Testes de microsserviços de ponta a ponta

Sheldon Smith
Computer Science
Baylor University
Waco, Texas, USA
sheldon_smith2@alumni.baylor.edu

Ethan Robinson
Computer Science
Baylor University
Waco, Texas, USA
Ethan_Robinson2@alumni.baylor.edu

Timmy Frederiksen
Computer Science
Baylor University
Waco, Texas, USA
timmyfrederiksen@gmail.com

Trae Stevens
Computer Science
Baylor University
Waco, Texas, USA
trae_stevens1@alumni.baylor.edu

Tomas Cerny
Computer Science
Baylor University
Waco, Texas, USA
tomas_cerny@baylor.edu

Miroslav Bures
Computer Science
Czech Technical University, FEE
Prague, Czech Republic
buresm3@fel.cvut.cz

Davide Taibi
M3S Cloud Group
University of Oulu
Oulu, Finland
davide.taibi@oulu.fi

***Abstract*—Testar sistemas de microserviços envolve uma grande quantidade de planejamento e resolução de problemas. A dificuldade de testar sistemas de microserviços aumenta à medida que o tamanho e a estrutura de tais sistemas se tornam mais complexos. Para ajudar a comunidade de microserviços e simplificar experimentos com testes e simulação de tráfego, criamos um benchmark de teste contendo cobertura completa de testes funcionais para dois sistemas de microserviços de código aberto bem estabelecidos. Através do design de nosso benchmark, buscamos demonstrar maneiras de superar certos desafios e encontrar estratégias efetivas ao testar microserviços. Além disso, para demonstrar o uso de nosso benchmark, realizamos um estudo de caso para identificar as melhores abordagens a serem tomadas para validar uma cobertura completa de testes usando descoberta de gráfico de dependência de serviço e descoberta de processos de negócios usando rastreamento.**

***Index Terms*—Microservices, Teste de Carga, Teste de Regressão Funcional, Teste de Funcionalidade, Benchmark**

I. Introdução

Os microsserviços são a abordagem principal para a construção de sistemas nativos da nuvem. A arquitetura de microsserviços (MSA) prescreve que um sistema compreende serviços implementáveis de forma independente que interagem [**?**]. Um microsserviço deve ter uma única responsabilidade, o que significa que eles gerenciam apenas uma parte específica das necessidades da organização. Os microsserviços permitem a evolução descentralizada de partes do sistema e sua escalabilidade seletiva. A criação de um serviço baseado em MSA ajuda a eliminar a dependência de determinadas tecnologias, permitindo que os usuários acessem a infraestrutura específica do serviço [**?**]. No entanto, os microsserviços são desenvolvidos e evoluídos por equipes independentes e descentralizadas, apesar do resultado que os usuários veem como um produto holístico.

Garantir que o sistema geral funcione adequadamente, independentemente de suas dependências internas e decomposição, é vital. Com a complexidade cada vez maior do sistema e especialmente quando descentralizado, é importante buscar a cobertura completa do sistema, o que muitas vezes é difícil de conseguir. Mas só então poderíamos descobrir se todas as funções do sistema são avaliadas para entradas adequadas e impróprias.

Juntamente com o teste funcional, é vital avaliar se o sistema responde adequadamente a várias quantidades de usuários e solicitações em carga simultânea. Esse teste de carga pode acompanhar os testes funcionais enquanto monitora os tempos de resposta do sistema.

Considerando os testes funcionais e de carga, identificamos que a comunidade de microsserviços está perdendo um benchmark de conjunto de testes que poderia ser usado para novos avanços e evolução da pesquisa neste campo. Devido a essa falta, elaboramos um estudo de caso para mostrar nosso progresso na geração desse benchmark.

O objetivo central de nossa contribuição é fornecer à comunidade científica um conjunto de testes abrangente para dois sistemas de microsserviços bem estabelecidos. Especificamente, estamos focando este trabalho em testes funcionais e de carga. Esses tipos de teste são uma parte importante do teste da funcionalidade geral desses sistemas.

Apresentamos os benchmarks do conjunto de testes por meio de um estudo de caso. Também identificamos e discutimos vários desafios que surgiram ao testar esses sistemas e apresentamos algumas das melhores práticas e correções para problemas comuns. Desenvolvemos um conjunto completo de testes de "melhor esforço" que abrange os pontos de extremidade dos vários microsserviços em cada sistema. O conjunto de testes pode ser usado por pesquisadores para validar seu trabalho em testes de microsserviços em um conjunto de testes comum.

As demais seções estão organizadas da seguinte forma. A Seção 2 fornece informações sobre os microsserviços e ferramentas. A Seção 3 cobre nosso estudo de caso seguido pelo benchmark apresentado na Seção 4. A Seção 5 conclui o artigo.

II. Background

O teste funcional garante que vários recursos do sistema funcionem de acordo com as especificações ou expectativas da funcionalidade do sistema. Os casos de teste normalmente são baseados nas especificações dos componentes de software e cada teste corresponde a um determinado requisito ou recurso de software. Isso ajuda a garantir que as partes do conjunto geral de software funcionem de forma independente. O foco nos requisitos de software garante que a saída seja consistente com as expectativas do usuário final.

Para simular a interação do usuário com sistemas de microsserviços, consideramos o sistema como uma caixa preta e usamos suas interfaces de usuário sem saber detalhes internos. Para auxiliar nesses testes, vários frameworks trazem a capacidade de projetar scripts de teste. Por exemplo, a estrutura do Selenium facilita o teste do sistema da web. Ele fornece um método de navegadores da Web automatizados que acionam eventos como se os usuários os tivessem feito.

O teste de regressão funcional verifica o sistema após a modificação, garantindo que todos os recursos, fluxos e funcionalidades do sistema estejam funcionando como na versão anterior ou por uma especificação ou expectativa de funcionalidade correta. Ele ajuda na garantia de qualidade do sistema e ajuda a evitar mudanças não intencionais no sistema causadas por mudanças apressadas.

O

Além da funcionalidade, também é importante analisar como um sistema se comporta durante várias condições de carga. O teste de carga é o processo de simulação da demanda do sistema que testa o comportamento sob várias condições. O termo 'carga' neste caso refere-se à taxa ou número de usuários e requisições acessando o determinado sistema [**?**]. O teste de carga ajuda a garantir que as ações do usuário no sistema sejam estáveis, avaliando como o sistema reage a várias quantidades de carga.

Existem três maneiras principais de executar o teste de carga. Essas técnicas de geração de carga incluem a utilização de usuários reais para gerar a carga, usando drivers de carga ou implantando os testes de carga em plataformas especiais [**?**]. Avaliar o comportamento de um sistema em várias quantidades de carga pode destacar áreas de melhoria, como gargalos ou áreas de falha no sistema. Esses problemas geralmente só podem ser descobertos usando o teste de carga porque eles ocorrem ou são visíveis apenas sob uma certa quantidade de carga no sistema.

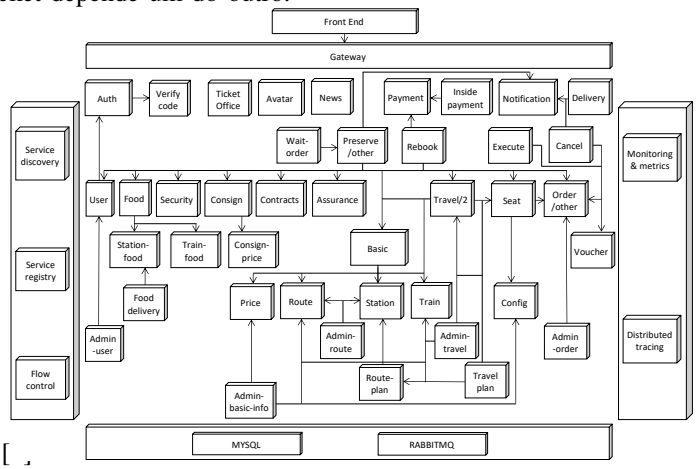
Da mesma forma, existem estruturas que podem ajudar na automação de testes. Por exemplo, Gatling [**?**] pode executar um script de teste e gerar relatórios de carga abrangentes.

Outra métrica importante no teste de software é a cobertura do teste. Ajuda a medir a quantidade de testes feitos em um determinado sistema. A cobertura dos testes mostra quais partes do sistema estão sendo executadas ao longo dos testes, medida por uma razão entre o número de elementos particulares do sistema cobertos pelos testes e o número total desses elementos no sistema.

A. Referências do sistema de microsserviços

Para demonstrar testes funcionais e de carga para microsserviços, foram utilizados dois sistemas bem estabelecidos e baseados na comunidade que fornecem uma ampla gama de funcionalidades que podem ser testadas ao longo de um estudo de caso. O Train-Ticket [**?**] é baseado na plataforma Java, e o eShopOnContainers [**?**] usa C#. Essa seleção de sistema nos permite ilustrar diferentes cenários em nosso estudo de caso, resultando em um benchmark de teste compartilhável para esses sistemas.

The Train-Ticket benchmark fornece um sistema de reserva de bilhetes de trem baseado em 47 microsserviços (a partir da versão 1.0.0). A figura **??** mostra o layout geral e a estrutura da arquitetura Train-Ticket. Ele mostra como o front-end, o sistema de monitoramento e os serviços interagem dentro do sistema. Essa figura permite que os usuários entendam como cada um dos microsserviços do sistema Train-Ticket depende um do outro.



Os casos de uso envolvidos com este sistema de microsserviço podem ser divididos em ações de usuário e administrador. Certas ações estão presentes para todos os usuários, como o sistema de login, enquanto outras dependem do tipo de usuário.

Existem seis casos de uso `usuário` principais dentro do sistema Train-Ticket. Esses casos são a procura de um trem, a reserva de uma passagem, a atualização da consignação, o pagamento da passagem, a retirada da passagem e a entrada na estação.

Os casos de uso do `admin` envolvem a adição, atualização e exclusão de vários elementos do sistema de emissão de bilhetes, como pedidos, rotas, planos de viagem, usuários, contatos, estações, trens, preços e configurações.

Para nosso estudo de caso, estamos usando a versão 1.0.0 do Train-Ticket, lançada em 9 de agosto de 2022 [**?**]. Este sistema de microsserviço foi criado pela Fudan University CodeWisdom Team. O objetivo original desse sistema era fornecer um sistema de referência para bilhetagem ferroviária [**?**].

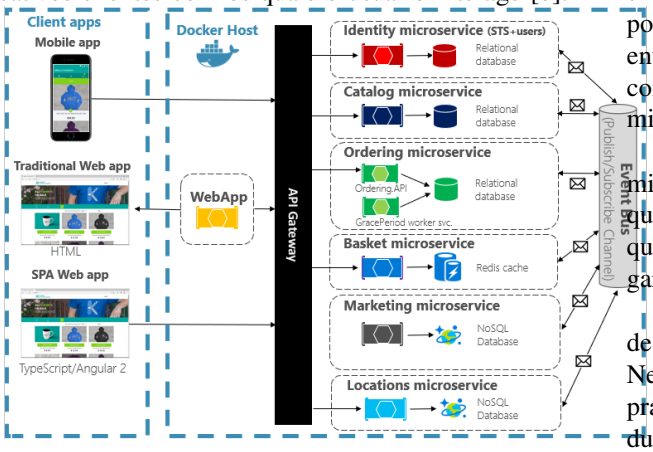
Este sistema foi criado usando uma infinidade de diferentes linguagens de programação e frameworks como Java (Spring

Boot, Spring Cloud), Node.js (Express), Python (Django), Go (Webgo) e MongoDB e MySQL para os bancos de dados [**?**].

O sistema de microsserviço eShopOnContainers é um exemplo de aplicativo de referência .Net Core [**?**]. O sistema está centrado em fornecer vários casos de uso envolvidos em aplicativos de compras eletrônicas. O front-end para este microsserviço é dividido entre dois aplicativos da web: um aplicativo da web tradicional feito com HTML e um aplicativo de página única (SPA) feito com typescript e Angular 2. Há também um componente de aplicativo móvel para esse sistema de microsserviço.

A arquitetura desse aplicativo é multiplataforma no servidor e no lado do cliente. A figura **??** mostra um layout da interação entre os aplicativos cliente e o host Docker. Dentro do host Docker, existem vários microsserviços autônomos com cada serviço contendo seus próprios dados ou banco de dados. Diferentes abordagens para a estrutura dos microsserviços são utilizadas, como padrões CRUD e DDD/CQRS. HTTP é a principal forma de comunicação entre esses microsserviços e os aplicativos clientes com os quais o usuário interage [**?**].

[h]



Há uma infinidade de casos de uso que o usuário pode executar neste aplicativo. Para interagir ao máximo com a funcionalidade, o usuário deve fazer login com a conta de demonstração pré-criada ou pode registrar uma nova conta no sistema. Um usuário pode filtrar os itens com base em vários campos, o que não depende do login do usuário no sistema. Um `usuário registrado` pode adicionar itens ao carrinho, que podem ser atualizados na página do carrinho. Os usuários podem concluir o processo de checkout para finalizar o pedido ou cancelar o pedido. Por fim, os usuários podem visualizar os pedidos anteriores que concluíram.

Para nosso estudo de caso, usamos a versão 5.0.0 do microsserviço eShopOnContainers. Este sistema de microsserviço é fornecido como um dos aplicativos de referência pela .NET Application Architecture [**?**]; além disso, possui amplas contribuições da comunidade em seu repositório de código-base.

III. Estudo de caso

O objetivo do nosso estudo de caso é criar um conjunto abrangente de testes centrados na funcionalidade de todos os endpoints do sistema, de modo a avaliar o sistema e sua funcionalidade sob estresse e determinar o comportamento de sistemas de microsserviços bem estabelecidos.

Como estamos nos concentrando em testes funcionais e de carga para nosso estudo de caso, nosso benchmark de teste está centrado na avaliação do tempo de resposta das ações, na interação da interface do usuário e na funcionalidade geral do sistema.

O teste de funcionalidade considera o sistema como uma caixa preta e desconhece a decomposição interna para microsserviços, que muitas vezes reflete a visão do usuário. O benchmark de teste de interface do usuário (baseado na web) ajuda a garantir a consistência durante a execução das ações do usuário.

O teste de funcionalidade por meio da interface do usuário garante que o sistema exiba as informações de maneira correta e consistente. Se houver variabilidade nas informações exibidas, isso poderá afetar muito a experiência do usuário.

O teste de carga visa identificar possíveis locais de gargalo e lentidão. Isso é importante porque a experiência do usuário pode ser significativamente afetada por esses problemas. No entanto, com o teste de carga, podemos considerar o sistema como uma caixa cinza e focar nos endpoints expostos ao middleware.

Como mencionado anteriormente, os sistemas de microsserviços de teste de carga são vitais para garantir que os vários microsserviços reajam bem a diferentes quantidades de carga. Isso pode ajudar a identificar áreas de gargalos e falhas no sistema.

Ao longo do processo de nosso estudo de caso, vários desafios surgiram ao testar esses sistemas de microsserviços. Neste estudo de caso, apresentaremos algumas das melhores práticas e correções para problemas comuns que podem surgir durante o teste.

A. *Estudo de caso de teste de regressão funcional*

Queríamos garantir a exatidão da funcionalidade fornecida pelo sistema de microsserviço implementando um conjunto de testes automatizado baseado na Web. Usamos o Selenium para essa finalidade e conseguimos garantir que os microsserviços funcionem e sejam exibidos corretamente em vários ambientes de navegador. Queríamos garantir a implementação de testes automatizados para testar minuciosamente a funcionalidade e o design dos microsserviços. Para este estudo de caso, nosso objetivo foi fornecer um conjunto de testes completo que inclui testes funcionais e de interface do usuário para simular de perto a experiência de um usuário ao utilizar os sistemas de microsserviços.

Usamos o framework Selenium para escrever testes automatizados que simulavam a experiência de um usuário dos sistemas de microsserviços Train-Ticket e eShopOnContainers.

1) *Desenho do Estudo*

Primeiro, criamos manualmente uma lista abrangente de casos de uso em cada microsserviço. Nossa abordagem para criar uma lista de casos de uso abrangentes foi registrar todas as

ações possíveis que o usuário poderia realizar ao interagir com o microsserviço e testar o resultado dessa ação. Acompanhamos os casos de uso em uma planilha e marcamos cada caso de uso à medida que o teste automatizado foi concluído. Manter uma lista de casos de teste nos ajudou a acompanhar quais casos de uso foram totalmente testados e forneceu um bom indicador de nosso progresso até a cobertura total. Este documento foi compartilhado junto com nosso benchmark de teste no Zenodo [**?**].

2) Procedimento do Estudo / Enfrentando as Armadilhas Iniciais

Descartamos o Katalon Web Recorder [**?**], pois os scripts resultantes continham muitos códigos não modularizados, repetitivos e ilegíveis. Prosseguimos manualmente com nossos próprios scripts Selenium. Executáveis como testes JUnit para nosso teste automatizado de navegador da web, testaríamos nosso trabalho usando um navegador da web Chrome. Esse navegador da Web nos permitiu visualizar como nosso código interagia com os microsserviços e nos permitiu interromper a execução e inspecionar o navegador da Web durante a depuração. Embora o uso de um navegador da web Chrome seja bom para depurar e escrever os testes, usar esse método não é eficiente ao executar vários testes, pois vários navegadores da web são gerados para cada teste e sobrecarregam a tela.

Agrupamos casos de uso semelhantes em testes para reduzir o número de testes executados. Agrupar os testes ajudou a reduzir etapas redundantes e aumentou a eficiência. Após a criação dos testes, em vez de executá-los no navegador Chrome, passamos a executar os testes no HTML Unit WebDriver, que é um driver da Web sem GUI. Ele suporta JavaScript e simulará um navegador da Web para testes com outros frameworks, como JUnit [**?**]. Isso também acelerou o tempo de processamento de nossos testes JUnit. No entanto, queríamos tornar nossos testes ainda mais eficientes, paralelizando os.

Utilizamos o framework TestNG com a capacidade de paralelizar os testes [**?**]. Além disso, o TestNG permite que o testador tenha mais controle sobre os testes e é capaz de especificar o número de threads em que os testes serão executados. Passamos do uso do JUnit como nossa estrutura de teste para o uso do TestNG. Após a troca de frameworks, paralelizamos nossos testes para torná-los mais eficientes. Após a paralelização dos testes do Selenium, agora o tempo necessário para executá-los foi reduzido drasticamente.

3) Resultados do Estudo

Como nosso procedimento era criar um conjunto abrangente de testes da funcionalidade fornecida pelo usuário, nosso objetivo era interagir com todas as funcionalidades possíveis expostas ao usuário para melhor simular a experiência do usuário. Como resultado do nosso estudo de caso de regressão funcional, acabamos testando um total de 51 casos de uso para o sistema TrainTicket e um total de 26 casos de uso para o sistema eShopOnContainers, conforme mostrado na Tabela **??**.

[h]

Conjunto de testes e **# de casos de uso do Selenium**	
Admin TrainTicket	33
Cliente TrainTicket	18
Combined TrainTicket	**51**
eShopOnContainers	**26**

final

Embora tenhamos nos esforçado para implementar totalmente todos os casos de uso possíveis, havia um caso de uso no sistema de microsserviço TrainTicket que não pudemos testar totalmente, que era o serviço de consignação. Embora tenhamos escrito completamente o teste para o serviço de remessa, não conseguimos implantar esse serviço no sistema TrainTicket. Portanto, nosso teste funcionaria teoricamente, mas não conseguimos verificar seus resultados.

O conjunto de teste de reserva TrainTicket abrange todos os casos de uso do lado do cliente, devido à sua co-dependência um do outro. O conjunto de teste de reserva envolve o uso do recurso de pesquisa regular para localizar e reservar uma passagem, pagar pela passagem, alterar a ordem da passagem, coletar a passagem e entrar na estação. O conjunto de teste de reserva também inclui o uso do recurso de pesquisa avançada para encontrar e reservar um segundo bilhete, cancelar esse bilhete e, finalmente, excluir todas as informações adicionadas no final do teste. Todos esses casos de uso são ações que um cliente normalmente realizaria ao reservar e gerenciar seu bilhete.

[h]

Conjunto de testes e nome do teste	
Lista de Admin	ConfigLista Lista de contatos Conecte-se Pedidos Lista de Preços ListadeRotas StationList TrainList TravelList Lista de usuários
Cliente	Reservas Conecte-se

final

∈

Conjunto de Testes	Teste	Casos de Uso
Cliente	Login do cliente	Login válido/inválido, Usuário/Senha inválido, Sair
	Reservas	Reservar Econômica, Reservar Primeira Classe, Criar/Salvar Contato, Reserve com Garantia, Refeições, com Consignar, Cancelar
	Lista de pedidos	Número de telefone inválido, Atualizar Consignar, Pagar pelo Bilhete
	Recolher Bilhete	Recolher Bilhete
	Entrar na Estação	Entrar na Estação
Administrador e lista de contatos e adicionar/atualizar/excluir contato	Login do administrador	Login válido/inválido, Usuário/Senha inválido, Sair
	Lista de Pedidos	Adicionar/Atualizar/Excluir Pedido
	Lista de Rotas	Adicionar/Atualizar/Excluir Rota
	Lista de Viagens	Adicionar/Atualizar/Excluir Viagem
	Lista de Usuários	Adicionar/Atualizar/Excluir Usuário
	Lista de Estações	Adicionar/Atualizar/Excluir Estação
	Lista de Trens	Adicionar/Atualizar/Excluir Trem
	Lista de Preços	Adicionar/Atualizar/Excluir Preço
	Lista de Configuração	Adicionar/Atualizar/Excluir Configuração

$ícioTabela[h]$

O conjunto de teste de checkout eShopOnContainers envolve preencher o carrinho com um item, navegar até a tela de checkout e verificar se o sistema precisa das informações formatadas para prosseguir com o checkout e fazer o pedido.

[!t]

Teste	Casos de Uso
Login e usuário/senha inválidos,	Login válido/inválido,
	Sair
e credenciais válidas/inválidas, Registro	Credenciais ausentes, Nome de usuário único, Senhas Correspondentes, Data de Expiração Inválida, Conecte-se
Navegar nas páginas e Próxima página,	Página anterior
Pesquisar/Filtrar por marcas,	Pesquisar por tipos, Pesquisa em vários campos
Adicionar Carrinho, Atualizar Carrinho (Salvar/Não Salvar),	Diminuir Carrinho (Salvar/Não Salvar), Remover item
Checkout	População de informações, Check-out válido/inválido

$ço-1emfinal$

$ço-2emfinal$

Tabela **??** e Tabela **??** lista todos os testes TrainTicket e eShopOnContainers Selenium. O número de testes para TrainTicket e eShopOnContainers é muito menor do que a quantidade de casos de uso que abrange cada microsserviço, conforme descrito na Tabela I. Como agrupamos casos de uso semelhantes, acabamos com muito menos testes do que casos de uso.

A paralelização das suítes de teste eShopOnContainers resultou em uma execução mais rápida dos testes. Antes de paralelizar nossos testes de regressão funcional, observamos que o tempo de execução seria em média de 25 segundos. Depois de implementarmos o Html Unit e a estrutura de teste TestNG, observamos que o tempo de execução seria em média de 6 a 7 segundos, que foi o tempo de execução do teste mais lento.

Embora nossos testes ainda sejam relativamente pequenos em tamanho, nosso estudo de caso é uma evidência de que a paralelização de testes de regressão funcional economiza tempo, especialmente para projetos maiores e conjuntos de testes.

B. Teste de Carga

Em nosso estudo de caso de teste de carga, queríamos nos concentrar em testar o tempo de resposta dos vários endpoints nos sistemas de microsserviços. Mais especificamente, queríamos fornecer testes para que a comunidade pudesse ver como esses microsserviços respondem a uma variedade de cargas diferentes. Procuramos oferecer aos interessados uma forma de visualizar os diversos problemas que podem surgir à medida que mais usuários acessam o sistema. Isso inclui ver os possíveis gargalos e o tempo de resposta do sistema.

Conforme elaborado em segundo plano, usamos Gatling (versão 3.9.2) para criar nossos testes de carga para ambos os sistemas. Nas próximas seções, mostraremos nosso processo para projetar e implementar testes baseados em carga enquanto trabalhamos para obter uma lista abrangente de testes avaliando os endpoints em cada microsserviço.

1) Desenho do Estudo

Para rastrear quais endpoints foram testados no microsserviço, criamos um documento de controle de versão para mostrar o progresso de nossos testes. No documento, listamos todos os pontos de extremidade que encontramos no sistema de microsserviço e informações adicionais, como o microsserviço específico em que os pontos de extremidade estavam, o arquivo de teste no qual o ponto de extremidade foi testado e, por último, se o ponto de extremidade foi totalmente coberto ou não . Isso ajudou a fornecer uma estrutura e um plano para os testes futuros. Isso nos permitiu dividir os arquivos de teste com mais eficiência, como basear os testes no microsserviço ao qual cada endpoint estava associado ou em um caso de uso. Este documento foi compartilhado junto com nosso benchmark de teste no Zenodo [**?**].

Para criar esta lista de controle de versão dos endpoints, precisamos preencher manualmente todos os endpoints e serviços dos quais eles fazem parte. Isso levou um tempo decente para criar e acompanhar quais testes testamos completamente. Teria sido mais eficiente e menos dependente de erro humano ter um sistema automatizado que rastreasse os endpoints que testamos analisando o código-fonte do teste. Isso reduziria uma quantidade significativa de tempo gasto para garantir que a lista de controle de versão e nossos testes estivessem sincronizados. No entanto, este processo estava fora dos limites do nosso estudo de caso, por isso não conseguimos implementar este sistema.

Além de criar uma lista de endpoints, também destacamos os principais casos de uso dentro de cada microsserviço. Isso nos ajudou a entender melhor como os usuários normalmente interagem com os microsserviços para ajudar a orientar nossos testes para focar nos casos de uso que o usuário executaria. Esta lista de endpoints está incluída no Zenodo [**?**].

2) Procedimento do Estudo / Enfrentando as Armadilhas Iniciais

Descartamos a ferramenta de gravação fornecida por Gatling. Escrever os testes manualmente nos permitiu desenvolver um código mais conciso e focado que estava fora dos limites do gravador.

Um problema que prevaleceu no início do processo de teste de carga foi garantir a autenticação adequada durante o teste. Quando um usuário efetua login no sistema fornecido, um token de autenticação é gerado, que é usado no cabeçalho das chamadas subsequentes para autenticar que o usuário tem acesso válido a esse endpoint. No entanto, dentro do Gatling, esse token de autenticação geralmente é codificado no cabeçalho e, portanto, torna-se obsoleto nas solicitações subsequentes. Isso causa uma resposta HTTP 403 proibida do sistema, bloqueando a solicitação. Para resolver esse problema, salvamos o token de autenticação que estava no corpo da resposta após concluir a solicitação de login. Você pode salvar uma parte específica do corpo da resposta usando a função `.check(jsonPath("$.token").saveAs("user_token"))` no Gatling. Isso salva o token em uma variável local chamada "`user_token`." A variável pode então ser acessada usando `${user_token}` no cabeçalho da solicitação para fornecer o token de autenticação válido.

Outra decisão importante com o teste de carga é como lidar com os parâmetros do formulário. Parâmetros de formulário podem ser usados em uma variedade de requisições HTTP, mas são associados principalmente com requisições `post`, `put`, e `patch` já que normalmente requerem que informações sejam enviadas para o endpoint. Ao longo de nosso estudo de caso para teste de carga, usamos dois tipos principais de fornecer parâmetros de formulário no Gatling: usando a função `formparam` ou a função `.body(RawFileBody()).`

A função `formparam` é usada para cada parâmetro necessário para o formulário. O formato é `.formparam(parameterName, parameterValue).` Descobrimos que essa opção é melhor quando há um número baixo de parâmetros. O uso dessa função ajuda o testador a ver facilmente todos os parâmetros dentro da chamada de ponto final. Abaixo está um exemplo de solicitação de postagem mostrando o uso da função `formparam` no Gatling. Este exemplo usa referências a variáveis definidas em outro lugar no arquivo de teste.

```
// Exemplo de uso da funo RawFileBody()
.exec(http("Exemplo de solicitao de postagem
    ↪ para login")
    .post("/api/v1/login")
    .formParam("Email", ``${email}")
    .formParam("Senha", ``${senha}")
    .cabealhos(...))
```

A outra opção é usar a função RawFileBody(), que pode ser usada com vários formatos de arquivo. Usamos essa função empregando arquivos JSON. Descobrimos que essa opção é

mais adequada para formulários que exigem um grande número de parâmetros. Isso ocorre porque você pode conter todos os parâmetros em um arquivo externo, em vez de distribuí-los por várias linhas no cenário de teste. Essa opção também é geralmente mais dinâmica do que usar a função formparam porque você pode usar uma variedade de formatos de arquivo diferentes, dependendo de qual formato se adapta melhor à solicitação fornecida. Essa opção envolve o fornecimento de um arquivo JSON dentro da estrutura do projeto que mapeia os nomes dos parâmetros para o valor de cada parâmetro. Abaixo está um exemplo de solicitação de postagem mostrando o uso da função `RawFileBody()` no Gatling.

```
// Exemplo de uso da funo formparam
.exec(http("Exemplo de solicitao de postagem
    ↪ para login")
    .post("/api/v1/login")
    .body(RawFileBody("login_form.json"))
    .cabealhos(...))
```

3) Resultados do Estudo

Antes de discutir os resultados de nossos testes de carga, é importante observar que os testes de carga fornecerão resultados diferentes para diferentes capacidades de servidor. Em nosso estudo de caso, implantamos o sistema de microsserviços por meio de um servidor Ubuntu. Executamos os testes em uma máquina Windows com capacidade de disco de 1 TB, 32 GB de memória e CPU de 6 núcleos de 3,20 GHz. Para o sistema de microsserviço implantado, tínhamos um único seguro para cada microsserviço no sistema.

A partir de nossa análise do microsserviço Train-Ticket, delineamos 240 endpoints em microsserviços. No entanto, apenas 41 desses serviços contêm endpoints. Ao longo de nosso estudo de caso, conseguimos completar a cobertura completa desses endpoints com o melhor de nosso entendimento. Dividimos os testes em 26 arquivos de teste Scala.

As tabelas **??** e **??** detalham o tempo de resposta do respectivo sistema para uma variedade de usuários no sistema para um determinado caso de uso. As colunas mostram o número de usuários durante o teste, o número de solicitações que levaram mais de 800 ms, o total de solicitações durante o teste e a porcentagem das solicitações que levaram mais de 800 ms. Essas informações são usadas para determinar a estabilidade do caso de uso, dados vários níveis de carga de exemplo. Cada carga foi aplicada ao sistema em um intervalo de 30 segundos para isolar a carga como o fator independente. Na seção **??**, discutiremos como formulamos essas métricas de referência.

A tabela **??** mostra os resultados de nosso teste de carga do caso de uso de reserva de passagem em Train-Ticket. Nesse caso de uso, o usuário faz login no sistema, encontra um trem e reserva uma passagem para esse trem. Para fins de isolamento do caso de uso de reserva de passagem, silenciamos as solicitações de cenário de login para que não aparecessem em nossos resultados.

Analisando os resultados do teste abrangendo 100 usuários, o sistema reage bem a esta baixa carga tendo menos de 1% da resposta acima de 800ms. Cargas de 500 e 1000 fazem com que 7,3% e 10,2% do sistema, respectivamente, fiquem acima do limite de 800ms. No entanto, esses resultados ainda passam por nossas especificações de medição, mencionadas na Seção **??**, porque o sistema foi capaz de lidar com a carga específica, pois a porcentagem de solicitações acima de 800ms ficou abaixo de 20%. Os últimos testes abrangendo cargas de 2.500 e 5.000 usuários ficam acima dessas especificações, mostrando assim que o sistema não responde bem a cargas iguais ou superiores a esses valores.

[h]

Número de usuários	**# Maior que 800ms**	**Total de solicitações**	**Porcentagem acima de 800ms**
100 e 3 e 706 e 0,4% 500 e 259 e 3.530 e 7,3% 1000 e 721 e 7.060 e 10,2%			
2500	3.813	17.650	21,6%
5000	15.249	35.300	43,2%

ço-1em final

Para nosso teste de carga do sistema de microsserviço eShopOnContainers, dividimos os testes em 6 arquivos de teste Scala principais. Os casos de uso diminuímos para este sistema porque o sistema tinha principalmente casos de uso para um tipo de usuário. Dividimos os testes com base no caso de uso que eles cobriram.

A tabela **??** apresenta as mesmas informações que a tabela **??**, mas abrange os resultados do caso de uso *checkout do pedido* no sistema de microsserviços eShopOnContainers. Nesse caso de uso, os usuários virtuais fazem login no sistema, acessam o carrinho e fazem o check-out. Semelhante a antes, removemos as solicitações do cenário de login de nossos resultados para isolar o cenário de checkout.

Os primeiros testes com 100 e 500 usuários, respectivamente, mostram uma pequena quantidade de tempo de resposta lento. Isso mostra que o sistema permanece estável durante esses baixos níveis de carga. Este benchmark de teste mostra que há um aumento dramático no tempo de resposta entre 1.000 usuários e 2.500 usuários. Uma carga de 1.000 usuários fica próxima de nossa especificação de medição prescrita de 20%. No entanto, o sistema responde mal ao lidar com 2.500 e 5.000 usuários. Como a maioria das solicitações leva mais de 800 ms para ser finalizada, o sistema não consegue responder com eficiência a cargas de mais de 1.000 usuários.

[h]

Número de usuários	# Maior que 800ms	Total de solicitações	Porcentagem acima de 800ms
100 e 0 e 2.244 e 0,0% 500 e 52 e 11.253 e 0,5% 1000 e 4.276 e 22.479 e 19,0%			
2500	35.952	56.343	63,8%
5000	75.910	112.593	67,4%

ço-1em final

Em ambos os sistemas, notamos que o microsserviço que controla o cenário de login reagiu especificamente mal a grandes quantidades de carga. Provavelmente, isso se deve ao nível de validação usado para verificar as credenciais do usuário. Isso destaca um gargalo importante dentro do sistema para os usuários.

IV. Referência proposta

Com nosso estudo de caso, fornecemos à comunidade um benchmark de teste dos sistemas de microsserviços que avaliamos. A criação de um benchmark de teste permite uma maneira fácil de mostrar à comunidade como os sistemas respondem a uma variedade de testes diferentes. Além disso, eles podem usar o benchmark para simular o sistema usado para realizar análises dinâmicas de sistemas, avaliações de segurança, resiliência ou testes de escala.

Uma chave para testar benchmarks é que eles são repetíveis. Pode servir como ponto de referência para comparar outros produtos e serviços. Esses benchmarks também podem ser usados para comparar os lançamentos de software passados, presentes e futuros com seus respectivos benchmarks. Uma clara evolução dos resultados do software pode ser rastreada usando esses benchmarks.

No geral, existem três componentes principais de benchmark [**?**].

1.] *Especificações de Carga de Trabalho*: Esta área de um benchmark abrange a determinação do tipo e frequência das solicitações a serem submetidas ao sistema em um determinado teste. Dentro do teste de carga, isso seleciona o número geral de usuários em um determinado período de tempo que o teste será executado. As especificações de carga de trabalho para teste de navegador da Web envolvem a avaliação da paralelização de testes.

Especificações de métricas: Este componente centra-se em determinar qual elemento específico de um determinado teste será usado para avaliação. Essas métricas podem ser simplesmente se o teste geral foi aprovado ou reprovado, o tempo de resposta das solicitações, a eficiência dos testes ou alguma outra métrica.

Especificações de medição: O último componente principal de um benchmark de teste é determinar como medir as métricas especificadas para avaliar os resultados. Essa determinação também é indicada como os critérios do Acordo de Nível de Serviço (SLA). Isso pode envolver um determinado limite para o número de solicitações aprovadas em um determinado tempo de resposta ou se todos os testes foram aprovados ou não.

No teste de carga, é importante manter a carga o mais realista possível para a carga de trabalho normal do sistema. Isso dará uma melhor compreensão de como o sistema reage em circunstâncias típicas. Infelizmente, há poucos dados sobre a quantidade normal de usuários que acessam esses sistemas de microsserviços. Com isso em mente, decidimos testar uma variedade de cargas diferentes, começando com um teste básico baixo de 100 usuários e indo até 5.000 usuários como nosso máximo. Isso ajudou a definir nossa especificação de carga de trabalho para 100 usuários, 500 usuários, 1.000 usuários, 2.500 usuários e 5.000 usuários. Para manter a consistência da execução do teste, essas cargas são executadas em um período de 30 segundos. As especificações métricas que utilizamos para medir os resultados foram o tempo de resposta de cada requisição. A cada teste, conseguimos dividir o número de respostas que levaram menos de 800ms, entre 800ms e 1200ms e maiores que 1200ms.

Verificou-se que o tempo de resposta mais preferido para os sistemas é de 0,1 segundos ou 100ms. No entanto, o limite máximo de tempo de resposta aceitável é normalmente definido em 1 segundo ou 1.000 ms [**?**]. Com isso em mente, decidimos definir nosso tempo de resposta aceitável em 0,8 segundos ou 800 ms, uma vez que se prestava bem à ferramenta de teste que usamos. Isso nos leva a determinar que a especificação das medições para os testes de carga seria a porcentagem de requisições que levaram mais de 800ms para serem concluídas. Optamos por utilizar uma porcentagem ao invés de um número específico para auxiliar melhor esta escala métrica em testes com diferentes cargas. Decidimos que um determinado caso de uso foi capaz de lidar com a carga específica se a porcentagem de solicitações que levaram mais de 800ms for menor que 20%

Ao escrever testes de regressão para um navegador da Web, é importante tentar fazer com que os testes sejam executados da maneira mais eficiente possível para reduzir o tempo de execução. Como os testes serão executados muitas vezes após alterações ou atualizações no código, eles precisam ter um tempo de execução baixo. Para diminuir o tempo de execução, utilizamos o framework TestNG para paralelizar os testes do Selenium para os microsserviços. Decidimos dedicar cada teste à sua própria thread para reduzir ao máximo o tempo de teste.

Agrupamos nossos testes de regressão por serviço, de modo que cada teste incluiria muitas afirmações sobre as várias partes do sistema fornecido. Usamos as afirmações individuais como nossas especificações métricas. As afirmações individuais e ações tomadas durante o curso do

teste são o fator determinante para a avaliação.

Para nossos testes de regressão, medimos as afirmações determinando se um teste coletivo passou ou falhou. Se uma das asserções em um teste falhar, isso fará com que todo o teste falhe. Medimos o sucesso do teste apenas se o teste inteiro passou ou falhou.

Após a conclusão, se nosso estudo de caso resultar em benchmark de teste para sistemas de microsserviços específicos, compartilharemos nosso conjunto de testes com o Zenodo [**?**]. Este site é um repositório aberto para os membros da comunidade compartilharem uma variedade de recursos, como artigos de pesquisa, conjuntos de dados, software de pesquisa e relatórios. Compartilhamos o benchmark de teste para permitir que outros pesquisadores e engenheiros da indústria de teste na comunidade interessados em rastrear os resultados dos testes os expandam.

V. Conclusão

Este artigo apresenta um novo conjunto de testes que pode ser efetivamente usado como benchmark para pesquisas sobre teste de software em sistemas baseados em microsserviços. Nossa abordagem considerou tanto o teste de regressão funcional quanto o teste de carga. Selecionamos dois sistemas de microsserviços bem estabelecidos para criar um benchmark de teste para fornecer à comunidade. Esses sistemas de referência foram Train-Ticket e eShopOnContainers.

Nossa abordagem dupla para avaliar o desempenho dos sistemas de microsserviço ajuda a destacar as principais áreas de falha nesses sistemas. O aspecto do teste de carga de nosso estudo de caso destaca áreas dentro dos endpoints que podem causar um gargalo ou falha devido a diferentes quantidades de carga. Compreender como o sistema responde a várias quantidades de carga pode ajudar os designers a entender as áreas para ajudar a melhorar a experiência do usuário. Nosso segundo aspecto de teste envolve a análise do sistema usando testes de regressão funcional.

Especificamente dentro dessa abordagem, usamos testes automatizados do sistema da Web para garantir que o sistema seja exibido de maneira correta e consistente em vários ambientes de navegador.

A contribuição deste trabalho para a comunidade é (1) fornecer um exemplo de código aberto de testes de regressão funcional automatizados e testes de carga para sistemas de microsserviços, pois os exemplos publicados anteriormente não são suficientes e (2) produzir um conjunto inicial de testes para um benchmark proposto de sistemas de microsserviços bem estabelecidos.

References

[1] J. Lewis and M. Fowler, "Microservices," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, 2018.

[3] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *2008 IEEE International Conference on Software Maintenance*, 2008, pp. 307–316.

[4] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015.

[5] Gatling, "Gatling," 2023. [Online]. Available: https://gatling.io/docs/gatling/reference/current/

[6] FudanSELab, "Train-ticket," 2022. [Online]. Available: https://github.com/FudanSELab/train-ticket

[7] N. Application, "eshoponcontainers," 2023. [Online]. Available: https://github.com/dotnet-architecture/eShopOnContainers

[8] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 323–324. [Online]. Available: https://doi.org/10.1145/3183440.3194991

[9] S. Smith, E. Robinson, T. Frederiksen, and T. Stevens, "Case study of test benchmark for microservices: Train-ticket and eshoponcontainers," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7877723

[10] Katalon, "Katalon automation recorder quickstart," 2023. [Online]. Available: https://katalon.com/resources-center/blog/katalon-automation-recorder

[11] G. S. Inc, "Htmlunit," 2023. [Online]. Available: https://htmlunit.sourceforge.io/

[12] C. Beust, "Testng," 2022. [Online]. Available: https://testng.org/doc/

[13] T. Hamilton, "What is benchmark testing?" [Online]. Available: https://www.guru99.com/benchmark-testing.html

[14] ——, "Response time testing - how to measure for api?" 2023. [Online]. Available: https://www.guru99.com/response-time-testing.html