Trabalho de Qualidade e Teste de Software
Entrega 2

Membros do grupo:
João Victor de Albuquerque Pletsch
Victor Correa da Silva Moreira
Márcio de Amorim Machado Ferreira
Pedro Paulo Sobral de Moraes

Escopo do sistema:

A aplicação escolhida é um sistema de gerenciamento de pedidos para um restaurante, que permite que os clientes visualizem o cardápio do restaurante, selecionem produtos disponíveis, adicione-os ao carrinho e finalize o pedido para entrega ou retirada.

Requisitos do Sistema:

1. O sistema deve permitir que os clientes realizem o cadastro para criar uma conta.
2. O cadastro deve solicitar informações como nome, endereço de e-mail e senha.
3. Após o cadastro, os clientes devem poder fazer login usando suas credenciais.
4. Os clientes devem ter a opção de recuperar a senha caso a esqueçam, utilizando um processo seguro de redefinição de senha.
5. O sistema deve permitir que os clientes visualizem o cardápio do restaurante.
6. Cada item do cardápio deve ser acompanhado de uma descrição, preço e imagem ilustrativa.
7. Os clientes devem ser capazes de adicionar produtos ao carrinho de compras.
8. O carrinho de compras deve exibir os itens selecionados, a quantidade de cada produto e o preço total.
9. Os clientes devem ter a opção de remover itens do carrinho antes de finalizar o pedido.
10. O sistema deve permitir que os clientes finalizem o pedido, indicando a forma de pagamento e o endereço de entrega.
11. Após a finalização do pedido, os clientes devem receber uma confirmação do pedido e informações sobre o tempo estimado de entrega.

1. Melhorar e aumentar os casos de testes unitários

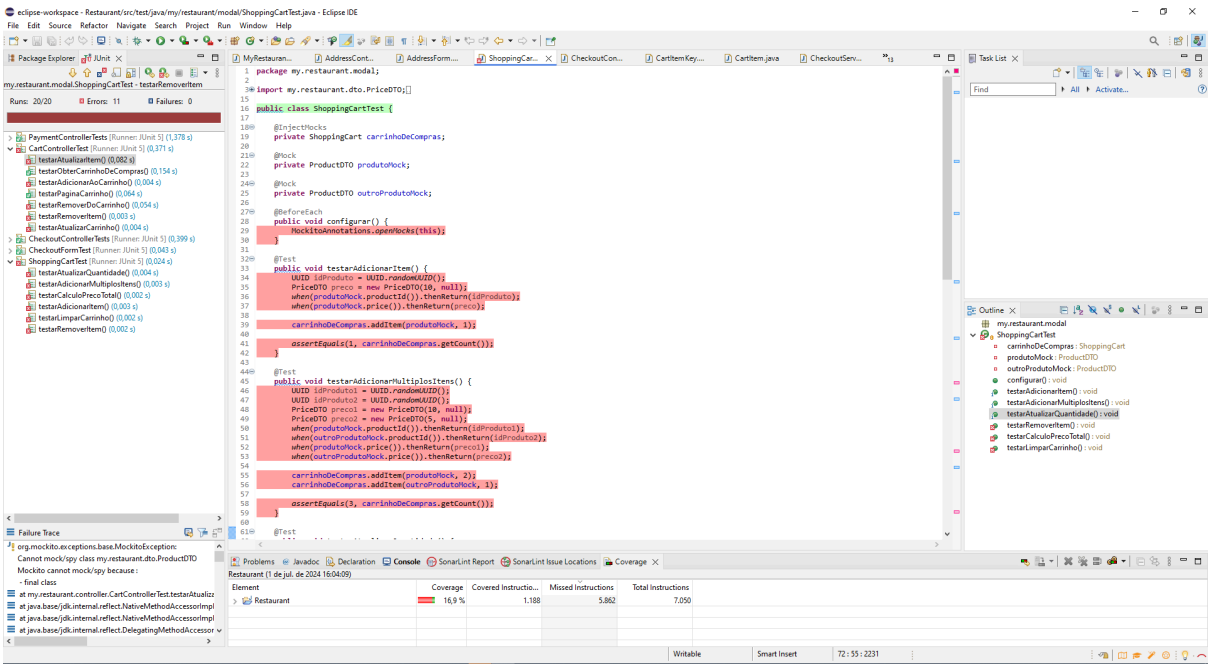
Os testes unitários são fundamentais para validar individualmente as menores unidades de código e poder assegurar que elas funcionam corretamente. A melhoria nos testes existentes focarão em aumentar a cobertura e a eficácia, garantindo que cada unidade seja testada de forma mais isolada. Os casos de testes já criados anteriormente foram para as classes ShoppingCart.java e CartController.java. Para elas, tinham sido criadas as classes de teste ShoppingCartTest.java e CartControllerTest.java.

Realizamos melhorias nos casos de testes destas classes utilizando o framework Mockito para isolar as dependências e melhorar a eficácia e robustez desses testes. Entretanto, infelizmente não conseguimos rodar novamente os testes nas IDEs utilizadas com sucesso,

devido a algum problema acontecendo no projeto que tem mostrado erros pelo código e por isso os testes criados estão também aparecendo com erros. Até o momento não conseguimos resolver, mesmo excluindo o projeto, abrindo e configurando novamente. Portanto, as classes de testes atuais foram melhoradas mas não conseguimos executá-las ainda.

ShoppingCartTest.java melhorada:
https://github.com/vaniacourses/Restaurant/blob/main/src/test/java/my/restaurant/modal/ShoppingCartTest.java

CartControllerTest.java:melhorada:
https://github.com/vaniacourses/Restaurant/blob/main/src/test/java/my/restaurant/controller/CartControllerTest.java



Problema no projeto impedindo que os testes sejam executados.

2. Implementar testes de sistema considerando requisitos funcionais

Para garantir que o sistema atenda aos seus requisitos funcionais, foram implementados testes de sistema usando a ferramenta de automação Selenium. Esses testes foram escritos em Python e executados no navegador Google Chrome. Os scripts de teste estão organizados na estrutura do projeto na pasta tests > system_tests > selenium > code. Nessa pasta foram criados 4 scripts para testar diferentes funcionalidades do sistema:

- “adicionar_produto_carrinho.py”:

Objetivo: testar a funcionalidade de adicionar um produto ao carrinho.
Ação: o script abre a página inicial do sistema e simula a ação de um usuário adicionando um produto do cardápio ao carrinho.

- “cadastro_teste.py”:

Objetivo: verificar o processo de cadastro de novos usuários.
Ação: preenche os campos necessários para o cadastro (e-mail, primeiro e último nome, senha) e envia o formulário, validando a criação bem-sucedida da conta.

- “login_teste.py”:

Objetivo: testar o processo de login do sistema.
Ação: insere as credenciais de usuário e verifica se o sistema realiza o login corretamente e redireciona o usuário para a tela Home.

- “finalizar_pedido_teste.py”:

Objetivo: automação parcial do processo de finalização de pedido.
Ação: o script começa abrindo a página inicial, e em seguida, o testador manualmente precisa adicionar um produto ao carrinho no site, e após isso ele precisa clicar em alguma tecla no terminal para que o script continue a execução, que redireciona para a página de checkout, preenchendo os dados pessoais e de pagamento (como nome, endereço e informações do cartão de crédito), e por fim confirma o pedido e verifica se uma página de sucesso com o número do pedido é exibida.

2.1 Para englobar os requisitos **funcionais**, utilizamos a técnica de **análise de valor limite** trazendo mais robustez para os nossos testes de sistema. Adicionamos mais dois scripts, sendo eles: “*cadastro_teste_limite_superior.py”* e “*cadastro_teste_limite_inferior.py”,* cada um para testar o seu respectivo limite.

Notamos que o limite inferior dos campos firstName, lastName e password é 0 pois são aceitos cadastros de 1 caractere por exemplo, e o limite superior para os mesmo campos é de 101, sendo possível cadastro de até 100 caracteres.

3. Indicação das medidas de cada atributo de qualidade da ISO 25010 seguindo uma escala.

Para o sistema de restaurante do nosso projeto, fizemos um ranking das 10 características e sub-características dos atributos de qualidade da ISO 25010, de como o sistema deve ser preferencialmente, listadas por ordem de maior prioridade:

1 - Correção Funcional

A correção funcional será fundamental para garantir que todas as operações críticas do sistema, como o cálculo de pedidos e a aplicação de descontos, sejam realizadas de maneira mais precisa e sem erros. Algum erro nisso poderia resultar em cobranças

incorretas, o que poderia frustrar os usuários e impactar negativamente a confiabilidade do serviço.

Para medir a correção funcional, poderia utilizar métricas como a taxa de erro em transações e a taxa de conformidade com os requisitos. A taxa de erro nas transações indica o percentual de transações que contêm erros de cálculo ou processamento, sendo mais ideal buscar um valor próximo a 0%. Já a taxa de conformidade com os requisitos avaliaria quantas funcionalidades operam de acordo com as especificações estabelecidas com o objetivo de atingir 100% de conformidade. Além disso, implementar testes automatizados e revisar os logs de transações são medidas bem eficazes para monitorar e manter essa correção, garantindo uma melhor precisão.

2 - Eficiência de Desempenho

A eficiência de desempenho seria muito importante para que o sistema responda rapidamente às ações dos usuários e opere de maneira eficiente em todos os horários, ainda mais que no sistema do restaurante, períodos de alta demanda como horários de refeição ou durante promoções especiais poderia causar uma grande carga no sistema.

Poderia se observar métricas como o tempo de resposta da página e o número de usuários simultâneos suportados. Esse tempo de resposta das páginas deve ser minimizado a fim de garantir que as principais, como a inicial e de cardápio, carreguem rapidamente e assim podendo manter os usuários engajados. Além disso, técnicas como de caching, compressão de recursos e alguns testes de carga poderiam ajudar para otimizar o desempenho e assegurar uma operação suave do sistema, mesmo em picos de uso.

3 - Disponibilidade

A disponibilidade vai assegurar que o sistema esteja operacional e acessível aos usuários sempre. Se o sistema estiver fora do ar durante horários de serviço ou picos de pedidos poderia causar uma perda de vendas e a insatisfação do cliente.

Para medir a disponibilidade, as métricas de percentual de uptime e tempo médio de recuperação são bem cruciais, sendo que o percentual de uptime deve ser o mais próximo possível de 100%, indicando que o sistema está quase sempre disponível e funcional, e o tempo médio de recuperação mede a rapidez com que ele é restaurado após uma falha, com o objetivo de ser o menor possível para minimizar o impacto das interrupções. Também um monitoramento contínuo e manutenção preventiva poderiam ser boas práticas para manter a alta disponibilidade, reduzindo o tempo de inatividade e melhorando a confiança dos clientes.

4 - Usabilidade

A usabilidade vai garantir que os usuários possam navegar e operar o sistema de maneira mais intuitiva e eficiente, e para esse sistema, essa usabilidade seria muito importante para que os clientes possam encontrar e pedir pratos facilmente, personalizar os seus pedidos e

finalizar compras sem problemas, e com uma interface intuitiva e fácil de usar aumentaria a satisfação do cliente e a probabilidade de completar uma compra.

Para avaliar a usabilidade, daria para considerar o tempo de conclusão de tarefas, a taxa de abandono de carrinho e uma pontuação de satisfação do usuário. O tempo de conclusão de tarefas deve ser o menor possível, indicando que eles possam realizar ações essenciais rapidamente e sem dificuldades. Já a taxa de abandono de carrinho deve ser minimizada o máximo possível, e a pontuação de satisfação do usuário, que pode ser medida em uma escala de 1 a 5, deve ser alta, indicando uma experiência positiva com o sistema. Também alguns testes de usabilidade e um feedback regular dos usuários poderiam ser fundamentais para identificar áreas de melhoria e otimizar a experiência.

5 - Segurança

A segurança seria essencial para proteger os dados sensíveis dos clientes, como informações pessoais e os detalhes de pagamento. Uma violação de segurança poderia ter consequências graves, como a perda de dados, danos à reputação e penalidades legais com a empresa do sistema.

Para medir essa segurança poderia ser usado métricas como o número de incidentes de segurança e o tempo médio de resolução de incidentes. O número de incidentes deve ser mantido ao mínimo possível, e o tempo médio de resolução de incidentes deve ser o menor possível também, para garantir sempre uma resposta rápida e eficiente a quaisquer ameaças que possam ocorrer. Além disso, poderia ser implementado políticas de segurança mais robustas como criptografia por toda essa parte sensível de dados no sistema e um monitoramento contínuo.

6 - Interoperabilidade

A interoperabilidade vai poder assegurar que o sistema possa interagir de forma eficaz com outros sistemas, como os gateways de pagamento e serviços de entrega que o sistema possa utilizar.

Para avaliar, poderia considerar o número de integrações bem-sucedidas e o tempo médio de implementação de novas integrações. O número de integrações bem-sucedidas deve ser maximizado, indicando que o sistema pode se conectar eficazmente com vários serviços externos, e o tempo médio de implementação de novas integrações deve ser minimizado também, permitindo que ele responda rapidamente a novas oportunidades de negócios e necessidades operacionais, por exemplo. Além disso, estabelecer processos claros para o desenvolvimento e teste de integrações com APIs externas seria fundamental para garantir que essa interoperabilidade seja mais eficaz ainda.

7 - Analisabilidade

A analisabilidade facilitaria muito a identificação e o diagnóstico de problemas pelo sistema, permitindo correções mais rápidas e eficientes, e uma alta analisabilidade permitiria que a

equipe de suporte técnico, por exemplo, mantenha o sistema funcionando corretamente e reduza o tempo de inatividade.

Para medir isso poderia ser usado o tempo médio de diagnóstico e o percentual de problemas resolvidos na primeira tentativa. O tempo médio de diagnóstico deve ser o menor possível, para facilitar a identificação rápida de problemas, e o percentual de problemas resolvidos deve ser maximizado, refletindo assim a eficácia da equipe de suporte técnico e a qualidade da documentação do sistema. E também o uso de ferramentas de monitoramento e logging poderiam ser essenciais para fornecer insights mais detalhados sobre o desempenho e ajudar na rápida detecção e resolução desses problemas.

8 - Adaptabilidade

A adaptabilidade seria importante para que o sistema seja facilmente ajustado para diferentes ambientes e requisitos, como novos dispositivos ou mudanças nas políticas de negócios, por exemplo. Uma capacidade de adaptação seria essencial para responder rapidamente a novas tendências e demandas do mercado, como a adição de novos métodos de pagamento.

Para avaliar essa adaptabilidade daria para considerar o tempo médio para adaptar o sistema a um novo ambiente e o número de plataformas suportadas, com esse tempo médio para adaptar o sistema devendo ser minimizado e permitindo ajustes rápidos e eficientes para novos requisitos ou dispositivos. E o número de plataformas suportadas deve ser maximizado para que o sistema possa operar em mais dispositivos sem a necessidade de modificações muito significativas. Além disso, algumas práticas de desenvolvimento modular e uma arquitetura flexível são importantes para suportar essa adaptabilidade e garantir que o sistema possa evoluir sempre.

9 - Comportamento Temporal

O comportamento temporal seria importante para avaliar como o sistema responde de maneira oportuna às interações dos usuários, que seria muito importante para manter uma experiência de usuário mais fluida e eficiente. Portanto, principalmente durante o processo de pedido e pagamento, um bom comportamento temporal seria um fator essencial, podendo com isso reduzir a probabilidade de abandono devido à lentidão, por exemplo.

Para medir isso poderia ser observado a latência de processamento de pedidos e o tempo de carregamento da página inicial. Essa latência deve ter o menor tempo possível para garantir que os pedidos sejam processados rapidamente, desde a seleção dos itens até a confirmação da compra, e o tempo de carregamento da página inicial deve ser o mais curto possível também. Além dessas duas, algumas técnicas como otimização no back-end, compressão de recursos e caching poderiam ajudar também.

10 - Testabilidade

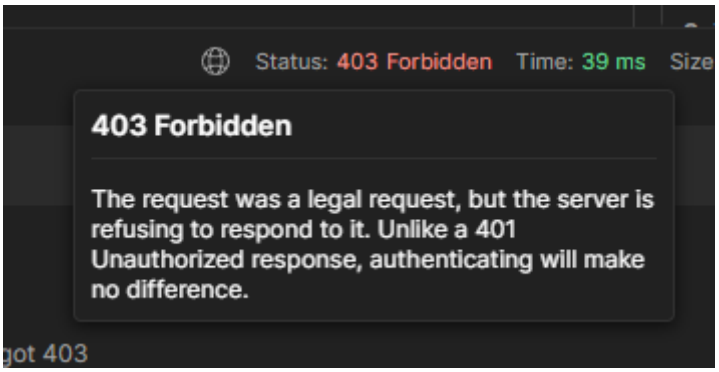
A testabilidade seria uma qualidade importante para garantir que o sistema possa ser testado de maneira eficaz, verificando se todas as suas funções operam conforme

esperado, além de que novas funcionalidades possam ser introduzidas com confiança e mudanças no código não comprometam o funcionamento geral. Isso poderia facilitar até a manutenção e evolução do sistema, permitindo respostas sempre mais rápidas e uma detecção precoce de falhas potenciais.

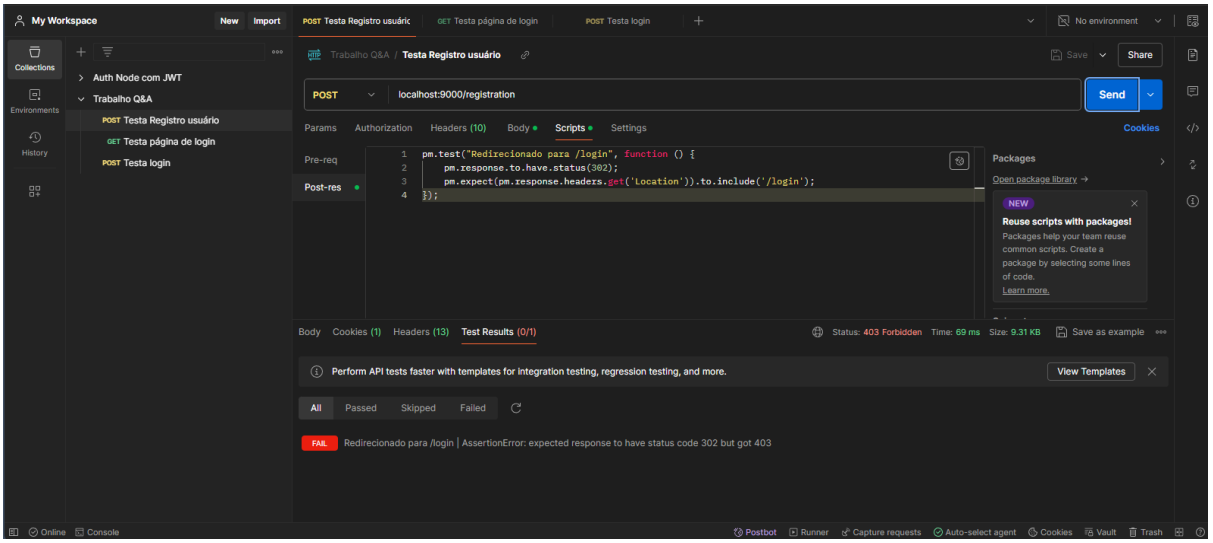
Para medir isso poderia ser utilizado uma cobertura de testes automatizados, que representaria o percentual de código coberto por testes automatizados. Por exemplo, 85% desse código sendo coberto. A escala deve ser de 0 a 100%, sendo que quanto maior, melhor. E para alcançar isso seria preciso desenvolver e manter uma suíte abrangente de testes que cubram as funcionalidades críticas e ao menos os principais fluxos de usuário.

4. Implementar casos de teste de integração

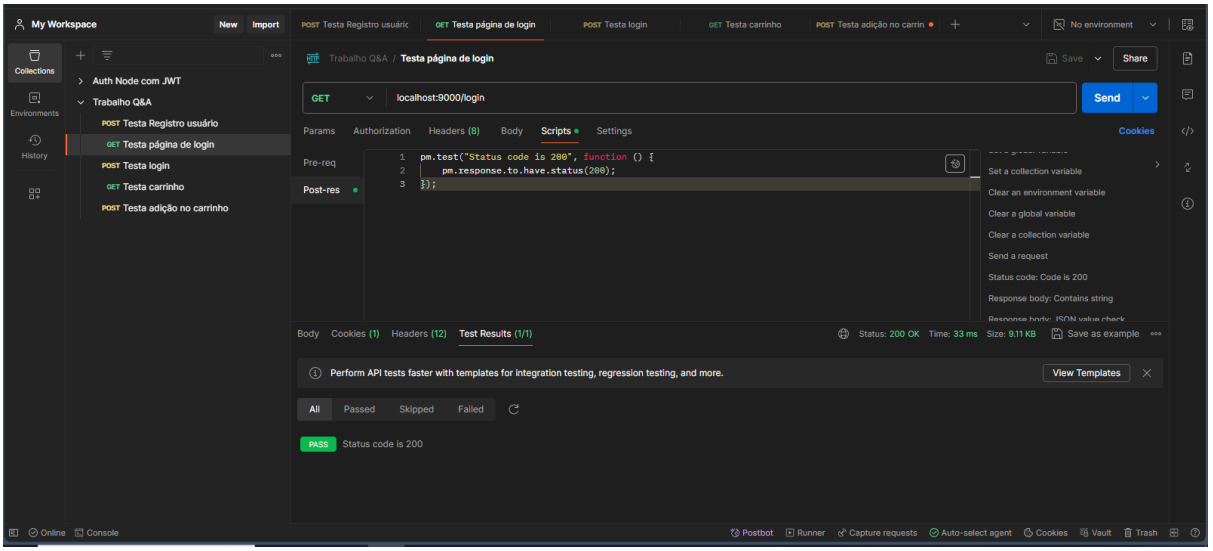
Utilizamos o Postman para fazer os testes de integração. Porém, por conta da aplicação utilizar o token CSRF (Cross-Site Request Forgery), e ele é frequentemente enviado como um cabeçalho HTTP com cada solicitação, não conseguimos realizar os testes de POST por não conseguir gerar esses tokens manualmente.



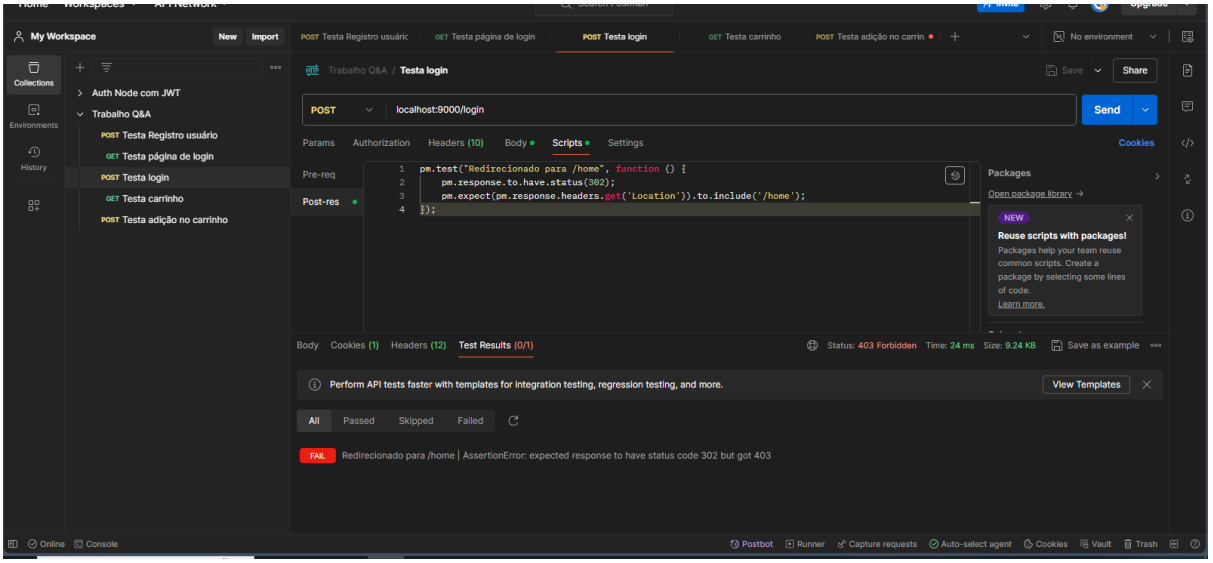
4.1 Sem autorização



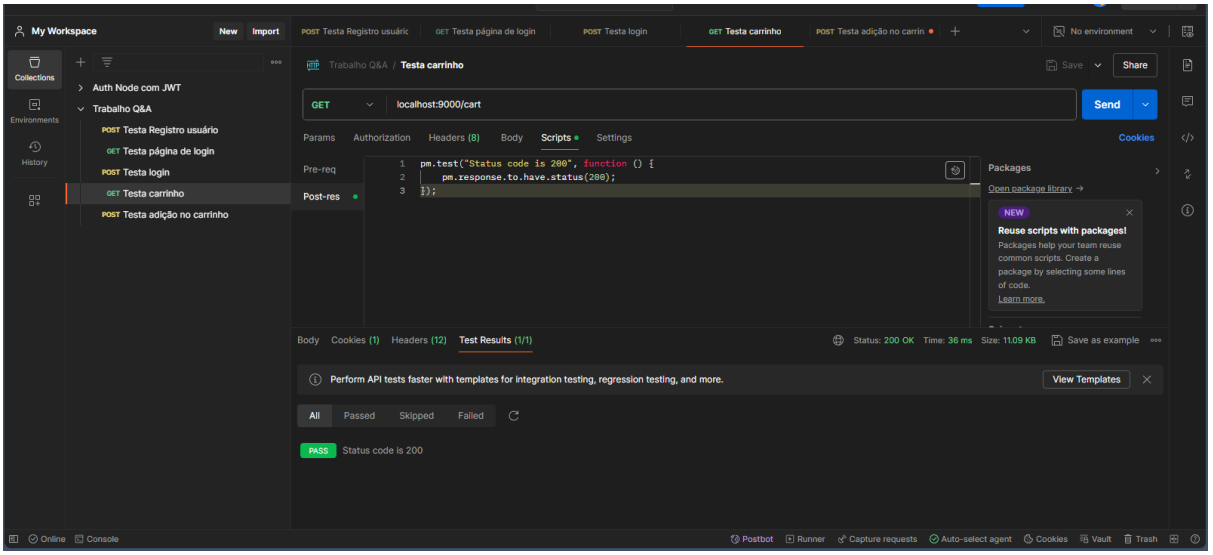
4.2 Teste registro usuário



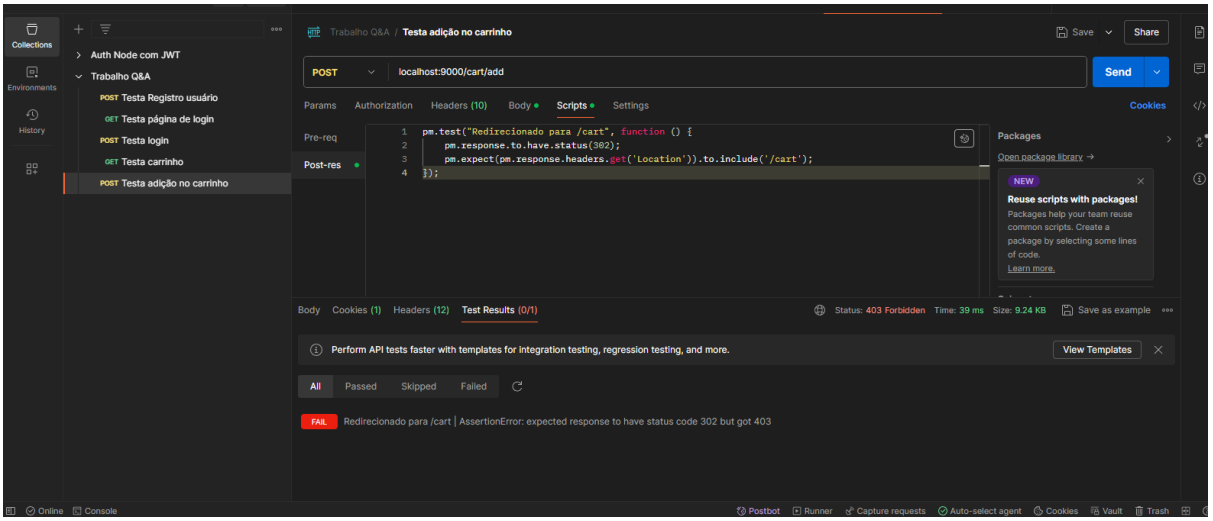
4.3 Teste página de login



4.4 Teste login



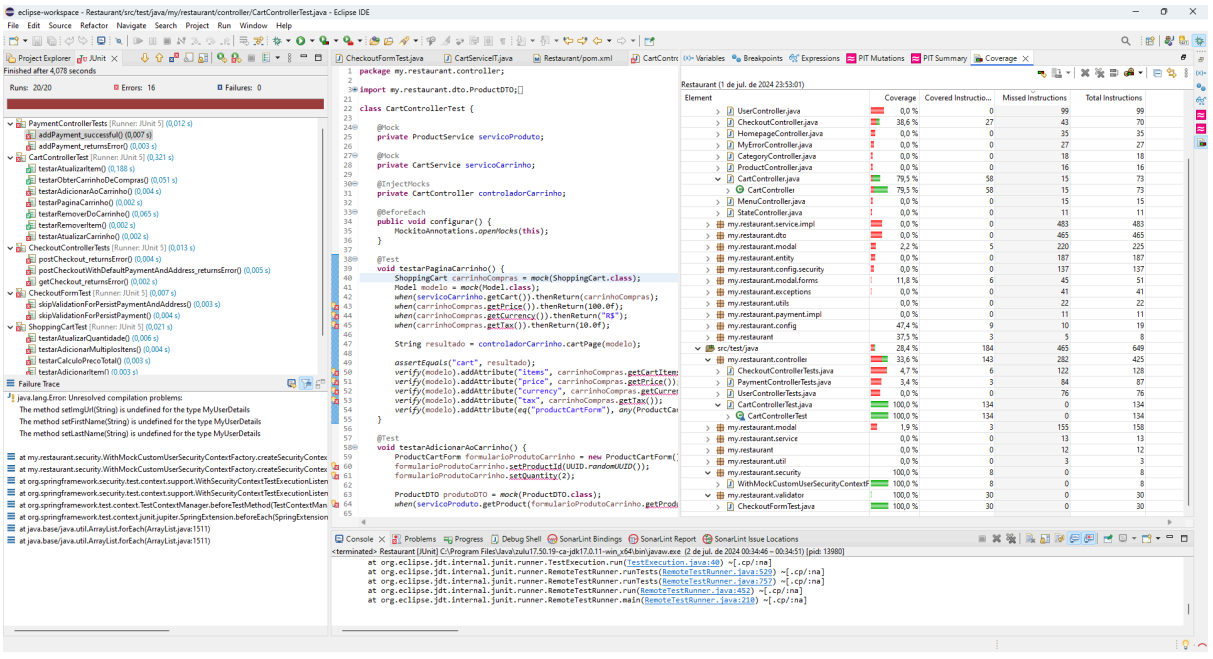
4.5 Teste carrinho



4.6 Teste adição no carrinho

5. Projetar e melhorar o conjunto de casos de teste

Alcançamos a cobertura de 80% da classe de carrinho de compras. Por algum motivo, a IDE do Eclipse está identificando vários erros no código, coisa que o VSCode não encontra, e por isso dificultou bastante alcançar maiores taxas de coberturas em outras classes. Focamos na classe de carrinho para tentar resolver estes problemas.



CartController.java	79,5 %	58	15	73
CartController	79,5 %	58	15	73
MenuController.java	0,0 %	0	15	15
StateController.java	0,0 %	0	11	11
my.restaurant.service.impl	0,0 %	0	483	483
my.restaurant.dto	0,0 %	0	465	465
my.restaurant.modal	2,2 %	5	220	225
my.restaurant.entity	0,0 %	0	187	187
my.restaurant.config.security	0,0 %	0	137	137
my.restaurant.modal.forms	11,8 %	6	45	51
my.restaurant.exceptions	0,0 %	0	41	41
my.restaurant.utils	0,0 %	0	22	22
my.restaurant.payment.impl	0,0 %	0	11	11
my.restaurant.config	47,4 %	9	10	19
my.restaurant	37,5 %	3	5	8
src/test/java	28,4 %	184	465	649
my.restaurant.controller	33,6 %	143	282	425
CheckoutControllerTests.java	4,7 %	6	122	128
PaymentControllerTests.java	3,4 %	3	84	87
UserControllerTests.java	0,0 %	0	76	76
CartControllerTest.java	100,0 %	134	0	134
CartControllerTest	100,0 %	134	0	134

Para melhor visualização.

-Adicionado após a apresentação-

Conseguimos utilizar o eclipse para criar mutantes com alguma dificuldade, porém não conseguimos matar estes mutantes pelo eclipse devido a muitos erros da IDE com nosso código(Falamos em sala que o Eclipse estava identificando erros inexistentes).
Além disso, a "*ShoppingCartTest.java*", uma das classes de testes que fizemos na entrega 1, não estava aparecendo nos relatórios dos testes de mutação, e não conseguimos descobrir o motivo.

Print do Eclipse:

Pit Test Coverage Report

Package Summary

my.restaurant.controller

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
12	0%	0/174	0%	0/55	0%	0/0

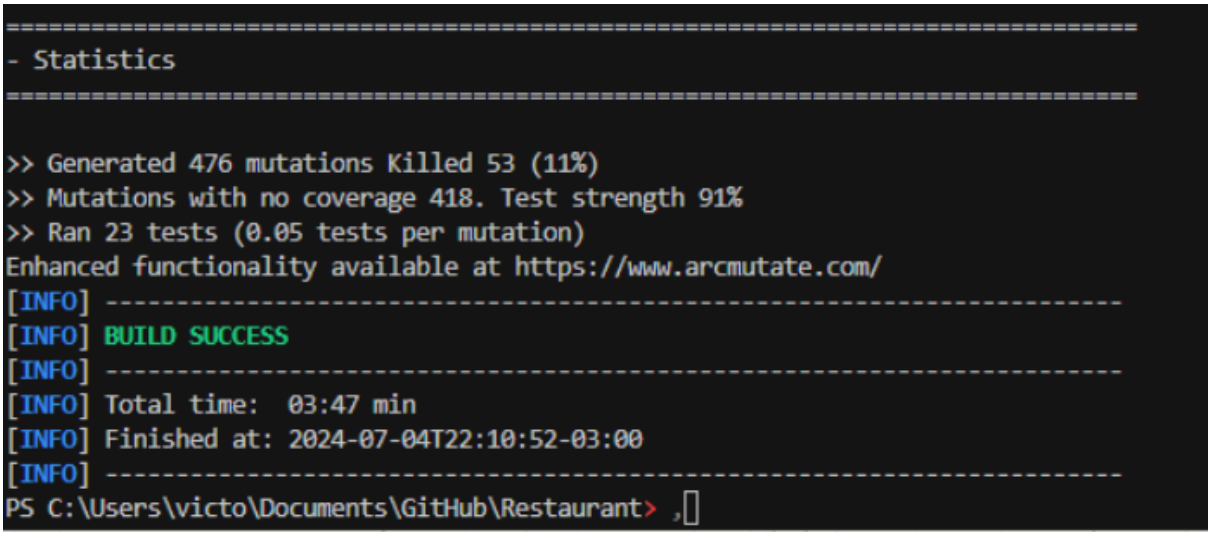
Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
AddressController.java	0%	0/27	0%	0/8	0%	0/0
CartController.java	0%	0/18	0%	0/5	0%	0/0
CategoryController.java	0%	0/6	0%	0/1	0%	0/0
CheckoutController.java	0%	0/18	0%	0/2	0%	0/0
HomepageController.java	0%	0/9	0%	0/1	0%	0/0
MenuController.java	0%	0/5	0%	0/1	0%	0/0
MyErrorController.java	0%	0/9	0%	0/6	0%	0/0
OrderController.java	0%	0/17	0%	0/4	0%	0/0
PaymentController.java	0%	0/32	0%	0/15	0%	0/0
ProductController.java	0%	0/5	0%	0/1	0%	0/0
StateController.java	0%	0/4	0%	0/1	0%	0/0
UserController.java	0%	0/24	0%	0/10	0%	0/0

Report generated by PIT 1.6.8

Como o VSCode está executando os testes perfeitamente pensamos em fazer os mutantes por lá e ficou assim.

Print do VSCode:



Infelizmente demora muito e não tem uma visualização muito boa, mas criamos algumas formas de matar alguns mutantes criados na classe "*CartController.java*"

O commit com as mudanças na classe de teste foi feito no gitHub.

Responsabilidades:

Entrega 2 (Peso 5): Realizar no mínimo:

- Melhorar e aumentar os casos de testes unitários, isolando suas dependências - João Victor, Victor e Pedro Paulo
- Implementar casos de teste de integração - Pedro Paulo e Victor
- Indicação das medidas de cada atributo de qualidade da ISO 25010 seguindo uma escala. Justificar as decisões para indicação das medidas. - Victor, João Victor, Pedro Paulo, Márcio (fizemos o ranking em sala de aula e depois escrevemos a documentação)
- Implementar testes de sistema considerando requisitos funcionais E pelo menos um atributo de qualidade (último opcional) - João Victor, Victor e Pedro Paulo
- Projetar e melhorar o conjunto de casos de teste, utilizando as técnicas: - Victor e Márcio
    - Funcional
    - Estrutural (ao menos 80% de cobertura no critério todas-arestas das classes não CRUD da entrega 1)
    - Baseada em Defeitos (80% de escore de mutação nas mesmas classes do teste estrutural das classes não CRUD da entrega 1)

- Apresentação (slides): Márcio, João Victor, Pedro Paulo e Victor