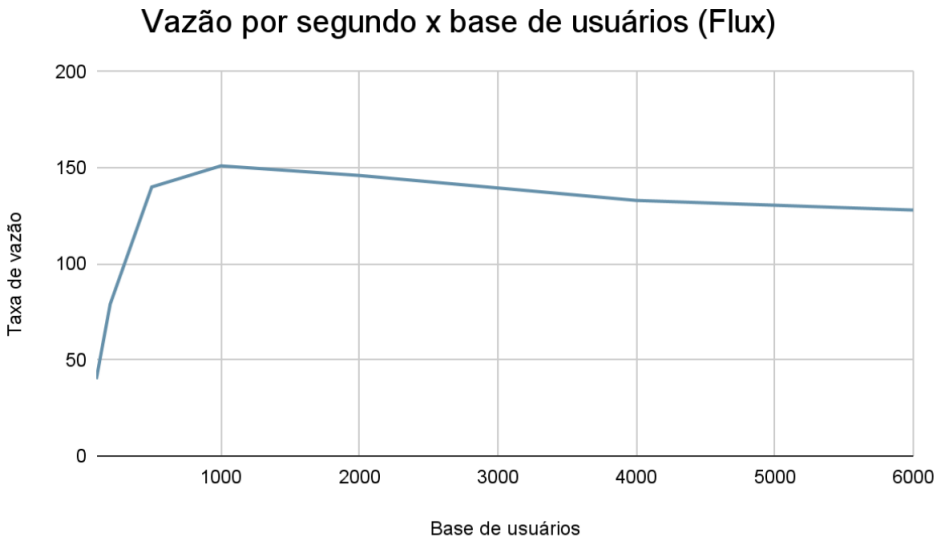

Figura 4 – Gráfico do Sistema Flux.

6 SEGUNDA ENTREGA

Como parte do modelo avaliativo da disciplina, ocorreu, durante a segunda unidade de aulas, a implementação de novas tecnologias que garantiam um melhor funcionamento e desempenho do sistema.

Tais configurações visam:

- Melhorar a maneira como a aplicação gerencia suas configurações utilizando Spring Cloud Config Server.
- Desacoplar o gerenciamento de rotas do DNS e habilitar técnicas de Load Balancing utilizando Spring Cloud Discovery Service e Spring Cloud Load Balance.
- Habilitar a rápida resposta de requisições amplamente utilizadas utilizando Cacheamento com Redis.

7 SPRING CLOUD CONFIG SERVER

7.1 Introdução ao Config Server

Um gargalo comumente encontrado em soluções de Software, e diretamente relacionado com a reprodução da aplicação em ambientes, sejam eles de desenvolvimento ou não, é a maneira de realizar a configuração do sistema.

Usualmente se utilizam arquivos de configuração que contém informações sobre chaves de acesso a serviços, senhas e usuários de bancos de dados, URLs internas e todo tipo de informação potencialmente confidencial que deve ter acesso restrito.

O problema em utilizar-se arquivos assim é que, além da dificuldade de compartilhar essas informações, pois é complexo garantir sua segurança, não há também garantias de que os usuários criarão corretamente esses arquivos. Ainda, o tempo consumido para realizar essa configuração também deve ser levado em consideração.

Visando solucionar esse problema, realizou-se a adição de um Servidor de Configuração no ReservaGo. Tal solução consiste na criação de um Servidor, que age como serviço, e que concentra todas as configurações da aplicação, tornando-as acessíveis através de URLs que podem ser requisitadas de dentro das aplicações.

Essa adição arquitetural elimina a necessidade de arquivos de configuração descentralizados, garantindo que todos os usuários terão acesso às mesmas configurações ao passo de que restringe os arquivos somente a pessoas de fato autorizadas e que todas essas configurações serão as corretas.

7.2 Implementação do Config Server

A escolha de tecnologia para o ReservaGo foi o Spring Cloud Config Server, por ser nativo do Spring e facilmente integrável.

Para implementá-lo, foi criada uma aplicação que continha o pacote Config Server do Spring e um repositório Git (armazenado no Github) que foi utilizado para armazenar os arquivos de configuração.

Dessa maneira, ao levantar-se o servidor de configuração, todos os serviços que implementarem o Spring Client poderão acessar os arquivos contidos no repositório através de requisições no cliente.

Além disso, existe um arquivo de configuração específico para o Config Server que serve para definir de que forma o serviço irá funcionar.

7.3 Implementação do Config Client

Para contatar o servidor de configuração através de chamadas, todos os serviços precisam ter a dependência Spring Cloud Config Client. Essa dependência é a responsável

```yaml
server:
  port: 8888

spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/erneani/reservago-config-repo.git
          search-paths: '{application}/{profile}'
          password: notausualpassword
          username: reservagochef
  security:
    user:
      name: root
      password: root
```

Figura 5 – Arquivo *application.yaml*, localizado no Servidor de Configuração.

por permitir o acesso ao repositório de configuração e também por realizar a configuração automática do projeto com os dados contidos no Github.

Quando levantados, os serviços buscarão importar as configurações diretamente do Servidor de Configuração.

```yaml
spring:
  application:
    name: checkout
  profiles:
    active: dev
  config:
    import: optional:configserver:http://localhost:8888

```

Figura 6 – Configurações do serviço de *Checkout* após implementação do Config Server.

8 SERVIÇO DE DISCOVERY E LOAD BALANCING

8.1 Balanceando chamadas na aplicação

Ao criar um Software que pretende ser entregue de maneira distribuída, é importante compreender que a maneira como os usuários terão seus pedidos distribuídos entre os servidores da aplicação não é aleatória e muito menos mágica.

Para uma distribuição de chamadas eficiente, garantindo que seus serviços se mantenham robustos e responsivos, é importante que um serviço de Balanceamento de Carga seja configurado.

Para tal configuração, é importante também automatizar a maneira como nomeamos e endereçamos nossas aplicações, uma vez que realizar tais configurações manualmente não só é lento, como também pode ser perigoso, uma vez que se pode realizar a configuração incorretamente e tornar o serviço indisponível, mesmo gastando uma pequena fortuna em servidores.

Logo, os serviços de Load Balancing e Discovery são essenciais para manter uma aplicação robusta, funcional e reativa.

8.2 Configurando o servidor de descoberta

O servidor de descoberta serve como uma camada de DNS da aplicação, mas, na verdade, automatiza o roteamento e o endereçamento dos serviços para que não seja necessário haver troca de informações através do próprio DNS.

Para configurar o Discovery Service no ReservaGo, se utilizou da biblioteca Spring Cloud Netflix Eureka, mantida oficialmente pelo Spring e testada pela própria Netflix.

```yaml
server:
  port: 8761

spring:
  application:
    name: discovery

eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
```

Figura 7 – Configuração do Servidor de *Discovery* no *Config Server*.

8.3 Configurando o Load Balancer

É importante ter um servidor de descoberta para automatizar os endereçamentos, mas ele sozinho não é capaz de realizar o balanceamento de carga.

Portanto, adiciona-se, também, o Load Balancer, um serviço que, de maneira inteligente, configurável ou aleatória, distribui as chamadas entre diferentes instâncias da aplicação.

Para configurar o Load Balancer na aplicação, utilizou-se o Spring Cloud Load Balancer, que fornece uma maneira integrada ao Spring de realizar essa configuração. A integração entre as aplicações agora funciona muito bem, mas é necessário utilizar uma

notação de balanceamento de carga para garantir que os Endpoints utilizados passarão por esse processo de load balancing.

Para tal, adiciona-se uma anotação sobre as funções que utilizarão essa funcionalidade ou, como é mais comum, no próprio serviço de chamadas HTTP.

```
19      @Bean
20      @LoadBalanced
21      public WebClient.Builder loadBalancedWebClientBuilder() {
22          return WebClient.builder();
23      }
```

Figura 8 – *Decorator* acima do *Builder* do cliente HTTP

9 SERVIDOR DE CACHE

9.1 O conceito de cacheamento de dados

É muito comum, durante a vida útil de um servidor, que diferentes partes do sistema recebam, também, uma proporção diferente de todas as chamadas feitas ao sistema.

Contudo, em alguns casos, é interessante se observar quais partes do sistema mais são requisitadas e avaliar se é possível criar uma espécie de resposta padrão para não só agilizar as respostas como também desafogar uma parte do sistema que tem uso contínuo e intenso durante todo o dia.

Pensando nessa grande quantidade de chamadas a partes específicas da aplicação, elaborou-se um conceito de Caching de aplicações WEB.

A ideia por trás do Caching é simples: partes bastante requisitadas da aplicação provavelmente devem entregar respostas iguais muitas vezes ao dia. Sabendo disso, pode-se armazenar essas informações em um canto de fácil acesso para que, toda vez que uma requisição com os mesmos parâmetros seja realizada, a resposta seja entregue sem necessidade de processamento, diretamente do servidor de cache.

9.2 Implementando cache no ReservaGo

Na aplicação ReservaGo, há também pontos que se beneficiariam muito de uma estrutura de Cache distribuído.

Um exemplo é a requisição para verificar disponibilidade de um Local, uma vez que não é frequente a atualização de locais, permitindo que essas informações passem um bom tempo no Cache sem necessidade de mutação e agilizando as chamadas.

Portanto, implementou-se o cache na aplicação utilizando Redis.

10 GATEWAY

10.1 Introdução ao Gateway

Uma vez que as aplicações modernas se tornaram capazes de dinamicamente disponibilizar os serviços através dos servidores de descoberta e também de reorganizar o fluxo de chamadas entre os serviços através do balanceamento de carga, passou-se a existir demanda por uma tecnologia que se responsabilizaria por dinamicamente realizar o roteamento das chamadas entre as instâncias dos serviços utilizando as regras aplicadas pelo Load Balancer. Para suprir essa necessidade, a tecnologia de Gateway foi criada.

Um Gateway é um Proxy Reverso que recebe as informações de roteamento do serviço de descoberta e, fazendo uso do Load Balancer, distribui as chamadas entre as diferentes instâncias da aplicação sem a necessidade de um cliente definir a instância alvo do serviço.

10.2 Implementando o Gateway

Como o ReservaGo é uma aplicação com múltiplos serviços e que implementa um servidor de descoberta e um balanceador de carga, é imprescindível que para o funcionamento ideal da aplicação se utilize um Gateway.

Dessa forma, implementou-se um Gateway utilizando o Spring Cloud Gateway, uma tecnologia integrada ao Spring que facilita a integração entre os serviços Spring já existentes.

```
4  spring:
5    application:
6      name: gateway
7    cloud:
8      gateway:
9        routes:
10         - id: admin-route
11           uri: lb://ADMIN/admin
12           predicates:
13             - Path=/admin/
14           filters:
15             - name: Retry
```

Figura 9 – Exemplo de configuração de rota estática no *Gateway*

Figura 10 – Configuração de filtro de *Retry* no *Gateway*

11 TERCEIRA ENTREGA

Como característica da terceira entrega, implementou-se modelos de resiliência e de messageria na aplicação.

12 RESILIÊNCIA

Para tornar a aplicação resiliente a erros e permitir melhor resposta aos usuários quando os dados dados começarem a trafegar na rede, utilizou-se a biblioteca Resilience4j.

12.1 Um pouco de contexto

Uma aplicação falha. Esse fato imprescindível, por mais assustador que seja, deve ser pensado e estruturado por quem escreve Software.

Mais do que garantir aos usuários que os dados circularão corretamente na aplicação, lidar com as falhas e deixar o usuário ciente dessas ações é importante para garantir a confiabilidade do produto e demostrar respeito com o usuário final, que é quem sofre com esses erros.

Diz-se, portanto, que toda maneira de lidar com erros na aplicação é uma tentativa de *tornar o produto resiliente*.

12.2 Implementando o Resilience4j

Famosa no ambiente Java de desenvolvimento, a biblioteca Resilience4j traz consigo um conjunto de boas práticas de resiliência que podem ser rapidamente configuradas

através dos arquivos de configuração de uma aplicação.

De fácil implementação, a biblioteca conta com diversas funcionalidades que auxiliam na tratativa de erros para os usuários finais, com um importante destaque aos conceitos de *Circuit Breaker*, *Retry*, *Rate Limit*, *Fallback* e *Bulk Head*.

No ReservaGo, todas as soluções destacadas anteriormente estão presentes, mas a mais interessante de se discorrer nesse relatório é o conceito de Circuit Breaker.

12.3 Circuit Breakers no ReservaGo

Um *Circuit Breaker* é uma funcionalidade que age como um disjuntor, em formato de Software, para solucionar problemas com erros.

Utilizando-se do conceito de **falha rápida**, o principal intuito de tal conceito é de garantir que, ao receber uma certa quantidade configurável (em porcentagem) de erros na aplicação, um serviço pare de vez para análise e toda requisição feita a ele será trabalhada, seja redirecionada ou apenas retornada imediatamente avisando a indisponibilidade do servidor.

```
return this.reactiveCircuitBreakerFactory.create("placeBreaker").run(check,
        throwable -> Mono.error(new ServiceNotRespondingException("Please, wait a minute while "
        + "our services come back to life!")));
```

Figura 11 – Utilização do *Circuit Breaker* no serviço de *Checkout*.

```
resilience4j:
  circuitbreaker:
    configs:
      default:
        registerHealthIndicator: true
        slidingWindowSize: 10  #How much time req
        permittedNumberOfCallsInHalfOpenState: 3
        slidingWindowType: TIME_BASED
        minimum-number-of-calls: 5
        wait-duration-in-open-state: 10s
        failure-rate-threshold: 30
        event-consumer-buffer-size: 10
        record-exceptions:
          - java.lang.RuntimeException
```

Figura 12 – Configuração do *Circuit Breaker* no *Config Server*.

13 MENSAGERIA

Na aplicação, nesta terceira unidade, também está presente o conceito de troca de mensagens.

Por mais simples que pareça, o conceito de troca de mensagens utilizando um *broker de mensagens* permite que a aplicação utilize um modelo totalmente diferente de arquitetura, que abstrai a complexidade de recursos de infraestrutura e permite aos codificadores focarem principalmente na maneira como a aplicação deve ser comportar e de que forma esses recursos podem ser escalados para uma maior utilização.

13.1 Arquitetura Event-Driven

Com o usufruto de serviços de mensageria, outras arquiteturas interessantes podem ser implementadas na aplicação.

Tendo em noção essa possibilidade, abaixo está o diagrama de organização da aplicação orientada a eventos.

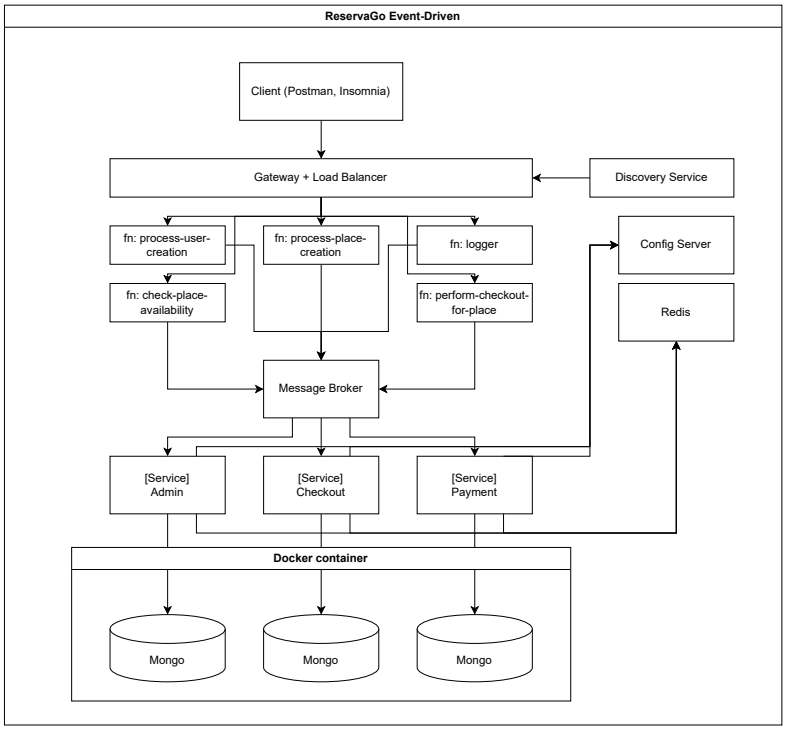

Figura 13 – Diagrama do Reservago com Arquitetura Orientada a Eventos

13.2 Implementando Mensageria no ReservaGo

Na aplicação, dois serviços de mensageria, Kafka e RabbitMQ, foram utilizados, cada um a seu modo.

Além disso, a implementação de ambos os serviços se deu ao lado de Cloud Functions, através da utilização de bibliotecas que performam o *bind* entre as Cloud Functions e o serviço de Mensageria utilizado (seja ele Kafka, seja ele RabbitMQ).

Os arquivos de configuração podem ser visualizados no repositório da aplicação, sob as *branchs* de rótulo `feat-rabbitmq` e `feat-kafka`.

14 CONCLUSÃO

Após a realização dos testes, de maneira não intuitiva, a aplicação construída com arquitetura MVC foi superior, em termos de vazão, à aplicação reativa.

Esse fato pode, claro, se dever a vários fatores. Alguns dos pontos importantes de mencionar neste presente relatório que podem ter apresentado um ambiente mais favorável para o MVC: são a arquitetura de testes, que deixou os testes em Flux por último, podendo ter ocasionado sobrecarga do banco de dados em algum momento e provocado lentidão; os endpoints chamados, que retornavam apenas objetos individuais, sem gerar uma fila significativa, não causando quebra de responsividade; falhas no desenvolvimento da aplicação com padrão Flux, como configurações que não foram seguidas à risca e ocasionaram na lentidão da aplicação como um todo.

Dentro dos sistemas, note que a capacidade de joelho do sistema MVC é atingida ao chegar em mil usuários, mas sua capacidade máxima de carga não é apresentada.

Enquanto isso, o sistema Flux apresenta sua capacidade de joelho pouco antes dos mil usuários e, ao atingir essa marca, atinge também sua capacidade máxima de carga, representada no Gráfico 2 pela queda de vazão com o aumento dos usuários.