

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

PUC Minas Virtual

Pós-graduação *Lato Sensu* em Arquitetura de *Software* Distribuído

Projeto Integrado

Relatório Técnico

SGC – Sistema de Gerenciamento Comercial

Lucas Figueiredo

Aracaju
Jun/2023

Lucas Daniel Leite Figueiredo

Relatório Técnico: SGC – Sistema de Gerenciamento Comercial

Projeto Integrado apresentado ao Programa
de Pós-graduação *Lato Sensu* em Arquitetura
de *Software* Distribuído da Pontifícia
Universidade Católica de Minas Gerais

Orientador: Luiz Alberto Ferreira Gomes

Aracaju
Jun/2023

RESUMO

Este projeto apresenta a solução adotada para o desenvolvimento de um sistema de e-commerce, abordando a sua arquitetura, os motivos que levaram a sua escolha, a forma como ela foi implementada, bem como suas vantagens e desvantagens. O *e-commerce* é a comercialização de produtos e serviços através da internet, uma modalidade de comércio em constante ascensão e com previsão de uma movimentar cerca de 7 trilhões de dólares já em 2025.

Muitas empresas passaram a adotar esse tipo de comércio em suas atividades, seja se utilizando das redes sociais como ferramentas para divulgação, marketing e comercialização de seus produtos, seja via sites de Marketplace ou através de suas próprias lojas virtuais.

O sistema em questão – o SGC – visa fornecer uma solução para empresas que estão buscando estabelecer as suas lojas virtuais, disponibilizando uma aplicação web com API que realiza o gerenciamento de clientes, produtos, estoque e vendas, e pode ser integrado à aplicações clientes como por exemplo “Single-Page Applications” e aplicativos móveis.

ABSTRACT

This work presents the solution for the development of an e-commerce application, addressing its architecture, why it was chosen, how it was implemented and its advantages and disadvantages.

E-commerce refers to the commercial transactions of services and goods that happen through the internet, a rapidly growing form of business that is expected to generate around 7 trillion dollars by 2025.

There are a lot of companies that embraced this type of business, be it using social media for marketing and sales, or through marketplace websites or their own online store.

The application we're presenting – the SGC – aims to provide a solution for companies looking to establish their online stores, offering an API Web Application which help them manage customers, products, inventory and sales. This application can be integrated with any kind of client application, such as Single Page Web Applications or even Mobile Applications.

Índice de Imagens

Figura 1: Topologia de uma Arquitetura em Camadas (Sommerville, 2009).....	12
Figura 2: Diagrama de Contexto da Aplicação SGC.....	19
Figura 3: Diagrama de Contêiner – Aplicação SGC.....	20
Figura 4: Diagrama de Componentes – Arquitetura Geral do Sistema – Aplicação SGC.....	21
Figura 5: Tela Inicial para criação de um novo projeto com o Framework .NET no Visual Studio.....	22
Figura 6: Diagrama de Componentes – API Web do ASP.NET Core – Aplicação SGC.....	23
Figura 7: Trecho do código de implementação do Hosted Service para a classe abstrada BaseConsumer.....	24
Figura 8: Trecho de Código com a Implementação do Consumidor de Mensagens “ClienteAtualizadoConsumer”.....	25
Figura 9: Diagrama do fluxo da operação de venda na aplicação SGC.....	26
Figura 10: Diagrama de Componentes – Módulo de Clientes da Aplicação SGC.....	27
Figura 11: Organização interna dos componentes do Módulo de Clientes.....	28
Figura 12: Diagrama de Componentes - Módulo Core.....	30
Figura 13: Configuração de Esquema de autenticação a ser utilizado pela aplicação Web.....	34
Figura 14: Configuração das Roles e Policies para Validação da Autorização do Usuário na Aplicação Web.....	35
Figura 15: Configuração de Roles para o usuário Cliente-1.....	35
Figura 16: Exemplo de configuração de Roles para um usuário do tipo Funcionário.....	35
Figura 17: Tentativa de requisição ao endpoint ext/cliente por um usuário não autenticado.....	36
Figura 18: Janela para autenticação do usuário no Keycloak.....	36
Figura 19: Requisição de cliente para acessar seus próprios dados na aplicação Web.....	37
Figura 20: Requisição de cliente para acessar dados de todos os clientes (apenas funcionários devem ter acesso a esse endpoint).....	37
Figura 21: Lista de Módulos que possuem referência à propriedade Endereço da entidade Cliente.....	39
Figura 22: Lista de arquivos que sofreram alteração na inclusão da propriedade Endereço na entidade Cliente através do comando Gitk do Git.....	39
Figura 23: Representação de Cliente para o módulo de Clientes – Com propriedade Endereço.....	40
Figura 24: Representação de Cliente no módulo de Vendas – Sem a propriedade de Endereço, já que esta não é relevante para as operações realizadas neste módulo.....	40
Figura 25: Disparo da solicitação da confirmação de uma venda.....	42
Figura 26: Atualização de status do produto para INATIVO.....	43
Figura 27: Registro das tentativas de baixa de estoque (registro de log em console do docker).....	43
Figura 28: Notificação informando que a reserva do produto não foi concluída com êxito.....	43
Figura 29: Registro da Venda com status atualizado para Reprovado, visto que o sistema não conseguiu realizar a reserva do produto.....	44
Figura 30: Solicitação de processamento por parte do cliente após status do produto estar atualizado no módulo de Vendas.....	44
Figura 31: Venda gerada pelo cliente.....	46
Figura 32: Requisição de confirmação de venda.....	46

Projeto Integrado – Arquitetura de Software Distribuídos

Figura 33: Registro em log com a requisição do cliente confirmando a operação de venda.....	47
Figura 34: Registro da compra solicitada pelo cliente com o status PROCESSANDO.....	47
Figura 35: Registros da reserva do produto após confirmação da venda.....	48
Figura 36: Registro de venda com status AGUARDANDO PAGAMENTO após o produto ser reservado em estoque....	48
Figura 37: Mensagem recebida pela aplicação web informando que o pagamento não foi efetuada com sucesso.....	49
Figura 38: Comando enviado pela aplicação web para que o módulo de vendas atualize o status da Venda em questão.	49
Figura 39: Venda com status de reprovada após comunicação de que o pagamento não foi efetuado com sucesso.....	49

Índice de tabelas

Tabela 1: Requisitos Funcionais do Sistema (B – baixo, M – médio, A – alto).....	14
Tabela 2: Requisitos Não-funcionais do Sistema (B – baixa, M – média, A – alta).....	15
Tabela 3: Restrições Arquiteturais do Sistema.....	16
Tabela 4: Mecanismos Arquiteturais do Sistema.....	17
Tabela 5: Atributos de Qualidade e Cenários Utilizados na Avaliação do SGC.....	31
Tabela 6: Cenário 1 de avaliação da arquitetura – Segurança.....	33
Tabela 7: Cenário 2 de avaliação de arquitetura – Manutenibilidade.....	38
Tabela 8: Cenário 3 da avaliação da arquitetura – Resiliência.....	42
Tabela 9: Cenário 4 da avaliação da arquitetura – Monitoramento.....	45

Sumário

1 INTRODUÇÃO.....	8
2 ESPECIFICAÇÃO ARQUITETURAL DA SOLUÇÃO:.....	10
2.1 CONTEXTO DO PROJETO:.....	10
2.2 ESTILO ARQUITETURAL:.....	11
2.3 REQUISITOS FUNCIONAIS:.....	12
2.4 REQUISITOS NÃO-FUNCIONAIS:.....	13
2.5 RESTRIÇÕES ARQUITETURAIS:.....	15
2.6 MECANISMOS ARQUITETURAIS:.....	15
3 MODELAGEM ARQUITETURAL:.....	17
3.1 DIAGRAMA DE CONTEXTO:.....	18
3.2 DIAGRAMA DE CONTÊINER:.....	19
3.3 DIAGRAMA DE COMPONENTES:.....	20
3.3.1 DIAGRAMA DE COMPONENTES – API Web do ASP.NET Core.....	21
3.3.2 DIAGRAMA DE COMPONENTES – Módulos.....	25
3.3.3 DIAGRAMA DE COMPONENTES – CORE.....	28
4 AVALIAÇÕES DA ARQUITETURA (ATAM).....	30
4.1 ANÁLISE DAS ABORDAGENS ARQUITETURAIS.....	30
4.2 CENÁRIOS.....	31
4.3 EVIDÊNCIAS DA AVALIAÇÃO.....	31
4.3.1 CENÁRIO 1 – SEGURANÇA.....	31
4.3.2 CENÁRIO 2 – MANUTENIBILIDADE.....	33
4.3.3 CENÁRIO 3 – RESILIÊNCIA.....	35
4.3.4 CENÁRIO 3 – MONITORAMENTO.....	37
5 AVALIAÇÃO CRÍTICA DOS RESULTADOS.....	39
6 CONCLUSÃO.....	41
7 REFERENCIAS BIBLIOGRÁFICAS.....	42

1 INTRODUÇÃO

E-commerce (ou comércio eletrônico) envolve o uso de meios digitais (internet, web, dispositivos móveis...) na realização de transações comerciais (produtos ou serviços) que envolvem troca de valor entre indivíduos e organizações (LAUDON, 2017).

A maioria das empresas que trabalham nessa área se utilizam de lojas virtuais e/ou plataformas online para conduzir as atividades como o marketing de seus produtos, o processo de venda e o gerenciamento da logística para entrega do produto ao seu destino final.

De acordo com o *eMarketer*, empresa de pesquisa de mercado e consultoria estadunidense que realiza estudos e análises sobre o comportamento e tendência do mercado digital, as vendas através de plataformas de e-commerce movimentará mais de 5 trilhões de dólares pela primeira vez na história, contabilizando mais de 1/5 (um quinto) do total de vendas registradas mundialmente. A expectativa é que esse número cresça ainda mais, atingindo a marca de 7 trilhões de dólares em 2025.

Laudon (2017) diz ainda que as empresas que trabalham com e-commerce podem ser classificadas de acordo com a natureza da relação de mercado – quem está vendendo para quem. Algumas dessas modalidades de e-commerce estão listadas abaixo:

- Business-to-Consumer (B2C): O modelo mais conhecido, normalmente envolve a venda/fornecimento de produtos, serviços ou até de conteúdo digital de uma empresa para um cliente.
- Business-to-Business (B2B): É o tipo de modalidade que envolve a transação comercial entre duas empresas, sendo a maior (em termos de quantidade de transações e valor monetário movimentado).
- Customer-to-Customer(C2C): Nessa modalidade há a interação entre clientes, onde um deles cumpre o papel de vendedor e o outro de comprador. Essa é a modalidade comumente se utiliza de plataformas de venda, como eBay ou OLX, para facilitar a busca por produtos, interação entre comprador e vendedor, bem como a transação comercial de fato.

Na modalidade B2C é comum vermos empresas se utilizando de diferentes canais de venda, como redes sociais, plataformas de marketplace (Amazon e Mercado Livre) ou através de suas próprias lojas virtuais.

A Exame em seu artigo “O que é e-commerce e para que serve?” apresenta algumas das vantagens de cada um desses canais, como o fato de que redes sociais possuem um público consolidado e toda estrutura necessária para realização das vendas, o grande fluxo de usuários em sites de Marketplace – que geralmente são a primeira opção na busca de produtos – e como as lojas virtuais dão mais autonomia às empresas na venda de seus produtos, com a possibilidade de geração de relatórios de vendas, gerenciamento de usuários e controle de estoque, que auxilia na tomada de decisões de negócio a partir de métricas geradas pelo próprio sistema.

Lojas virtuais, no entanto, possuem um custo de desenvolvimento e manutenção mais altos que as demais modalidades citadas. Esse custo, por sua vez, pode ser ainda maior caso o projeto não

siga qualquer tipo de padrão ou boas práticas, resultando em um sistema mal projetado, mal otimizado e que pode se tornar difícil de gerir com o passar do tempo.

O objetivo desse trabalho é apresentar uma aplicação API Web para gerenciamento de uma loja online para empresas que desejam migrar para esse tipo de modalidade. O sistema deve atender as necessidades básicas de uma empresa de e-commerce, como realizar o gerenciamento de clientes, estoque e vendas, ser de fácil implementação e fácil integração com aplicações clientes (aplicações web, aplicativos para dispositivos móveis...). Se faz necessário que a aplicação esteja estruturada de modo que seja possível o incremento de novas funcionalidades à medida que se faça necessário.

Os objetivos específicos deste trabalho são:

- Apresentar um sistema que atenda as necessidades de uma empresa de e-commerce de gerir usuários, estoque e vendas online;
- Apresentar a arquitetura desse sistema de forma clara e objetiva, listando os requisitos funcionais e não-funcionais que ela deve suprir;
- Apresentar alguns dos fluxos de operações realizadas pelo sistema, como a operação de venda, mostrando todas as etapas pelas quais ela passa e os componentes envolvidos;
- Listar os pontos de melhoria e quais seriam os próximos passos a serem realizados para a implementação do sistema no ambiente de produção.

2 ESPECIFICAÇÃO ARQUITETURAL DA SOLUÇÃO:

2.1 CONTEXTO DO PROJETO:

A arquitetura de um software representa um conjunto de decisões significativas de projeto sobre como um software é organizado para que, atingindo seu objetivo final, apresente uma gama de atributos de qualidades que foram requeridos pelo cliente ao solicitar o desenvolvimento do sistema (KEELING, 2017).

Uma decisão de projeto pode ser significativa por uma série de fatores, seja pela sua influência no custo do projeto ou tempo a mais de desenvolvimento do projeto que deve ser despendido para atendê-la. Dessa forma, uma decisão crítica tomada de forma errada pode colocar todo o projeto a perder (KEELING, 2017).

Para o desenvolvimento do projeto deste trabalho, levantamos um cenário onde deseja-se desenvolver uma solução para empresas de e-commerce que desejam sair do marketplace e redes sociais, expandindo o seu negócio com a sua própria loja virtual. Focando nesse público-alvo em específico, decidiu-se que o sistema não precisa, e nem deve, ser complexo a ponto de requerer dezenas de funcionários apenas para mantê-lo e deve estar apto a evoluir naturalmente, podendo receber novas funcionalidades para atender com facilidade novos cenários que surgirem no mercado.

Um diferencial para a aplicação em questão consiste no fato de que além de atender as necessidades de uma loja online, ela também deve poder ser utilizada como sistema de gerenciamento para lojas físicas, possibilitando que as empresas atuem em duas frentes utilizando o mesmo sistema.

A aplicação deve possuir um módulo de clientes, onde é possível cadastrá-los e realizar a manutenção dos mesmos; um módulo de produtos, onde deve ser possível cadastrar e gerenciar produtos e o seu estoque, e um módulo de vendas, responsável pelas operações de vendas da empresa.

Um componente importante para uma loja online, o gerenciamento de pagamentos não será realizado pela aplicação devido à natureza sensível que essa operação apresenta. O sistema, no entanto, deve estar apto a ser integrado a outras aplicações que realizarão essa função, notificando quando uma venda for realizada e executando ações específicas quando for notificado da ocorrência de um pagamento.

Esta decisão foi tomada a fim de manter o nosso sistema agnóstico à plataforma de pagamentos que estiver sendo utilizada no momento, de modo que esta poderia ser substituída futuramente sem causar qualquer impacto em nossa aplicação. Vale ressaltar que essa aplicação responsável pelo gerenciamento de pagamentos, no entanto, não faz parte do escopo deste trabalho.

2.2 ESTILO ARQUITETURAL:

Como mencionado anteriormente, o nosso sistema consistirá em três módulos distintos: Um módulo de clientes, um módulo de produtos e um módulo de vendas. Enquanto podemos assumir que produtos e clientes são entidades que podem operar completamente independentes entre si, o mesmo não é o caso para o módulo de vendas, que necessita dos outros dois módulos mencionados anteriormente: uma venda não pode ser realizada para um cliente que não esteja cadastrado, bem como não pode ser processada caso não haja estoque o suficiente do produto selecionado.

Com esse cenário em mente, poderíamos sugerir o desenvolvimento do nosso sistema baseando-o em uma arquitetura monolítica dividida em camadas, onde cada módulo viraria uma entidade em nosso banco de dados e este se integraria à nossa aplicação através da camada de persistência de dados. Acima desta nós teríamos uma camada de serviços, responsável por realizar as operações de negócio a partir de requisições recebidas da camada de apresentação, que em nosso caso seria a API.

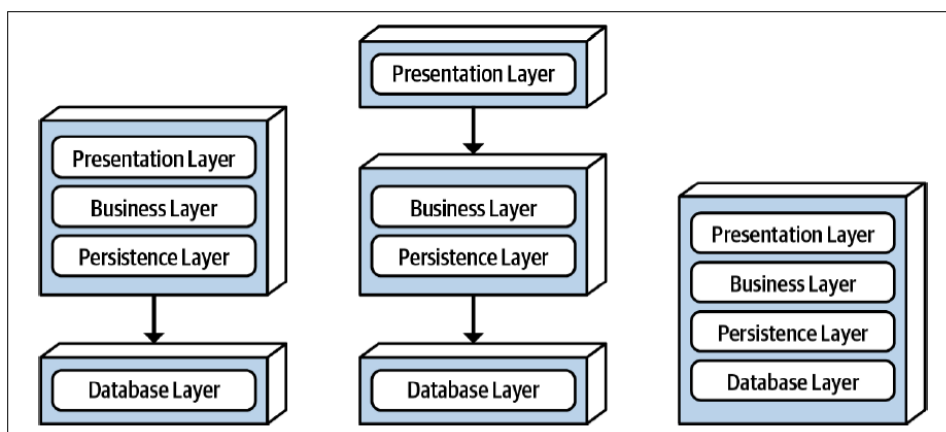


Figura 1: Topologia de uma Arquitetura em Camadas (Sommerville, 2009)

No entanto, apesar de a arquitetura monolítica ser, em comparação aos demais tipos de arquitetura, a mais fácil e menos custosa de se desenvolver, desejamos que o nosso sistema seja flexível o bastante para que novas funcionalidades pudessem ser adicionadas a ele com facilidade, sem a necessidade de realizar a remodelagem de toda a aplicação. Essa evolução pode não ser facilmente alcançada utilizando uma arquitetura monolítica clássica, principalmente em um cenário onde há regras de negócio complexas e que envolvem várias entidades diferentes dentro do sistema.

Por exemplo: a operação de venda em uma arquitetura monolítica requereria um método ou função responsável pela coordenação de todo o processo, que trabalharia com dados das três entidades básicas do sistema, resultando no acoplamento entre as mesmas e trazendo complexidade ao sistema. Nessa situação, qualquer modificação em uma das entidades poderia refletir negativamente no nosso método de venda e este se tornaria um ponto de preocupação constante sempre que houvesse qualquer tipo de alteração no sistema.

Para contornar este problema se decidiu por isolar os módulos internos completamente e orquestrar a comunicação entre eles através de um mediador. Essa comunicação ocorrerá através de eventos e comandos transmitidos por um “Message Broker”, semelhante à forma que uma arquitetura distribuída baseada em eventos trabalha.

Iremos, portanto, combinar as arquiteturas monolítica e a baseada em eventos a fim de aproveitarmos as vantagens que elas apresentam, enquanto compensamos os seus aspectos negativos, de modo que a aplicação atenda aos requisitos solicitados.

A lista detalhada dos requisitos funcionais e não-funcionais que levaram à decisão do estilo arquitetural adotado, seguido das restrições arquiteturais impostas ao desenvolvê-la, estão descritos na próxima seção.

2.3 REQUISITOS FUNCIONAIS:

Os requisitos funcionais descrevem o que um sistema deve ou não deve fazer, de como deve reagir a entradas específicas e de como deve se comportar em determinadas situações. Os requisitos funcionais irão, por vezes, refletir a forma como uma organização opera, detalhando como o sistema deve agir de acordo com a atual estrutura organizacional e procedimentos internos da empresa (SOMMERVILLE; Ian, 2009).

Para o nosso sistema, os requisitos funcionais estão listados abaixo:

ID	Descrição Resumida	Dificuldade	Prioridade
RF01	O sistema deve permitir o autocadastramento de clientes	B	A
RF02	Cada cliente deve ter acesso apenas às suas informações	B	A
RF03	Apenas funcionários com o nível de permissão “administrador do sistema” podem desbloquear (status: ativo) ou bloquear (status: inativo) o cadastro de um cliente cadastrado no sistema.	B	B
RF04	Os clientes cadastrados e com cadastros ativos no sistema devem conseguir realizar compras de produtos cadastrados no sistema, desde que estejam autenticados.	M	A
RF05	Os clientes devem poder editar suas informações cadastrais, exceto o e-mail utilizado para autenticação, desde que seu cadastro não esteja bloqueado.	B	B
RF06	O sistema deve validar o CPF informado por um cliente e não deve permitir que dois cadastros apresentem o	B	B

	mesmo CPF.		
RF07	O sistema deve permitir o cadastro e edição dos dados de produtos por funcionários com o nível de permissão “gerente de produtos”.	B	A
RF08	O sistema deve realizar a baixa/incremento do estoque de produto automaticamente no momento em que uma venda é realizada.	A	A
RF09	O sistema deve reprovar a venda caso ela contenha produtos que não possuem quantidade suficiente em estoque.	B	A
RF10	O sistema deve permitir a manutenção de estoque (baixa/incremento) manual realizado por funcionários com o nível de permissão “gerente de produtos”.	B	A
RF11	Devido à utilização deste sistema também em uma loja física, ele deve permitir que funcionários com o nível de permissão “gerente de vendas” possam realizar o cadastro e editar as informações cadastrais de um cliente, desde que este não esteja com o cadastro bloqueado.	A	A
RF12	Devido à utilização deste sistema também em uma loja física, ele deve permitir que funcionários com o nível de permissão “gerente de vendas” possam registrar vendas para clientes cadastrados no sistema, desde que estes não estejam com os seus cadastros bloqueados.	M	A
RF13	Uma venda só pode ser processada/editada por quem a criou ou por quem tiver a permissão de nível “administrador do sistema”.	M	M
RF14	Uma venda só pode ser cancelada/estornada se ela fora concluída em no máximo 48 horas.	B	M

Tabela 1: Requisitos Funcionais do Sistema (B – baixo, M – médio, A – alto)

2.4 REQUISITOS NÃO-FUNCIONAIS:

Estes requisitos, como o próprio nome sugere, não estão diretamente relacionados com os serviços específicos que são fornecidos pelo sistema ao usuário final. Requisitos não-funcionais dizem respeito a propriedades do sistema como o seu tempo de resposta ou disponibilidade (SOMMERVILLE; Ian, 2009). Esses requisitos irão auxiliar na definição de como um sistema deve ser estruturado.

Os requisitos não-funcionais do nosso sistema podem ser observados abaixo:

ID	Descrição	Prioridade
RNF01	Segurança – O sistema não deve permitir que usuários tenham acesso a dados de outros usuários.	A
RNF02	Manutenibilidade – O sistema deve ser decomposto em módulos independentes que não se comunicam diretamente entre si, de modo a evitar o acoplamento entre diferentes partes do sistema.	M
RNF03	Persistência – O sistema deve utilizar o SQLite como mecanismo de implementação de banco de dados relacional.	B
RNF04	Resiliência – Em caso de erro, as mensagens assíncronas devem ser reprocessadas exponencialmente pelo menos 3 vezes. Caso o erro persista, a mensagem a ser reprocessada irá ser persistida em um arquivo de log com a causa do erro.	M
RNF05	Monitoramento – O sistema deve manter logs das requisições e execuções de serviços, informando o autor da requisição. O log deve ser gerado em um arquivo JSON, gerando 1 novo arquivo por dia.	A

Tabela 2: Requisitos Não-funcionais do Sistema (B – baixa, M – média, A – alta)

Os requisitos não-funcionais guiaram a forma com que a arquitetura do sistema foi estabelecida, em especial os requisitos RNF02 e RNF03: A adoção de filas evitará o acoplamento dos diferentes módulos de sistema, tornando-o mais robusto e apto a receber manutenção com mais facilidade, apesar de aumentar o nível de complexidade do desenvolvimento inicial da aplicação.

A escolha do SQLite como solução de implementação do banco de dados relacional do sistema decorre do fato de que o seu banco de dados fica salvo em um arquivo de extensão “.db” que pode ser facilmente extraído e armazenado para backup em um disco local. Por estarmos tratando de uma solução voltada para pequenas e médias empresas de e-commerce, a utilização do SQLite como solução de persistência remove da equação a complexidade que a manutenção e backup de ferramentas de banco de dados mais complexas, como o MS SQL Server ou Postgres, adicionariam. O desenvolvimento de novas versões dessa aplicação utilizando diferentes SGBDs deve ocorrer futuramente, dando mais opções a quem desejar utilizar o sistema em sua empresa.

2.5 RESTRIÇÕES ARQUITETURAIS:

Com os requisitos definidos nós podemos listar as restrições arquiteturais do sistema, as quais limitarão as possíveis soluções a serem implementadas para atingir os requisitos especificados previamente.

ID	Descrição
R01	Este sistema deve ser uma aplicação WEB
R02	A aplicação deve ser desenvolvida em C#, utilizando o framework .NET 6
R03	A aplicação deve oferecer uma API que adere aos princípios e padrões arquiteturais REST.
R04	A aplicação deve ser containerizada utilizando Docker como plataforma de virtualização.

Tabela 3: Restrições Arquiteturais do Sistema

2.6 MECANISMOS ARQUITETURAIS:

Mecanismos arquiteturais são conceitos técnicos utilizados para atingir soluções para problemas que são comumente encontrados no desenvolvimento de um software, realizando a ligação entre requisitos arquiteturais e tecnologias concretas.

Abaixo podem ser observado os mecanismos arquiteturais utilizados em nossa aplicação:

Análise	Desenho	Implementação
Persistência	Banco de Dados Relacional	SQLite
Versionamento	Aplicação de Versionamento de Código	Git
Comunicação entre Componentes	“Message Broker”	Redis
Interface de Comunicação	Web API	.NET Web API
Autenticação	OAuth2	Keycloak
Auditoria	Log	Serilog
Deploy	Containerização	Docker

Tabela 4: Mecanismos Arquiteturais do Sistema

3 MODELAGEM ARQUITETURAL:

A modelagem arquitetural de um software visa apresentar uma representação gráfica de como o sistema está estruturado, como seus componentes interagem entre si e como o sistema como um todo interage com o ambiente externo.

Para essa modelagem se optou por utilizar o Modelo C4, criado pelo Simon Brown, que consistem em um conjunto de diagramas hierárquicos em camadas utilizados para descrever a arquitetura de uma aplicação em diferentes níveis de abstração. Os 4 níveis do Modelo C4 são:

- Diagrama de Contexto: apresenta a macroestrutura do sistema e como ele interage com o meio externo.
- Diagrama de Contêiner: dá ênfase aos contêineres internos do sistema e como eles interagem entre si.
- Diagrama de Componentes: adentra nos contêineres listados no diagrama anterior, apresentando os seus componentes internos, os seus relacionamentos e detalhando as tecnologias utilizadas.
- Diagrama de Classe: detalha a estrutura interna de cada um dos componentes, apresentando como as classes que os compõem estão estruturadas.

Para esse projeto trabalharemos apenas com os diagramas de contexto, contêiner e componentes.

3.1 DIAGRAMA DE CONTEXTO:

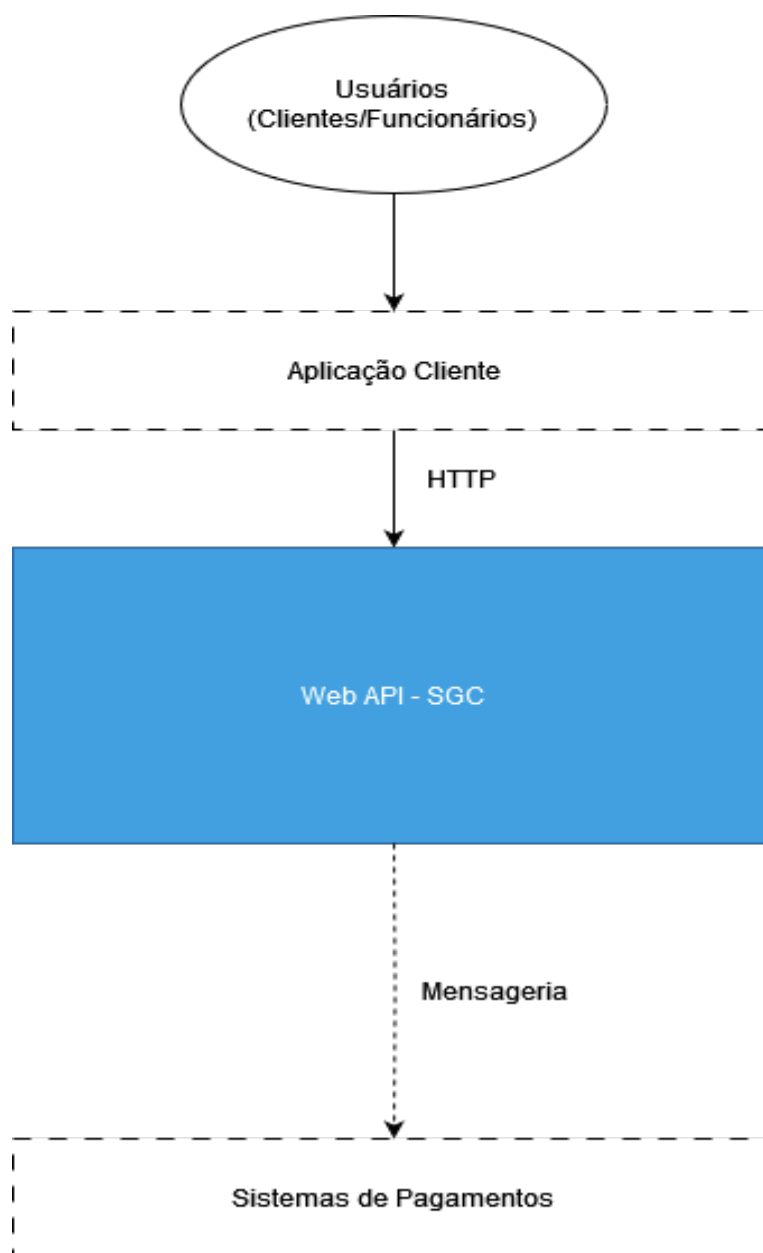


Figura 2: Diagrama de Contexto da Aplicação SGC

O diagrama de contexto apresentado mostra a relação entre a nossa aplicação e o meio externo. O nosso sistema encapsulará todas as operações de negócio da aplicação e disponibilizará uma API para comunicação com aplicações clientes (representado no diagrama por uma aplicação front-end). A nossa aplicação também poderá se comunicar com outras aplicações, como por exemplo o sistema responsável por processar o pagamento das compras realizadas pelos clientes, através de mensagens gerenciadas pelo “Message Broker”.

3.2 DIAGRAMA DE CONTÊINER:

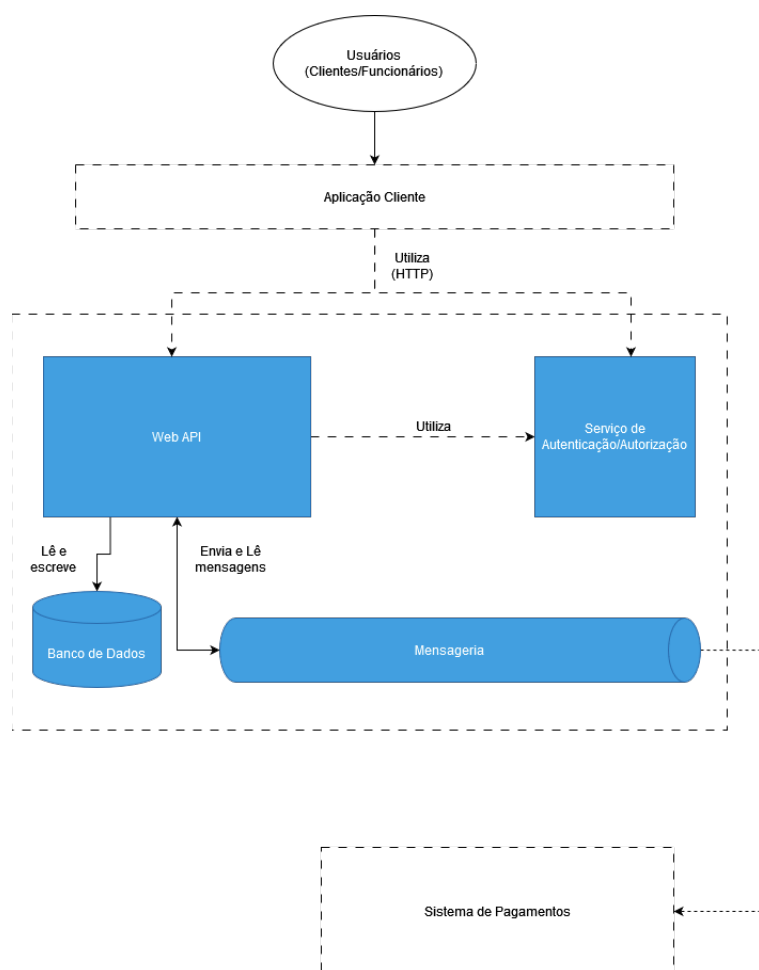


Figura 3: Diagrama de Contêiner – Aplicação SGC

O diagrama de contêiner enfatiza os contêineres internos da aplicação desenvolvida neste trabalho. O coração do sistema é a Web API desenvolvida utilizando o framework .NET que, além de disponibilizar uma interface para requisições externas de aplicações clientes, concentra toda a lógica de negócios da aplicação. Os dados gerados a partir das operações realizadas na aplicação API serão persistidas em um banco de dados relacional (no nosso caso o SGBD escolhido foi o SQLite).

A aplicação API também está integrada a um “Message Broker”, de onde ele irá ler e para onde publicará mensagens que serão processadas assincronamente. A utilização do “Message Broker” visa manter os componentes internos da Aplicação API desacoplados e também serve como uma porta de comunicação do nosso sistema a serviços externos que vierem a ser desenvolvidos posteriormente, como o sistema de gerenciamento de pagamentos. Optou-se por utilizar o Redis como serviço de gerenciamento de mensagens pela sua facilidade de implantação, que já atende de forma eficiente as necessidades do sistema.

Por fim, a aplicação API está configurada para utilizar um serviço de gerenciamento e autenticação de usuários, sejam eles clientes ou funcionários. Optou-se por utilizar o Keycloak para realizar essa função por ser um software livre, já bastante conhecido no mercado e que possui suporte ao protocolo OAuth2.

3.3 DIAGRAMA DE COMPONENTES:

O diagrama de componentes representa os componentes internos de um contêiner e a relação entre eles. Como pôde ser observado no diagrama de contêiner apresentado anteriormente, nós temos a aplicação web, desenvolvida utilizando o framework .NET, e alguns outros componentes, soluções já disponíveis no mercado, que foram integrados ao nosso sistema e também listados na seção anterior: O Redis como um “Message Broker”, o Keycloak como serviço de gerenciamento de acessos e o SQLite como sistema de gerenciamento de banco de dados.

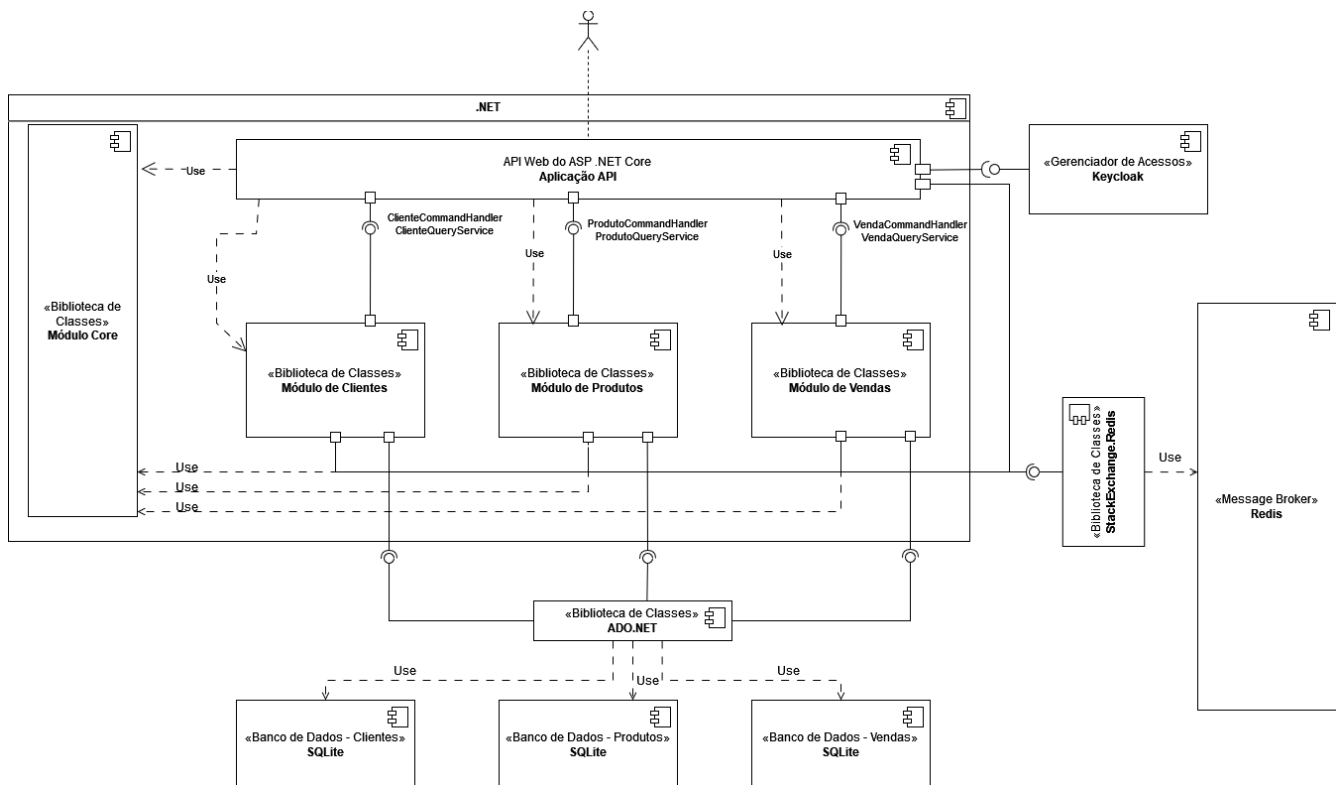


Figura 4: Diagrama de Componentes – Arquitetura Geral do Sistema – Aplicação SGC

A aplicação web pode ser dividida em 4 componentes principais: A API, desenvolvida utilizando o projeto API Web do ASP.NET Core e 3 bibliotecas de classes, representando cada um dos 3 módulos na nossa aplicação: O módulo de Clientes, o módulo de Produtos e o módulo de Vendas.

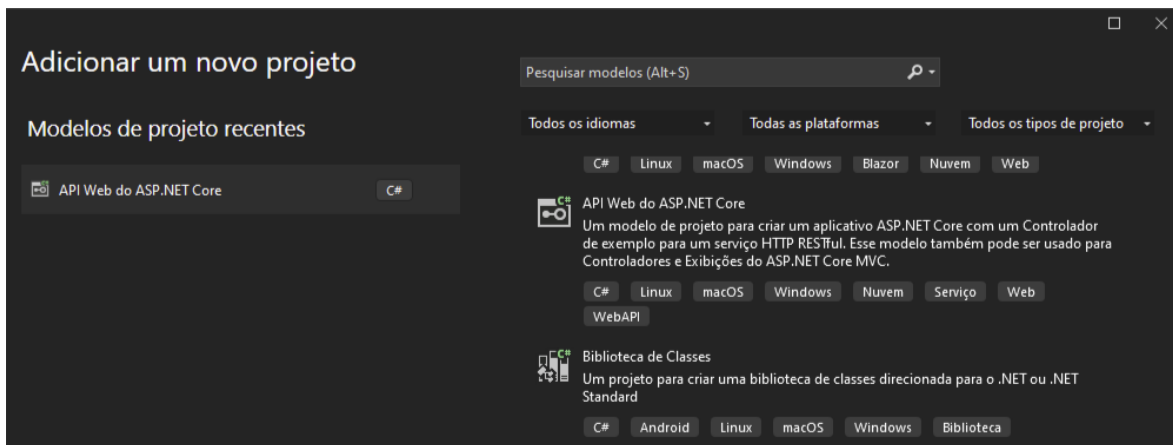


Figura 5: Tela Inicial para criação de um novo projeto com o Framework .NET no Visual Studio

O projeto API Web está configurado para utilizar o Keycloak a partir do protocolo OAuth2, validando o Token nesse componente e a partir dele verificando quais as permissões, ou “roles”, que o usuário que está usando o sistema possui.

Podemos ver que tanto a API quanto os módulos do sistema fazem uso do "Message Broker", enquanto os módulos internos também utilizarão o SQLite a fim de realizar a leitura e persistência de dados.

Para entender melhor a relação dos componentes individuais da aplicação Web além da relação entre eles e os serviços externos mencionados anteriormente, apresentaremos o diagrama de componentes destes componentes internos.

3.3.1 DIAGRAMA DE COMPONENTES – API Web do ASP.NET Core

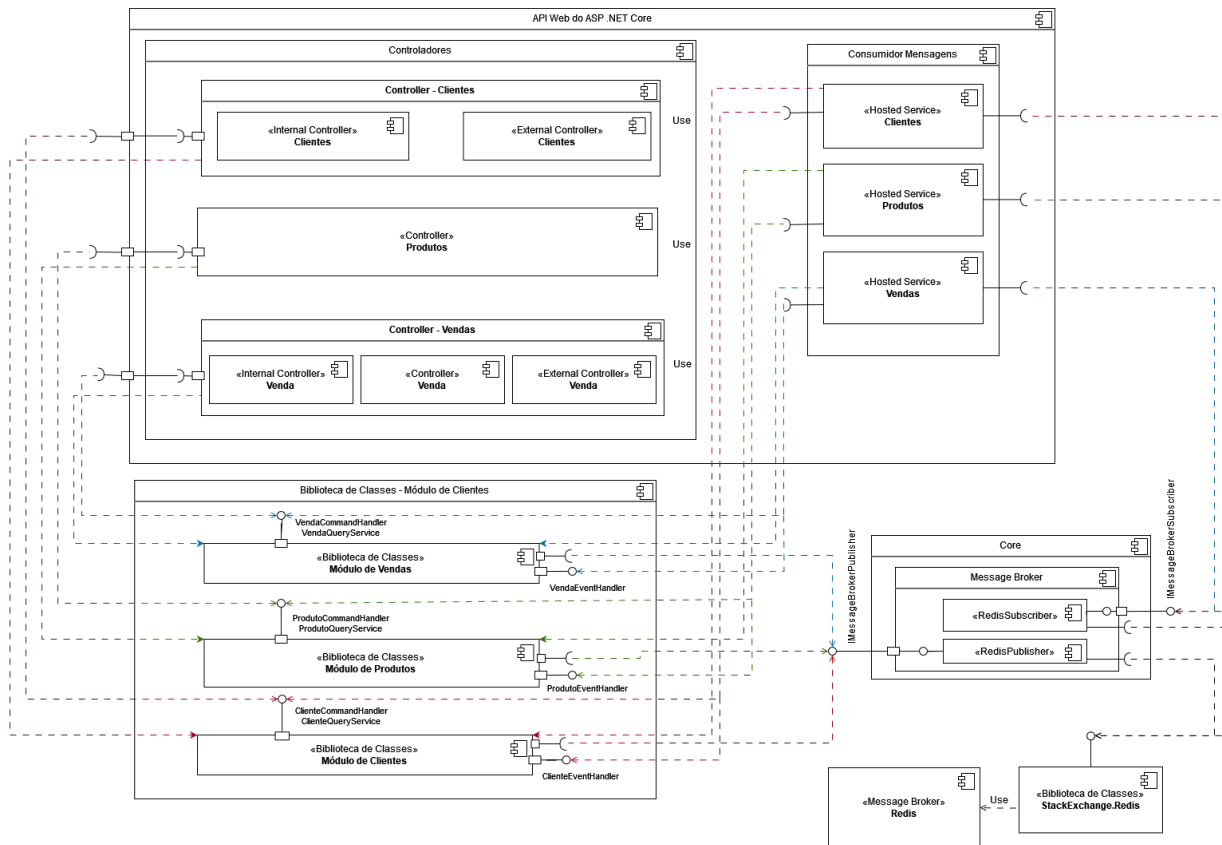


Figura 6: Diagrama de Componentes – API Web do ASP.NET Core – Aplicação SGC

O projeto API Web do .NET será o responsável por receber as requisições externas de aplicações clientes através de chamadas HTTP realizadas para os seus “controllers” (controladores). Há três grupos de “controllers” – um por módulo de sistema – para fins de organização. Esses grupos irão conter 1 ou mais “controllers” e estes irão validar se o usuário que está realizando a chamada possui permissão para leitura/modificação do recurso solicitado e, em caso afirmativo, repassarão a requisição para o módulo responsável.

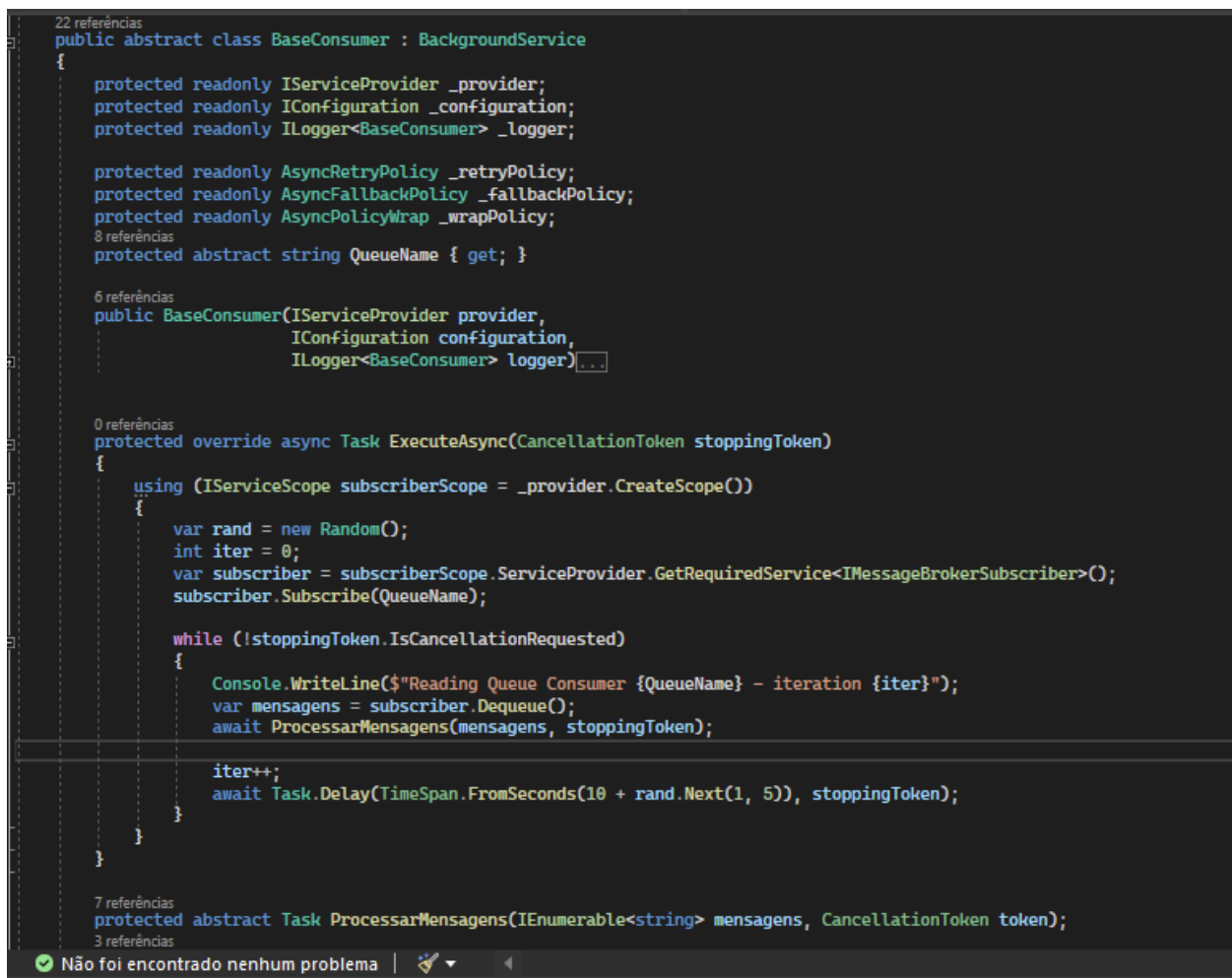
A organização mencionada no parágrafo anterior tem por objetivo separar diferentes funções de um mesmo módulo por grupos de usuários: Como a aplicação é voltada para a implementação de uma loja online, mas também atendendo o cenário de implementação em uma loja física, ela atenderá basicamente dois grupos: Os usuários externos (clientes) e os usuários internos (funcionários).

Quanto às funcionalidades que cada um desses grupos deve ter acesso, funcionários deverão poder cadastrar vendas, produtos e usuários, enquanto que clientes deverão ter acesso apenas às suas informações e também a realizar compras online. Assim, nem sempre o formato das requisições será o mesmo para estes dois grupos e por isso diferentes “controllers” foram implementados para cada um deles quando necessário. O módulo de produtos possui apenas 1 controller porque sua manutenção é realizada apenas internamente, ou seja, por funcionários. O módulo de venda apresenta 3 porque um de seus “controllers” contém funções comuns aos dois grupos mencionados anteriormente, como a inclusão, alteração e remoção de itens de uma venda.

Além dos “controllers”, uma outra funcionalidade que o projeto API Web do ASP.NET Core disponibiliza, e que foi utilizada em nossa aplicação, é a da implementação de “Hosted Services”. Os “Hosted Services” implementam uma classe abstrata própria, a “BackgroundService”, que permite que um serviço permaneça em funcionamento ininterruptamente durante toda a vida útil da aplicação, ou até atingir uma condição preestabelecida.

Por esse motivo os “Hosted Services” são os componentes ideais para operarem como consumidores de mensagens publicadas ao “Message Broker”, sendo responsáveis por receber e processá-las enquanto a aplicação estiver em funcionamento. Essa solução foi adotada, como já mencionado anteriormente, para restringir o acoplamento dos módulos da aplicação.

Abaixo podemos ver a implementação da classe abstrata “BaseConsumer”, desenvolvida com os métodos e propriedades que devem ser implementadas pelas classes concretas dos consumidores de mensagens do sistema e um dos exemplos de consumidor: O “ClienteAtualizadoConsumer”, que lê mensagens publicadas no canal de clientes do “Message Broker” e deve notificar o módulo de venda quando um dos clientes teve seus dados atualizados.



```

22 referências
public abstract class BaseConsumer : BackgroundService
{
    protected readonly IServiceProvider _provider;
    protected readonly IConfiguration _configuration;
    protected readonly ILogger<BaseConsumer> _logger;

    protected readonly AsyncRetryPolicy _retryPolicy;
    protected readonly AsyncFallbackPolicy _fallbackPolicy;
    protected readonly AsyncPolicyWrap _wrapPolicy;
    8 referências
    protected abstract string QueueName { get; }

    6 referências
    public BaseConsumer(IServiceProvider provider,
                        IConfiguration configuration,
                        ILogger<BaseConsumer> logger)...

    0 referências
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        using (IServiceScope subscriberScope = _provider.CreateScope())
        {
            var rand = new Random();
            int iter = 0;
            var subscriber = subscriberScope.ServiceProvider.GetRequiredService<IMessageBrokerSubscriber>();
            subscriber.Subscribe(QueueName);

            while (!stoppingToken.IsCancellationRequested)
            {
                Console.WriteLine($"Reading Queue Consumer {QueueName} - iteration {iter}");
                var mensagens = subscriber.Dequeue();
                await ProcessarMensagens(mensagens, stoppingToken);

                iter++;
                await Task.Delay(TimeSpan.FromSeconds(10 + rand.Next(1, 5)), stoppingToken);
            }
        }
    }

    7 referências
    protected abstract Task ProcessarMensagens(IEnumerable<string> mensagens, CancellationToken token);
    3 referências

```

Não foi encontrado nenhum problema

Figura 7: Trecho do código de implementação do Hosted Service para a classe abstrata BaseConsumer


```

2 referências
public class ClienteAtualizadoConsumer : BaseConsumer
{
    0 referências
    public ClienteAtualizadoConsumer(IServiceProvider provider,
                                     IConfiguration configuration, ILogger<BaseConsumer> logger) ...
    3 referências
    protected override string QueueName => "cliente-*";
    2 referências
    protected async override Task ProcessarMensagens(IEnumerable<string> mensagens, CancellationToken token)
    {
        using (IServiceScope scope = _provider.CreateScope())
        {
            VendaEventHandler handler = scope.ServiceProvider.GetRequiredService<VendaEventHandler>();
            foreach (var mensagem in mensagens)
            {
                await _wrapPolicy.ExecuteAsync(async (context) =>
                {
                    var deserialized = JsonConvert.DeserializeObject<ClienteVendaAtualizadoEvent>(mensagem);
                    if (deserialized != null)
                    {
                        _logger.LogInformation("Dequeue: {mensagem}", deserialized.Serialize());
                        await handler.Handle(deserialized, token);
                    }
                },
                new Context()
                {
                    ["mensagem"] = mensagem
                });
            }
        }
    }
}

```

Figura 8: Trecho de Código com a Implementação do Consumidor de Mensagens “ClienteAtualizadoConsumer”

Um exemplo mais completo de como essa solução funciona pode ser observada a partir do cenário de confirmação de uma venda: Quando uma venda é processada, em vez de uma chamada direta ao módulo de produtos, uma mensagem é publicada ao "Message Broker" informando que uma venda foi realizada e quais os seus produtos e quantidades. O “Hosted Service” consumidor que está configurado para receber as mensagens do canal de vendas irá acionar o módulo de produtos para que haja a atualização das quantidades em estoque dos produtos em questão.

Esse processo segue com a confirmação de baixa no estoque e a notificação de que os produtos da venda estão reservados e aguardando a confirmação de pagamento. Quando este é confirmado, também via mensagem publicada em um canal específico no "Message Broker", o status da venda é alterado para “venda confirmada”.

O processo leva ainda em conta situações em que não há quantidade suficiente de produto em estoque ou quando uma notificação de “pagamento rejeitado” é recebida pelo consumidor de mensagens responsável por repassar essa informação ao módulo de vendas.

O fluxo dessa operação pode ser observado no diagrama abaixo.

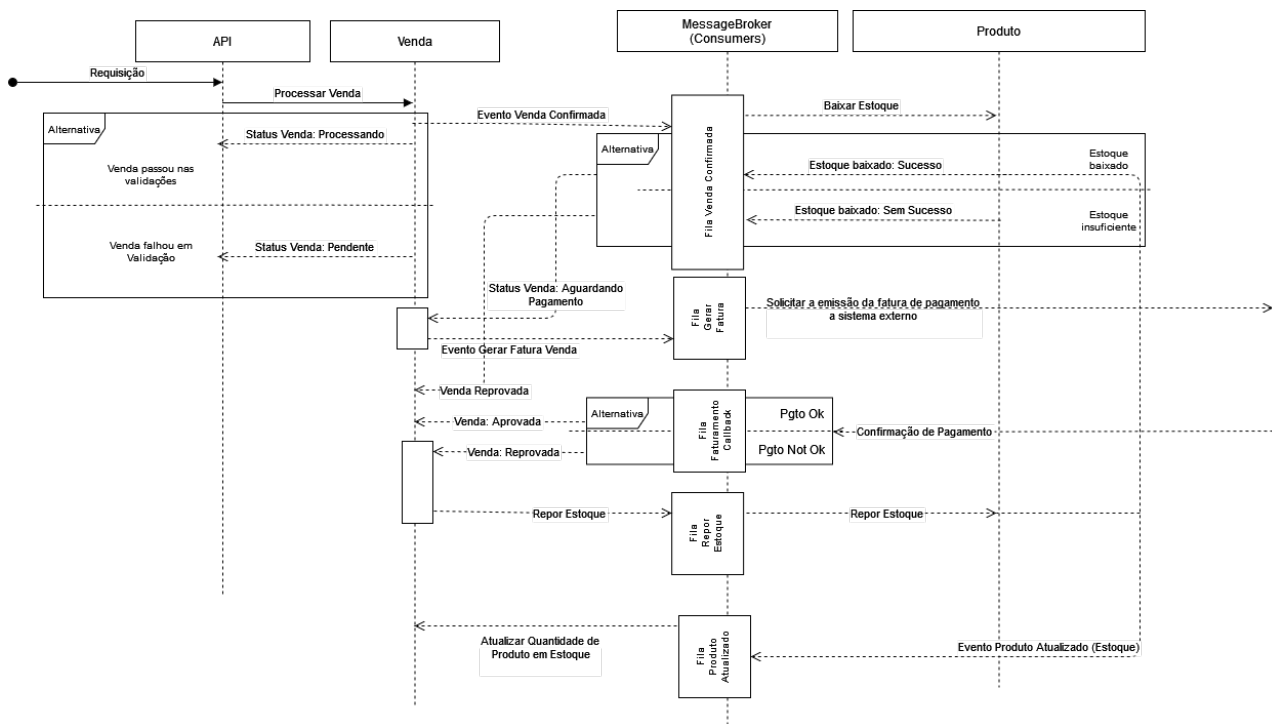


Figura 9: Diagrama do fluxo da operação de venda na aplicação SGC

A solução adotada não somente elimina o acoplamento entre os módulos como também dá a possibilidade de que, no futuro, outros serviços possam tirar proveito das mensagens publicadas: Em um eventual desenvolvimento de um serviço de e-mails que notifica o usuário quando a venda foi realizada, basta que o “Hosted Service” em questão dispare o comando ao serviço utilizando as informações obtidas via “Message Broker”.

3.3.2 DIAGRAMA DE COMPONENTES – Módulos

Os módulos internos do sistema seguem a mesma estrutura e por esse motivo realizaremos a análise de apenas um deles: o módulo de clientes. O que será apresentado nessa seção se aplica aos demais módulos.

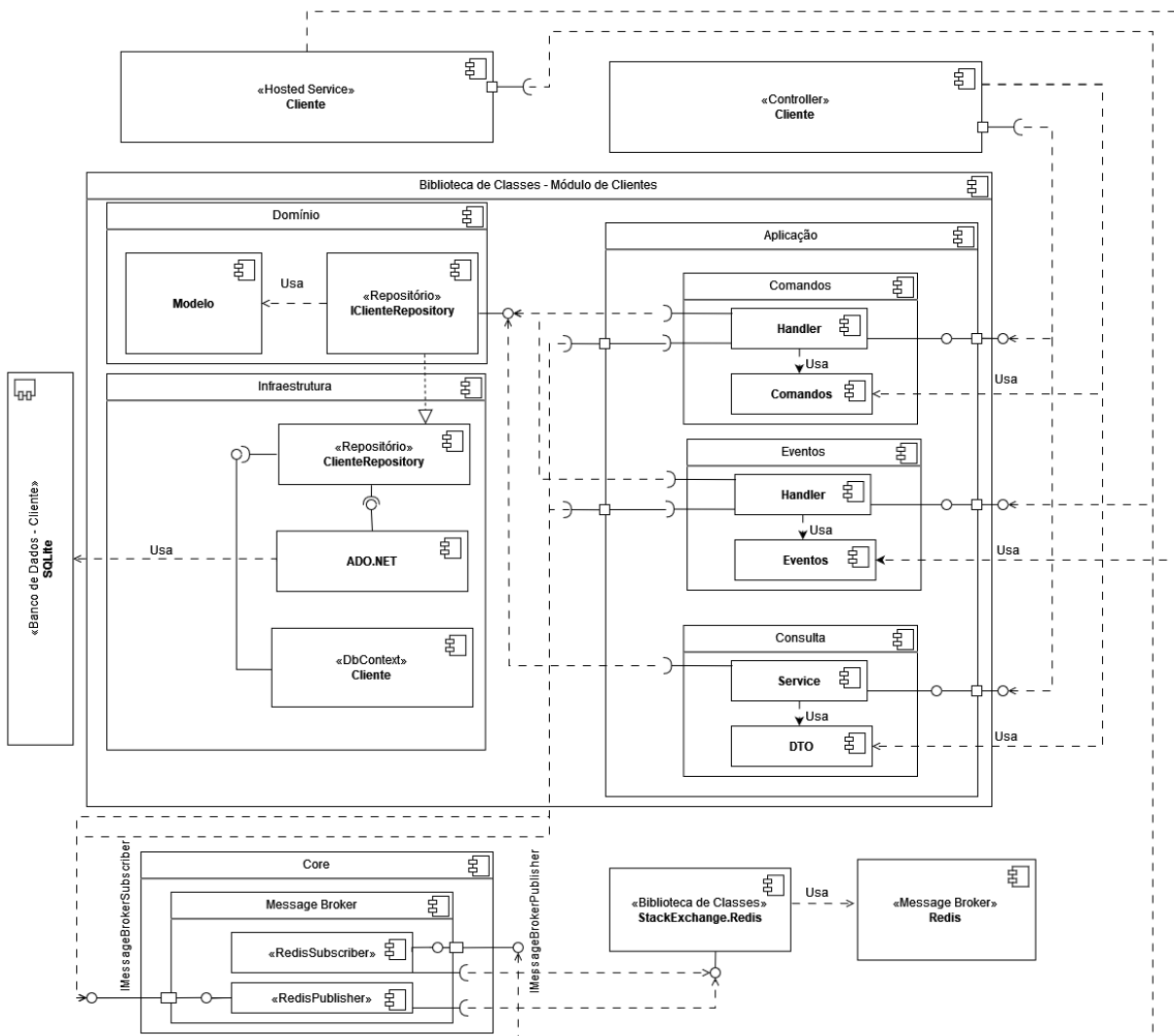


Figura 10: Diagrama de Componentes – Módulo de Clientes da Aplicação SGC

Toda a lógica de negócios da aplicação está contida em seus módulos internos e estes poderiam ser estruturados de diversas formas, como, por exemplo, em camadas – modelo, serviço e persistência – que é um padrão bastante comum de se encontrar no mercado.

Optou-se, no entanto, em desenvolver estes componentes utilizando os princípios de projeto dirigido por domínios, ou “domain driven design”, uma abordagem de desenvolvimento de software focada no domínio e na lógica de negócios do sistema. O DDD consiste em um número de princípios definidos e publicados em 2003 por Eric Evans no livro “Domain-Driven Design: Atacando as complexidades no coração do software” que, quando implementados, tornam o sistema mais robusto e de fácil manutenção.

Temos um sistema composto de entidades bem definidas (clientes, venda e produtos), cada um destes com suas próprias regras de negócio e que são o coração da aplicação e a razão pela qual ele foi desenvolvido em primeiro lugar. Esse conjunto de características torna esse cenário de desenvolvimento ideal para a implementação do DDD.

Adentrando na estrutura do módulo nós temos a camada de domínio, que conterá os modelos e a interface de implementação do repositório para persistência. Os modelos também conterão todas as regras de validação e operações pertinentes a eles, como por exemplo a validação do CPF informado para a criação de um novo registro de cliente no sistema. A camada de domínios é utilizada pela camada de infraestrutura, que realizará a implementação de fato do repositório, e é o responsável por buscar e persistir os dados diretamente do SGBD através da interface fornecida pelo Dapper, o micro ORM adotado para a atividade em questão.

Por fim temos a camada de aplicação, composta pelos comandos, eventos e serviço de consulta. O serviço de consulta, como o próprio nome diz, será utilizado para atender todas as requisições de leitura solicitadas pelos usuários.

Já comandos e eventos representam as instruções de ações que o sistema deve realizar. Eles encapsulam as informações necessárias para essa execução em um objeto que é repassado para o “Handler”, o componente (em nosso caso, uma classe) responsável pelo processamento dos comandos/eventos, coordenando sua execução e interação com outros componentes do sistema, como o repositório e o “Message Broker”.

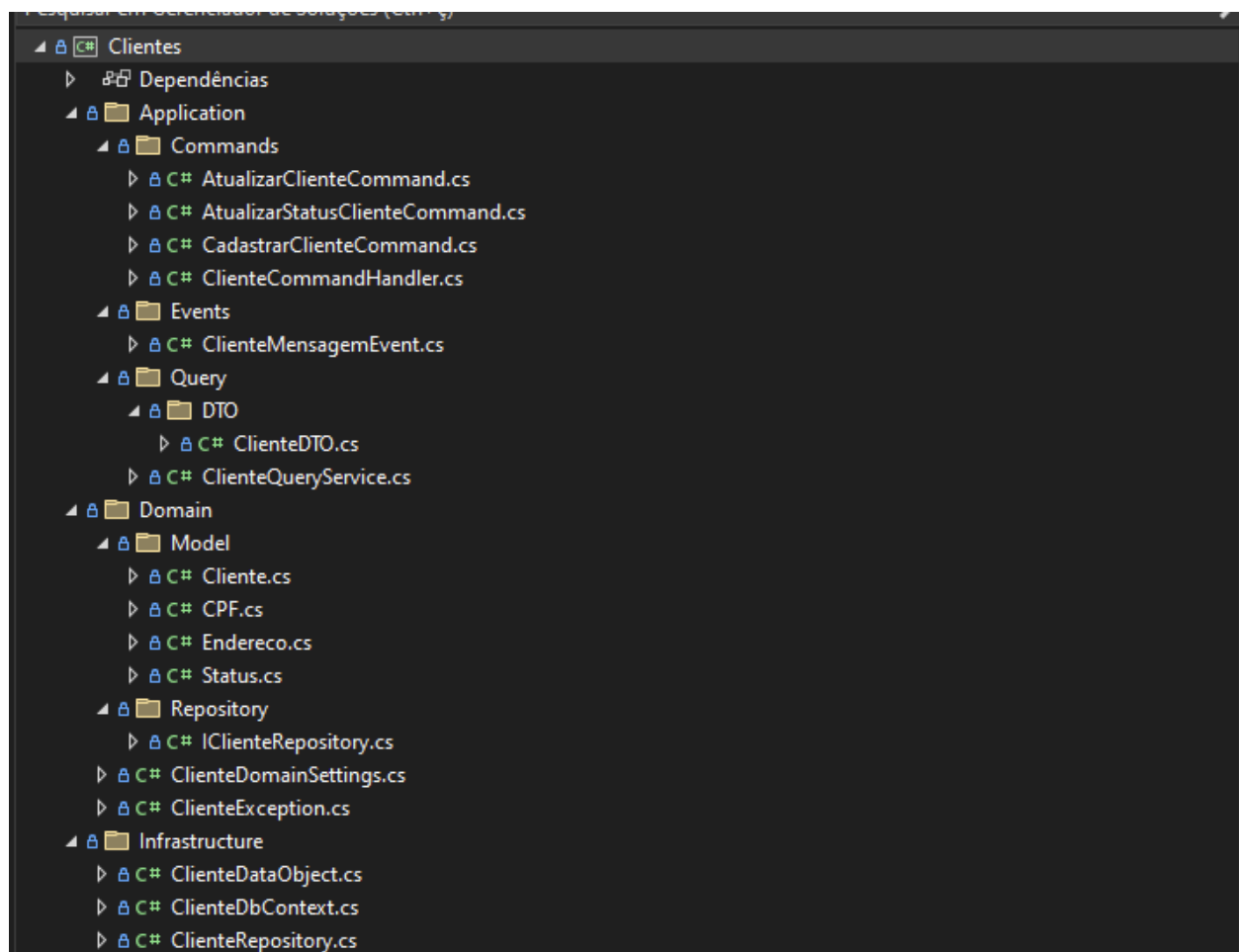


Figura 11: Organização interna dos componentes do Módulo de Clientes

A adoção dessa abordagem apresenta diversas vantagens para o nosso sistema: novas funções podem ser facilmente implementadas, bastando criar novos comandos e o seu respectivo “Handler” para executá-lo; cada comando representa apenas uma instrução no sistema, o que assegura a segregação de responsabilidades e promove o princípio da responsabilidade única, um dos 5 postulados do SOLID (acrônimo que se refere aos 5 princípios de padrão de projeto que tem por objetivo tornar o software mais flexível, manutenível e compreensível); a implementação de comandos remove a necessidade de que outros módulos do sistema tenham conhecimento sobre a assinatura do método a ser executado na camada de aplicação, sendo necessário apenas criar um objeto referente ao comando e passá-lo ao “Handler”.

3.3.3 DIAGRAMA DE COMPONENTES – CORE

Quanto ao módulo Core, ele não participa ativamente de nenhuma operação de negócios do sistema e consiste basicamente em classes, interfaces e implementações que são utilizadas nos demais componentes do sistema, os quais foram agrupados nesse módulo para evitar a repetição desnecessária de código no sistema.

Como destaque podemos mencionar as interfaces de comandos, entidades e raízes agregadas, utilizadas pela camada de aplicação e domínio dos módulos internos, a interface IDbContext e classe abstrata UnitOfWork<T>, fundamentais na implementação do repositório, e a implementação do “Message Broker”. Os detalhes da sua estrutura podem ser observados na figura 12.

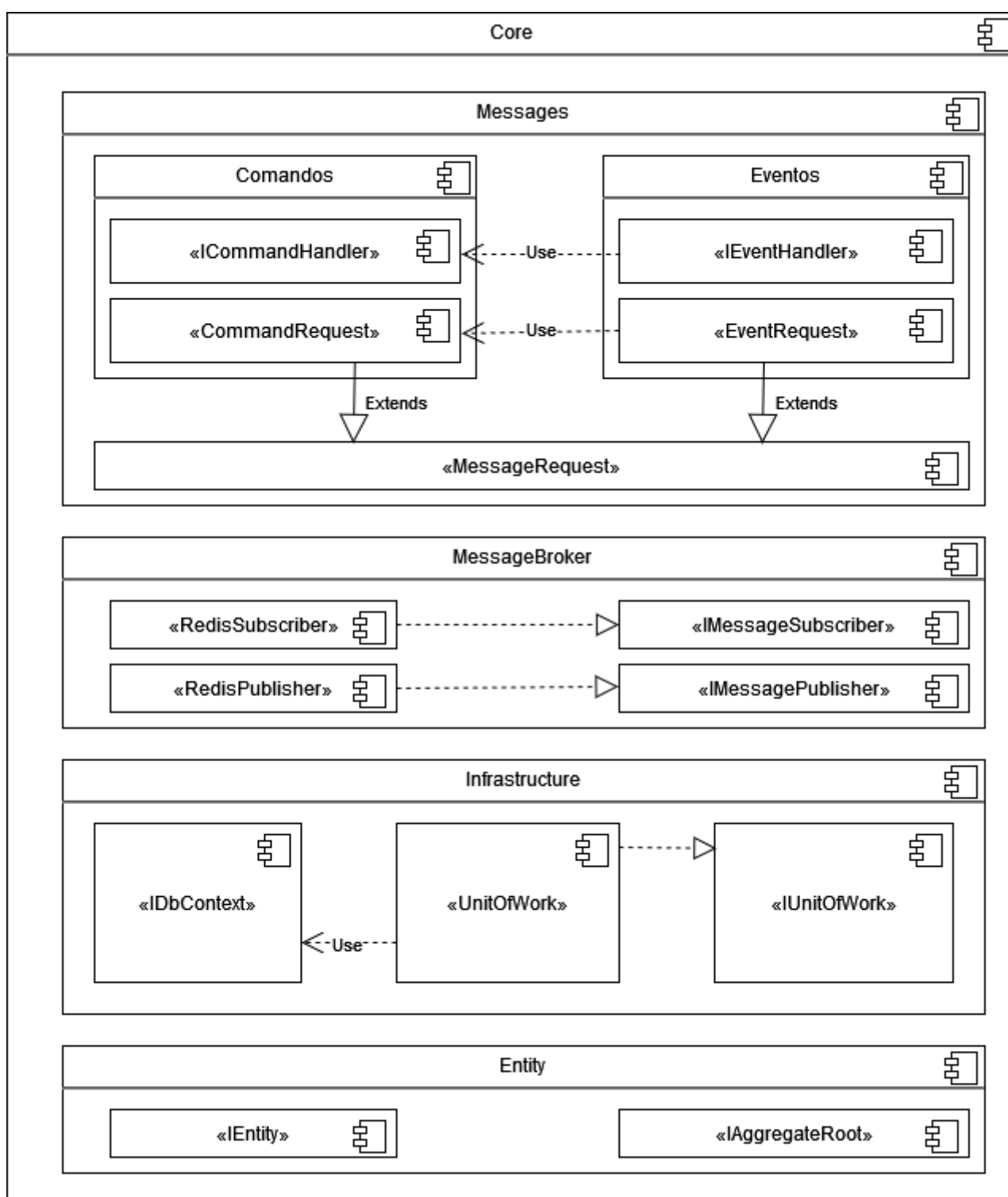


Figura 12: Diagrama de Componentes - Módulo Core

O código da aplicação, onde é possível conferir com mais detalhes todos os pontos mencionados na seção anterior se encontra disponível no repositório <https://github.com/lucasdaniellf/PUC-Minas-SGC>.

4 AVALIAÇÕES DA ARQUITETURA (ATAM)

Os atributos de qualidade de um software são determinados principalmente pela arquitetura escolhida para o desenvolvimento da aplicação. Em outras palavras, performance, manutenibilidade e disponibilidade são mais impactadas pela forma como o sistema está estruturado do que pelo código, linguagem de programação escolhida ou estrutura de dados utilizadas no desenvolvimento da aplicação (KAZMAN et al., 1998).

Isso não significa que estas últimas características mencionadas não são importantes e não têm impacto na performance ou manutenibilidade de um sistema de software, mas sim que a arquitetura tem um peso maior na qualidade do sistema (KAZMAN et al., 1998).

Uma forma de avaliar a arquitetura de uma aplicação é utilizando a metodologia ATAM (*Architecture Tradeoff Analysis Method*), que consiste na avaliação do quanto um projeto de arquitetura de software atende os requisitos de qualidade que foram definidos para o sistema, o quanto bem atende a estes requisitos e os possíveis pontos negativos (KAZMAN et al., 1998).

A análise a seguir irá se utilizar dessa metodologia na avaliação da arquitetura adotada para o sistema apresentado nesse trabalho.

4.1 ANÁLISE DAS ABORDAGENS ARQUITETURAIS

A tabela 5 apresenta os atributos de qualidade a serem atendidos e os cenários utilizados na realização da análise.

Atributos de Qualidade	Cenários	Importância	Complexidade
Segurança	Clientes só devem ter acesso aos seus próprios dados.	A	M
Manutenibilidade	Os módulos do sistema devem ser completamente independentes e desacoplados.	A	A
Resiliência	O sistema deve tentar enviar a mensagem de um módulo para outro até 3 vezes antes de disparar uma mensagem de erro.	M	B
Monitoramento	Todas as ações de usuários e serviços dentro do sistema devem ser registradas em log.	A	B

Tabela 5: Atributos de Qualidade e Cenários Utilizados na Avaliação do SGC

4.2 CENÁRIOS

Os cenários escolhidos para análise estão listados abaixo:

- 1) Segurança: Quando um cliente tentar acessar um endpoint que liste dados de outros clientes ou informações de vendas que não estejam relacionadas a ele, deverá receber um retorno 403 (Forbidden) da aplicação.
- 2) Manutenibilidade: Ao adicionar novas propriedades às entidades do sistema, **como a inclusão do endereço na entidade Cliente**, ou realizar alterações em regras de negócio de uma entidade, não será necessário realizar alteração em nenhum outro módulo da aplicação para que ela continue funcionando.
- 3) Resiliência: Ao enviar uma mensagem através do message broker para realizar a baixa de estoque de um produto que fora vendido a um cliente, em caso de erro (timeout do banco de dados, por exemplo), a aplicação deve tentar processar a mensagem novamente mais duas vezes antes de realizar o log do erro.
- 4) Monitoramento: Ao realizar uma venda, o sistema deve logar o usuário que solicitou a venda, bem como cada um dos passos da operação (baixa de estoque, retorno do pagamento) a medida em que elas vão acontecendo, mantendo “correlation id’s” que podem ser utilizados para rastrear uma requisição do início ao fim.

4.3 EVIDÊNCIAS DA AVALIAÇÃO

4.3.1 CENÁRIO 1 – SEGURANÇA

Atributo de Qualidade:	Segurança
Requisito de Qualidade:	Um cliente não deve conseguir acessar dados relacionados a outros clientes.
Preocupação:	
Os usuários (clientes) devem se sentir seguros o bastante para utilizarem o sistema. Os seus dados são privados e não podem ser acessados por outros clientes ou usuários não-autorizados.	
Cenário(s):	
Cenário 1 – Segurança: Bloquear acesso a dado de um cliente por usuário não autorizado.	
Ambiente:	
Sistema em operação normal	
Estímulo:	
O cliente tentará enviar uma requisição através de um endpoint ao qual não possui permissão,	

tentando acessar dados pessoais e informações de vendas de um outro cliente.	
Mecanismo:	
Inclusão de um sistema (Keycloak) responsável pelo gerenciamento de usuários e das suas permissões dentro do sistema, onde os usuários deverão autenticar-se para conseguir enviar requisições à aplicação Web.	
Adição de uma política de autorização nos endpoints da API da aplicação, de modo que clientes só conseguirão acessar uma quantidade limitada de endpoints e nunca obter acesso a dados de outros clientes/usuários.	
Medida de resposta:	
A requisição enviada pelo cliente não autenticado deverá ter um retorno do tipo 401 – Unauthorized. A requisição enviada por um usuário/cliente autenticado para um endpoint ao qual não tem acesso deverá ter um retorno do tipo 403 – Forbidden (Proibido).	
Considerações sobre a arquitetura:	
Riscos:	Descontinuação ou uma falha crítica que seja descoberta no sistema de gerenciamento de acessos adotado.
Pontos de Sensibilidade:	Adoção de um sistema de autenticação que deve ser configurado com as Roles e Policies definidas na aplicação, a qual não precisará ter conhecimento de informações como login e senha de usuário.
Tradeoff:	Necessidade de configuração do sistema externo de gerenciamento com as Roles e Policies, de modo que ocorra a comunicação entre ele e a aplicação Web.

Tabela 6: Cenário 1 de avaliação da arquitetura – Segurança

O cenário 1 considerará o requisito de segurança em relação a privacidade dos dados dos clientes que se cadastrarão e utilizarão o sistema em questão. O requisito estabelece que clientes devem ter acesso apenas a seus dados dentro do sistema, não conseguindo acessar dados pessoais ou informações de compras de outros clientes. A solução adotada para suprir esse cenário foi composta dos itens abaixo:

1. Autenticação de usuários: Os clientes (bem como demais usuários, como funcionários e gerentes) deverão estar autenticados para utilizar o sistema. O processo de autenticação de um usuário será realizado pelo “Keycloak”, um software de gerenciamento de acessos que pode ser facilmente configurado para cuidar do processo de autenticação de usuários em aplicações Web.
2. Autorização de usuários: Adotou-se uma política de “Roles” e “Claims” para acessar diferentes endpoints da API da aplicação web. Qualquer usuário que não esteja autenticado receberá um código de resposta do tipo 401 (Unauthorized) e receberá um código 403 (Forbidden) como retorno caso esteja autenticado mas não possua a autorização necessária para acessar um destes endpoints.
3. Inclusão das Roles e Claims no sistema de gerenciamento de usuários: Para manter as configurações de autenticação e autorização em um só lugar, após a definição das Roles que a aplicação Web deve possuir, estas foram configuradas no Keycloak para que sejam automaticamente (ou manualmente, pelo responsável pela manutenção da aplicação) vinculadas aos novos usuários.

Para utilizarmos o Keycloak em conjunto com a aplicação será utilizado uma imagem Docker que operará dentro da rede interna criada pelo arquivo do “Docker Compose” responsável por disponibilizar todos os serviços necessários para que o SGC opere conforme o esperado.

Dentro dessa instância do Keycloak será configurado um “Realm”, que nada mais é que o espaço responsável pelo gerenciamento de usuários, aplicações, roles e grupos. Um usuário pertence e realiza o login dentro de um Realm (https://www.keycloak.org/docs/latest/server_admin/).

A aplicação Web do SGC é então configurada como um cliente do keycloak, de modo que validará contra o Realm criado as informações de usuários que estão tentando acessar os seus endpoints, validando o Token de acesso que é repassado no Header da requisição.

```
0 referências
public static class AuthenticationAuthorizationExtensionServices
{
    1 referência
    public static IServiceCollection AddAuthenticationAuthorization(this IServiceCollection services, IConfiguration configuration)
    {
        services.AddAuthentication(opt =>
        {
            opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
            opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        }).AddJwtBearer(opt =>
        {
            opt.Authority = configuration.GetSection("Authentication")["Authority"];
            opt.Audience = configuration.GetSection("Authentication")["Audience"];
            opt.RequireHttpsMetadata = false;
        });
    }
}
```

Figura 13: Configuração de Esquema de autenticação a ser utilizado pela aplicação Web

```

services.AddSingleton<IAuthorizationRequirement, AtualizarVendaAuthorizationRequirement>();
services.AddSingleton<IAuthorizationHandler, AtualizarVendaAuthorizationRequirementHandler>();

services.AddSingleton<IAuthorizationRequirement, LerVendaAuthorizationRequirement>();
services.AddSingleton<IAuthorizationHandler, LerVendaAuthorizationRequirementHandler>();

services.AddAuthorization(opt =>
{
    opt.AddPolicy(Policies.PoliticaAtualizarVenda, policy => policy.AddAtualizarVendaAuthorizationRequirement());
    opt.AddPolicy(Policies.PoliticaLerVenda, policy => policy.AddLerVendaAuthorizationRequirement());

    opt.AddPolicy(Policies.PoliticaGerenciamentoProduto, policy => policy.RequireRole(Roles.GerenteProdutos));

    opt.AddPolicy(Policies.PoliticaAcessoInterno, policy => policy.RequireAssertion(context => context.User.Claims.Where(c => c.Type == ClaimTypes.Role).Any(r => r.Value != Roles.Cliente)));
    opt.AddPolicy(Policies.PoliticaAcessoExterno, policy => policy.RequireAssertion(context =>
        !context.User.Claims.Where(c => c.Type == ClaimTypes.Role).Any(r => r.Value != Roles.Cliente) && !string.IsNullOrEmpty(context.User.FindFirstValue(ClaimTypes.Email))));
});

return services;

```

Figura 14: Configuração das Roles e Policies para Validação da Autorização do Usuário na Aplicação Web

Todo novo cadastro de usuário incluirá automaticamente nele a role de *Cliente*. Caso o usuário seja um funcionário da empresa, a role específica deverá ser fornecida a ele pelo administrador do sistema diretamente no Keycloak.

cliente-1@sgc.com

Details	Attributes	Credentials	Role mapping	Groups	Consents	Identity provider links	Sessions
<input type="text" value="Search by name"/> → <input type="checkbox"/> Hide inherited roles <input type="button" value="Assign role"/> <input type="button" value="Unassign"/>							
<input type="checkbox"/> Name	Inherited	Description					
<input type="checkbox"/> account manage-account	True	`\${role_manage-account}`					
<input type="checkbox"/> account view-profile	True	`\${role_view-profile}`					
<input type="checkbox"/> account manage-account-links	True	`\${role_manage-account-links}`					
<input type="checkbox"/> my-sgc-app Cliente	True	–					

Figura 15: Configuração de Roles para o usuário Cliente-1

funcionario@sgc.com

Details	Attributes	Credentials	Role mapping	Groups	Consents	Identity provider links	Sessions
<input type="text" value="Search by name"/> → <input type="checkbox"/> Hide inherited roles <input type="button" value="Assign role"/> <input type="button" value="Unassign"/>							
<input type="checkbox"/> Name	Inherited	Description					
<input type="checkbox"/> account manage-account	True	`\${role_manage-account}`					
<input type="checkbox"/> account view-profile	True	`\${role_view-profile}`					
<input type="checkbox"/> account manage-account-links	True	`\${role_manage-account-links}`					
<input type="checkbox"/> my-sgc-app Cliente	True	–					
<input type="checkbox"/> my-sgc-app GerenteProdutos	False	–					
<input type="checkbox"/> my-sgc-app GerenteVendas	False	–					

Figura 16: Exemplo de configuração de Roles para um usuário do tipo Funcionário

Os endpoints da aplicação Web são configurados para serem acessados apenas por usuários que possuem as Roles ou estão dentro da política necessária. Abaixo o processo para um cliente conseguir acessar os seus dados dentro da aplicação, bem como o que acontece ao tentar acessar dados de outros clientes:

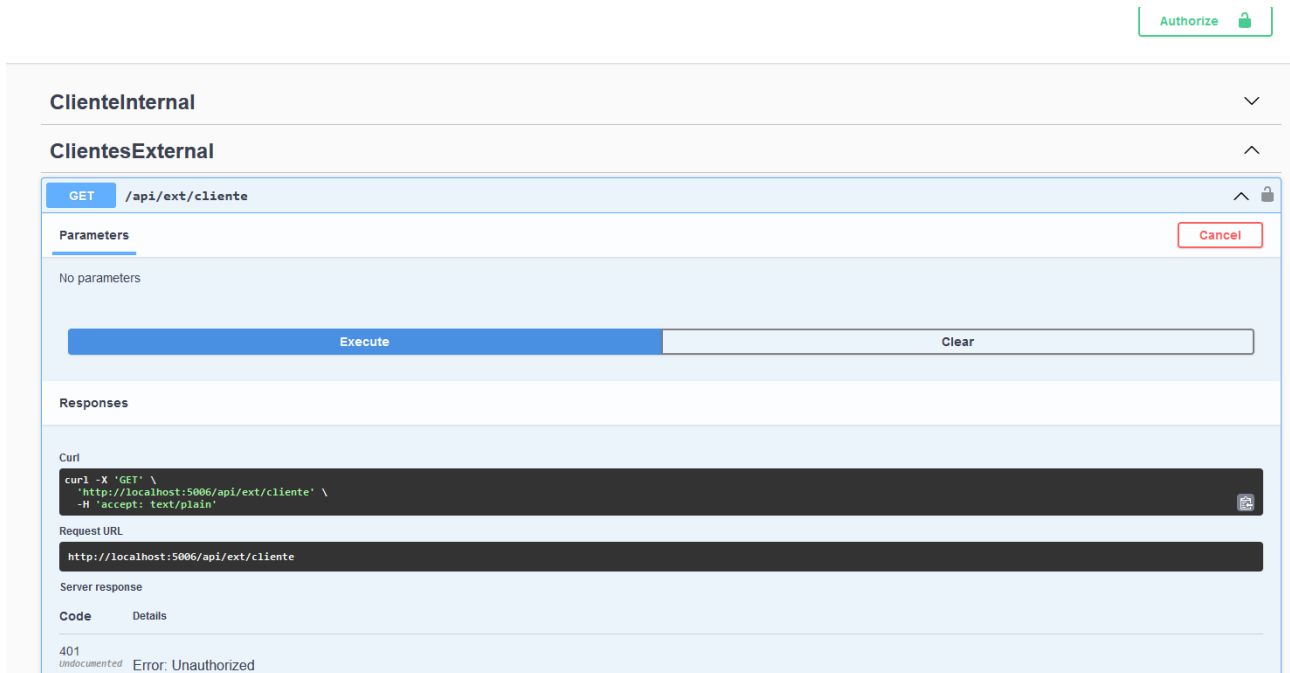


Figura 17: Tentativa de requisição ao endpoint ext/cliente por um usuário não autenticado

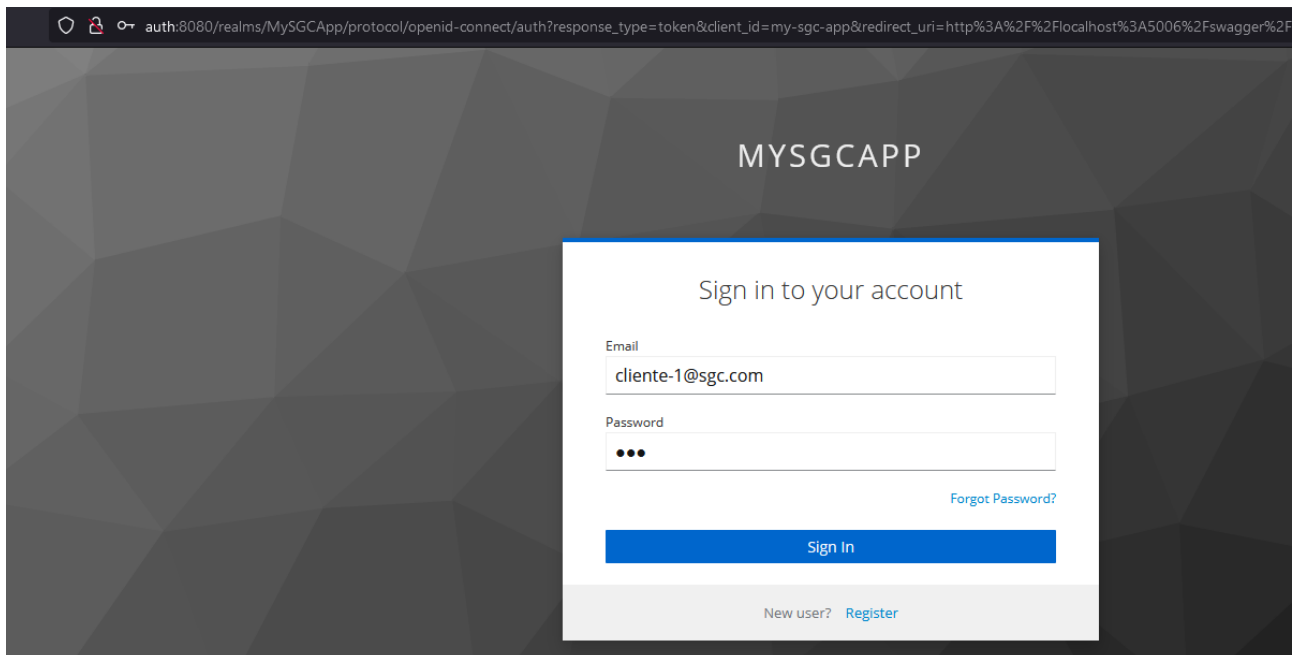


Figura 18: Janela para autenticação do usuário no Keycloak

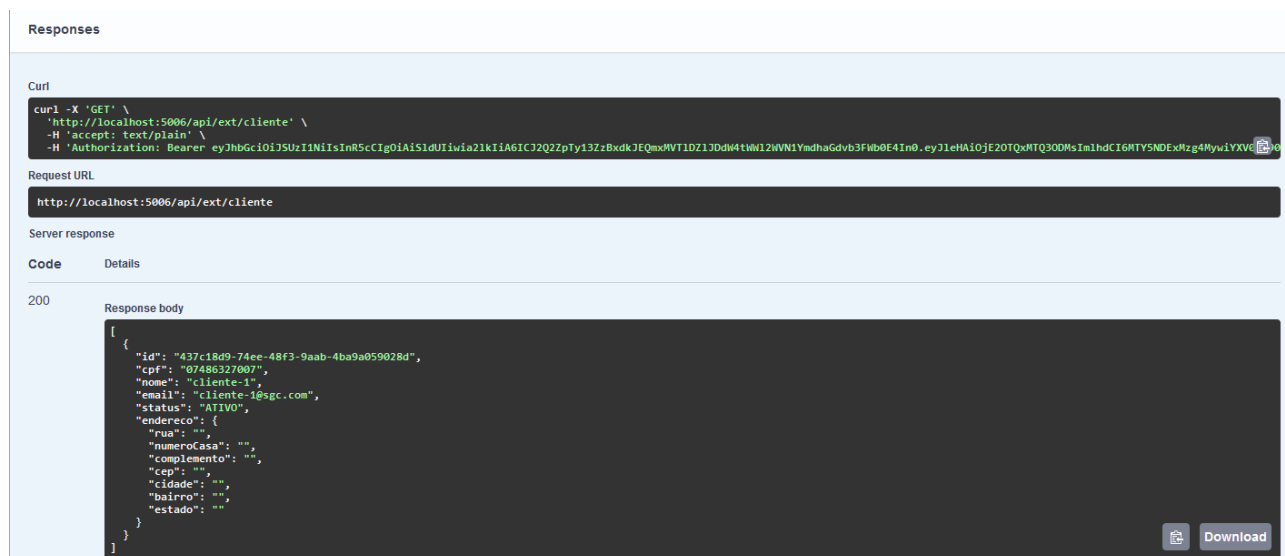


Figura 19: Requisição de cliente para acessar seus próprios dados na aplicação Web



Figura 20: Requisição de cliente para acessar dados de todos os clientes (apenas funcionários devem ter acesso a esse endpoint)

4.3.2 CENÁRIO 2 – MANUTENIBILIDADE

Atributo de Qualidade:	Manutenibilidade
Requisito de Qualidade:	Alteração de um dos módulos do sistema não deve impactar os demais.
Preocupação:	
A fim de termos um sistema que pode ser expansível e de manutenção facilitada, qualquer alteração de regra de negócio ou inclusão de novas propriedades nas entidades do sistema devem impactar apenas o módulo em questão, não impactando negativamente os demais.	
Cenário(s):	
Cenário 2 – Manutenibilidade: Inclusão da propriedade Endereço no módulo de Clientes não	

requer ajuste de nenhum outro módulo.	
Ambiente:	
Sistema em operação normal	
Estímulo:	
Manutenção por parte do time de TI no código da aplicação para atender uma solicitação de negócios de incluir o endereço físico como propriedade da entidade cliente (visando atender o cenário de entregas em domicílio).	
Mecanismo:	
Separação dos módulos da aplicação de modo que eles operam completamente independentes entre si e se comunicam de forma assíncrona através de um message broker.	
Medida de resposta:	
Os demais módulos do sistema devem apresentar o mesmo comportamento/resposta antes e depois da alteração realizada no módulo de clientes.	
Considerações sobre a arquitetura:	
Riscos:	Caso a nova propriedade de uma entidade seja necessária em mais de um módulo, será preciso replicar a propriedade em mais de 1 local no sistema e atualizar os campos de mais de 1 banco de dados.
Pontos de Sensibilidade:	Separação do sistema em diferentes módulos que não se comunicam diretamente entre si e que não compartilham nem o mesmo banco de dados.
Tradeoff:	Apesar de novas funcionalidades, propriedades ou ainda alterações em regras de negócio puderem ser realizadas em um módulo com mínimo ou nenhum impacto nos demais, essa abordagem adiciona uma camada de complexidade à forma como o sistema está estruturado, requerendo estudo por parte de novos desenvolvedores que trabalharão com o código.

Tabela 7: Cenário 2 de avaliação de arquitetura – Manutenibilidade

O cenário 2 considerará o requisito de manutenibilidade em relação a como um módulo do sistema pode ser alterado causando nenhum impacto nos demais. A solução adotada para suprir esse cenário foi descrita na seção 3.3, onde a forma como os componentes da aplicação web estão estruturados é detalhada.

Podemos comprovar essa característica verificando os arquivos que sofreram alteração no código da aplicação quando a propriedade de endereço foi incluída à entidade cliente (figuras 21 e 22): não houve nenhum tipo de alteração nos módulos de produtos ou vendas, mesmo que este possua sua própria representação da entidade Cliente (figuras 23 e 24), que é alimentada através de requisições que são recebidas pelo message broker.

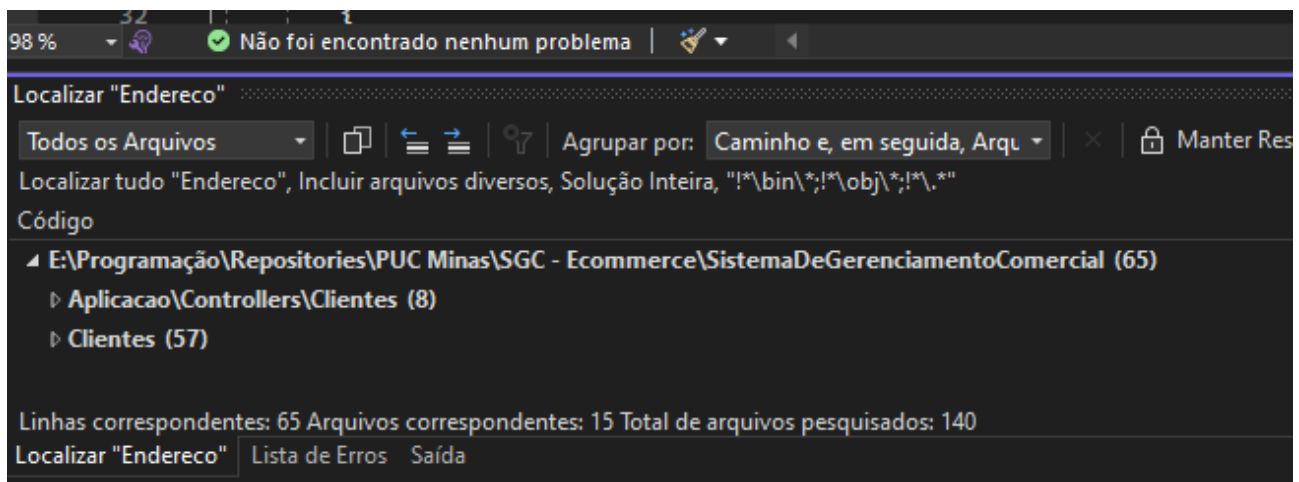


Figura 21: Lista de Módulos que possuem referência à propriedade Endereço da entidade Cliente

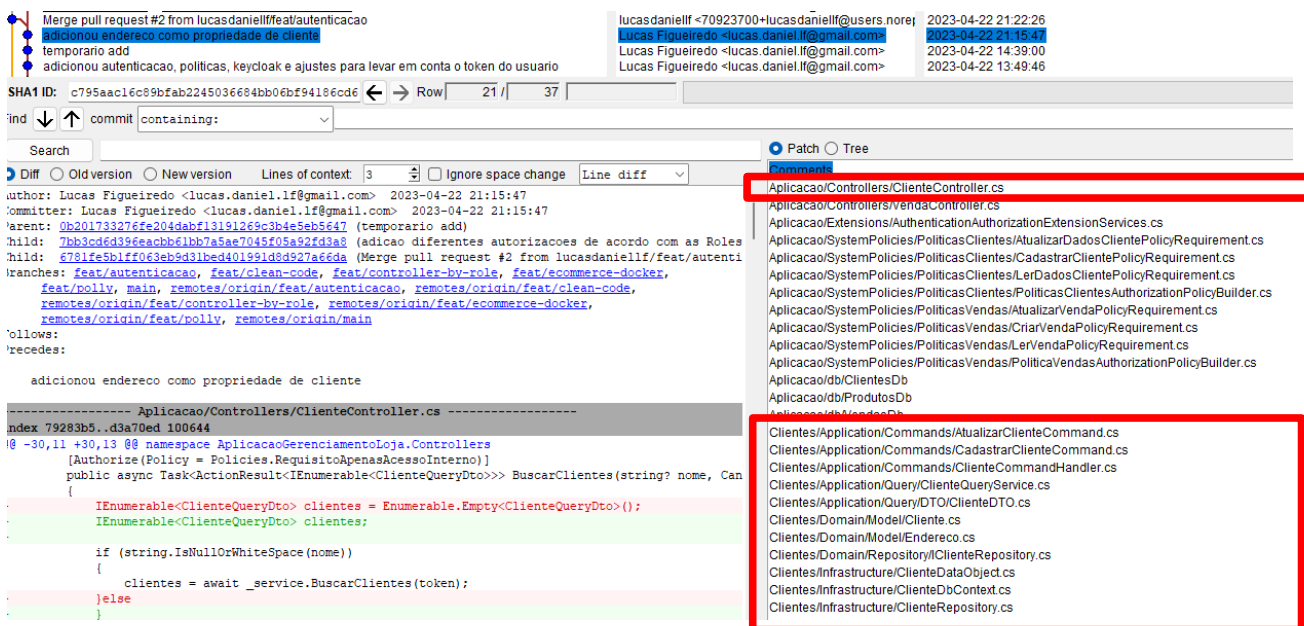


Figura 22: Lista de arquivos que sofreram alteração na inclusão da propriedade Endereço na entidade Cliente através do comando Gitk do Git

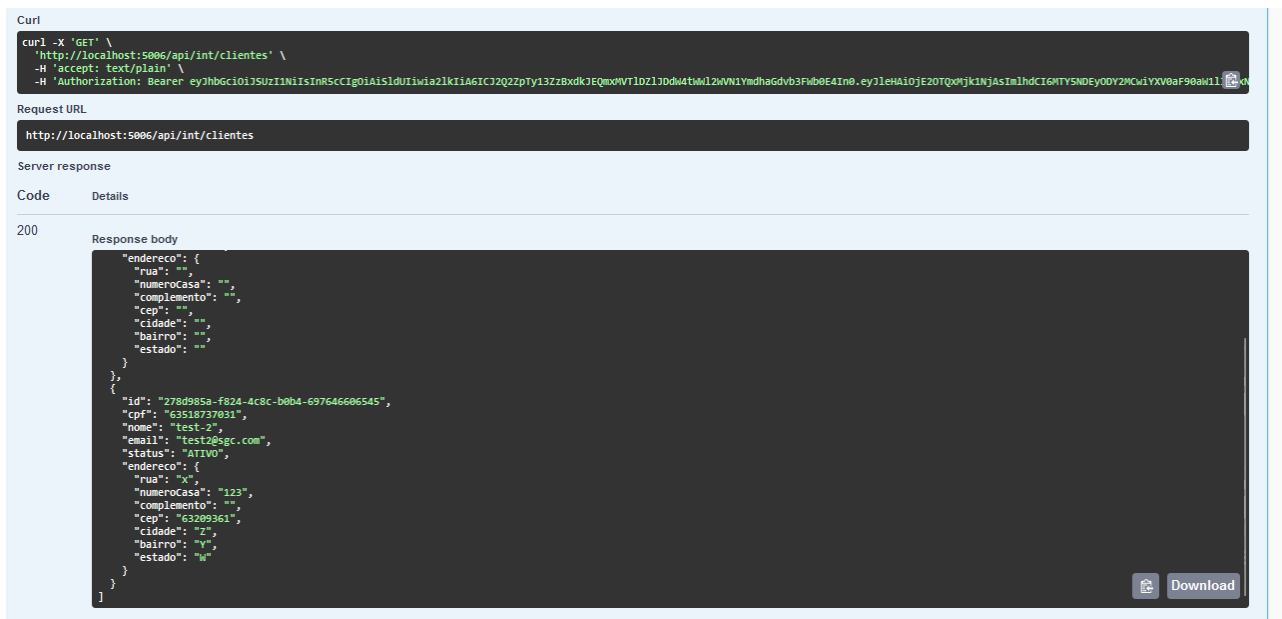


Figura 23: Representação de Cliente para o módulo de Clientes – Com propriedade Endereço

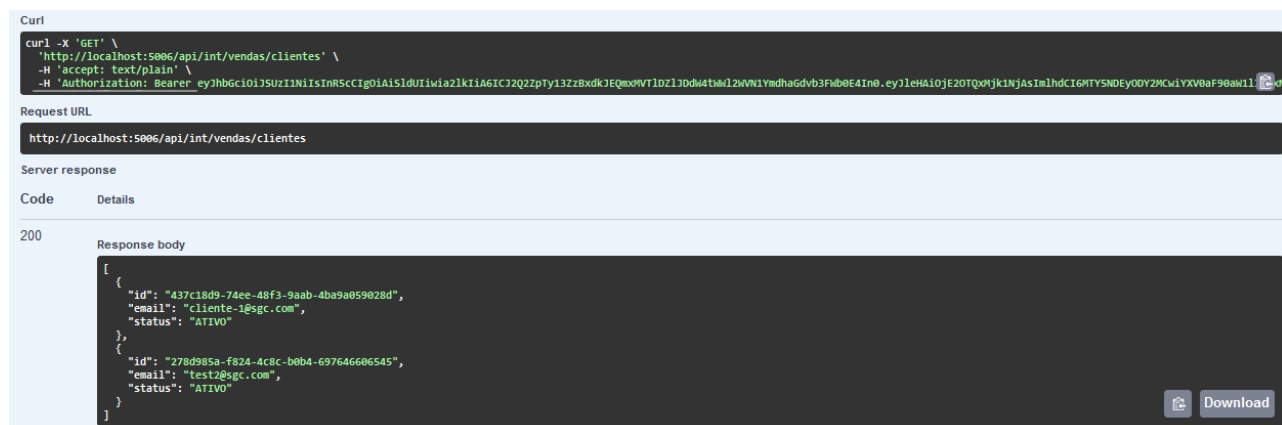


Figura 24: Representação de Cliente no módulo de Vendas – Sem a propriedade de Endereço, já que esta não é relevante para as operações realizadas neste módulo

4.3.3 CENÁRIO 3 – RESILIÊNCIA

Atributo de Qualidade:	Resiliência
Requisito de Qualidade:	O sistema deve tentar processar as mensagens do message broker pelo menos 3 vezes antes de disparar uma mensagem notificando sobre o erro de processamento.
Preocupação:	
A fim de termos um sistema com um baixo número de solicitações que resultem em erro, deseja-	

se tentar o processamento de mensagens enviadas através do message broker no mínimo 3 vezes. Dessa forma, em caso de indisponibilidade do banco de dados, a aplicação não reportaria um erro imediatamente, tentando processar a mensagem novamente dentro de alguns segundos.	
Cenário(s):	
Cenário 3 – Resiliência: Aplicação deve tentar processar a mensagem de baixa de estoque de um produto que é enviada pelo message broker após a confirmação de uma venda pelo menos 3 vezes antes de reportar um erro.	
Ambiente:	
Sistema com alta demanda sobre o banco de dados de produtos.	
Estímulo:	
Confirmação de compra por parte de um usuário/cliente.	
Mecanismo:	
Utilização da biblioteca Polly do .NET 6 para reproprocessamento automático de mensagens enviadas através do message broker em intervalos de tempo pré-definidos.	
Medida de resposta:	
Após erro, o sistema tentará executar o processamento da mensagem mais duas vezes antes de reportar o erro. Todas essas tentativas estarão registradas em log, onde podemos ver quantas vezes a tentativa de processamento ocorreu e a mensagem de erro original.	
Considerações sobre a arquitetura:	
Riscos:	Ao reproprocessar a mensagem de uma fila, as demais mensagens demorarão mais tempo até começarem a ser processadas. Em uma fila com 10 mensagens, caso haja o número máximo de reproprocessamento para todas elas, o tempo de processamento da última mensagem será acrescido de 140 segundos, o que pode dar a impressão de lentidão para o usuário.
Pontos de Sensibilidade:	Utilização da biblioteca Polly para o .NET 6, para o reproprocessamento das mensagens que apresentarem erro. Esse reproprocessamento ocorre em intervalos de tempo que crescem exponencialmente com o número de tentativas:

	primeira tentativa após aguardar 1 segundo, a segunda após 4 segundos e a terceira após 9 segundos.
Tradeoff:	Impressão de lentidão para o usuário quando muitas mensagens que se encontrem na fila precisam ser processadas mais de 1 vez até executarem com sucesso.

Tabela 8: Cenário 3 da avaliação da arquitetura – Resiliência

O cenário 3 considerará o requisito de resiliência em relação a como uma mensagem/comando que é enviada de um módulo para outro irá ser reprocessada automaticamente em caso de erro. Essa solução busca atender principalmente os cenários onde o banco de dados se encontra indisponível para executar uma operação de “insert” ou “update” em um determinado momento, mas pode estar disponível alguns segundos depois, evitando o disparo/alerta de uma mensagem de erro desnecessária.

Durante o primeiro processamento da mensagem, em caso de erro, o sistema aguardará 1 segundo até tentar novamente. Persistindo o erro, ele aguardará 4 e por fim 9 segundos antes da última tentativa. Caso o erro persista mesmo após essas 4 tentativas, a aplicação realiza o registro do erro em log e executa então os passos que estão programados para situações onde o comando não é executado com sucesso.

Para simular um erro forçado, uma requisição de confirmação de venda é realizada no módulo de Vendas por um cliente (figura 25) e, antes de a baixa de estoque do produto ocorrer, ele será marcado com o status de inativo no módulo de produtos por um funcionário (figura 26). Por regra, um produto e seu estoque não poderão ser alterados uma vez que ele esteja com a marcação de inativo.

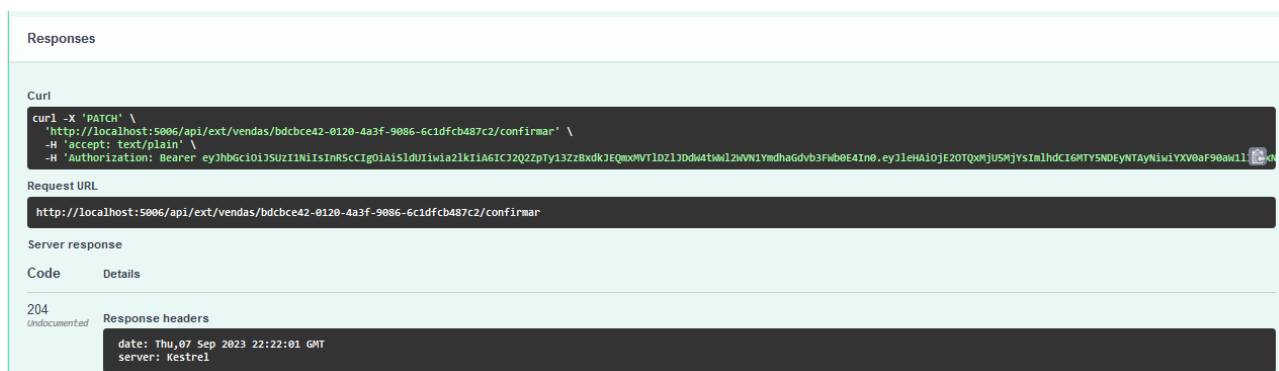


Figura 25: Disparo da solicitação da confirmação de uma venda

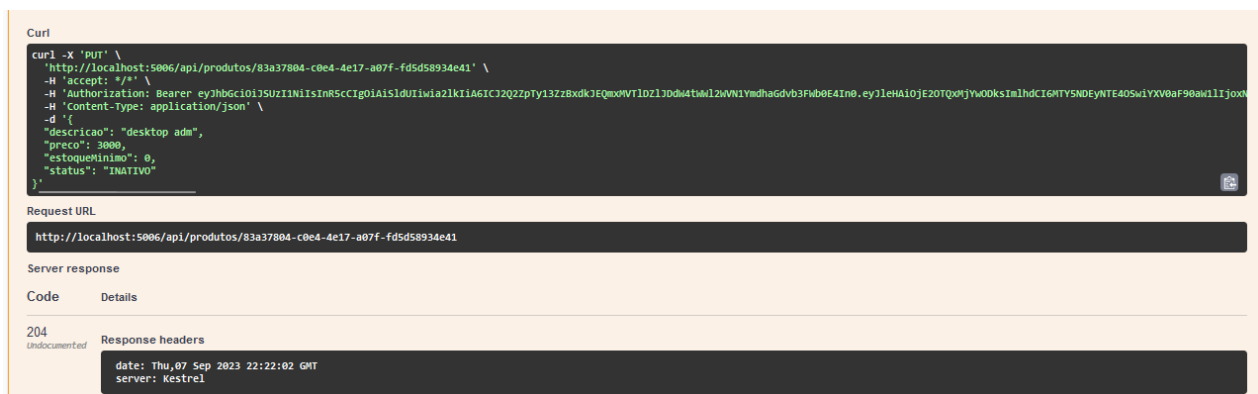


Figura 26: Atualização de status do produto para INATIVO

A aplicação tentará realizar a baixa do produto para essa venda por um total de quatro vezes até que, esgotado o número de tentativas, realiza o registro da exceção (erro) disparada pelo módulo de produtos em log (figura 27) e notifica que a mensagem de confirmação de venda não foi finalizada com sucesso (figura 28), atualizando o status da venda de PROCESSANDO para REPROVADO (figura 29).

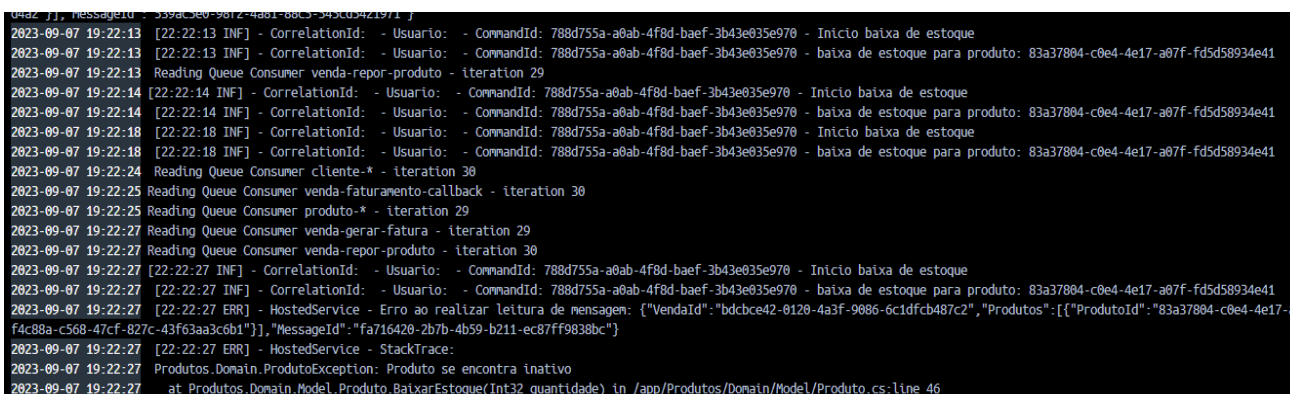


Figura 27: Registro das tentativas de baixa de estoque (registro de log em console do docker)

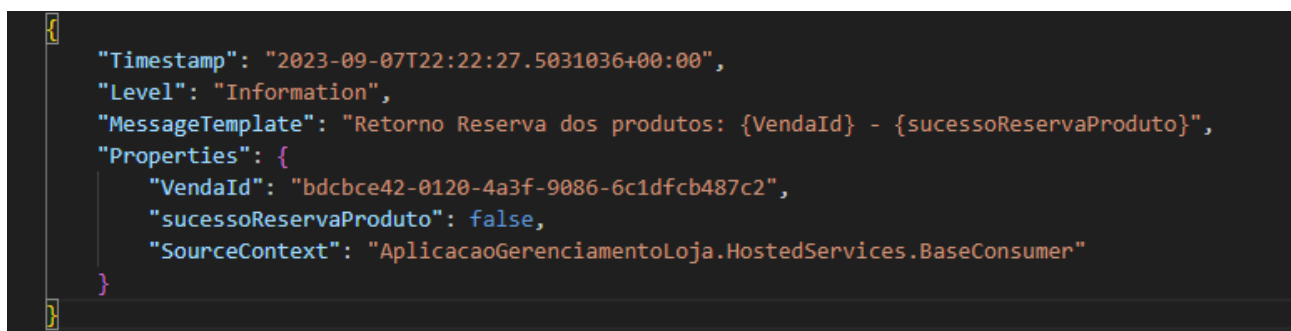
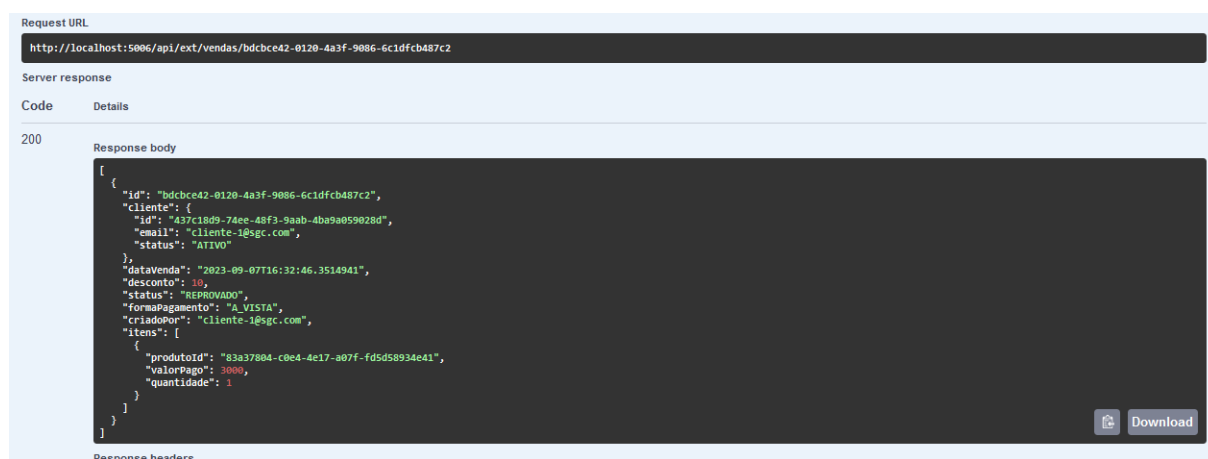


Figura 28: Notificação informando que a reserva do produto não foi concluída com êxito

Podemos observar pelo horário de escrita das mensagens em log que o tempo entre a primeira tentativa de baixa de estoque e a notificação de que houve falha no processo leva

exatamente 14 segundos, conforme estimado previamente, já que o sistema aguarda 1 segundo, 4 segundos e por fim 9 segundos entre cada uma das tentativas.

Apesar de a resiliência ser interessante para a realização de tentativas de comunicação em cenários onde um serviço ou o banco de dados se encontra indisponível, ele pode ocasionar esse acréscimo de tempo no processamento de uma requisição em situações como a do teste descrito.



Response headers

Figura 29: Registro da Venda com status atualizado para Reprovado, visto que o sistema não conseguiu realizar a reserva do produto

Caso o cliente tente reprocessar a venda novamente, ele receberá a informação que o produto está com status “inativo”, até que o status do produto seja alterado novamente.



Figura 30: Solicitação de processamento por parte do cliente após status do produto estar atualizado no módulo de Vendas

4.3.4 CENÁRIO 4 – MONITORAMENTO

Atributo de Qualidade:	Monitoramento
Requisito de Qualidade:	O sistema deve manter log de todas as requisições e processamento de comandos e eventos realizados dentro do sistema.
Preocupação:	

A fim de rastrear possíveis bugs ou explicações para comportamentos inesperados da aplicação, uma vez que se encontre em ambiente de produção, será necessário manter logs de todas as requisições e operações realizadas pela aplicação.	
Cenário(s):	
Cenário 4 – Monitoramento: Aplicação deve manter logs de todos os processos e requisições realizadas.	
Ambiente:	
Sistema em operação normal	
Estímulo:	
Confirmação de compra por parte de um usuário/cliente.	
Mecanismo:	
Utilização da biblioteca Serilog do .NET 6 para manter o registro de requisições e processamento de comandos/eventos dentro do sistema.	
Medida de resposta:	
Todos os registros da operação interna realizada pela aplicação quando uma operação de compra é realizada deverá estar registrada nos arquivos de log gerados pela aplicação. Os logs conterão um “Correlation Id” que auxiliará no rastreamento das mensagens, identificando quais delas pertencem a mesma operação.	
Considerações sobre a arquitetura:	
Riscos:	Descontinuação da biblioteca de modo que seria necessário substituí-la, o que não seria um problema caso a solução para geração dos arquivos de log fosse desenvolvida pelo time de desenvolvimento.
Pontos de Sensibilidade:	Utilização da biblioteca Serilog para o .NET 6, para a criação dos arquivos de log, que serão armazenados em um diretório específico definido nas configurações da aplicação.
Tradeoff:	-

Tabela 9: Cenário 4 da avaliação da arquitetura – Monitoramento

O cenário 4 considerará o requisito de monitoramento em relação a como o processamento de uma requisição realizada por um usuário será registrado em um arquivo físico para eventuais investigações e auditoria.

O registro em log é importante não somente para rastreamento de bugs dentro do sistema, como também para registrar quais ações foram disparadas por quais usuários, que podem ser utilizadas na identificando possíveis falhas nas políticas de acesso utilizadas pela aplicação.

Para teste, será utilizado realizado uma operação de venda (compra por parte de um cliente) comum e todas as ações serão registradas nos logs, desde o disparo da requisição até a notificação de que a venda foi realizada com sucesso.

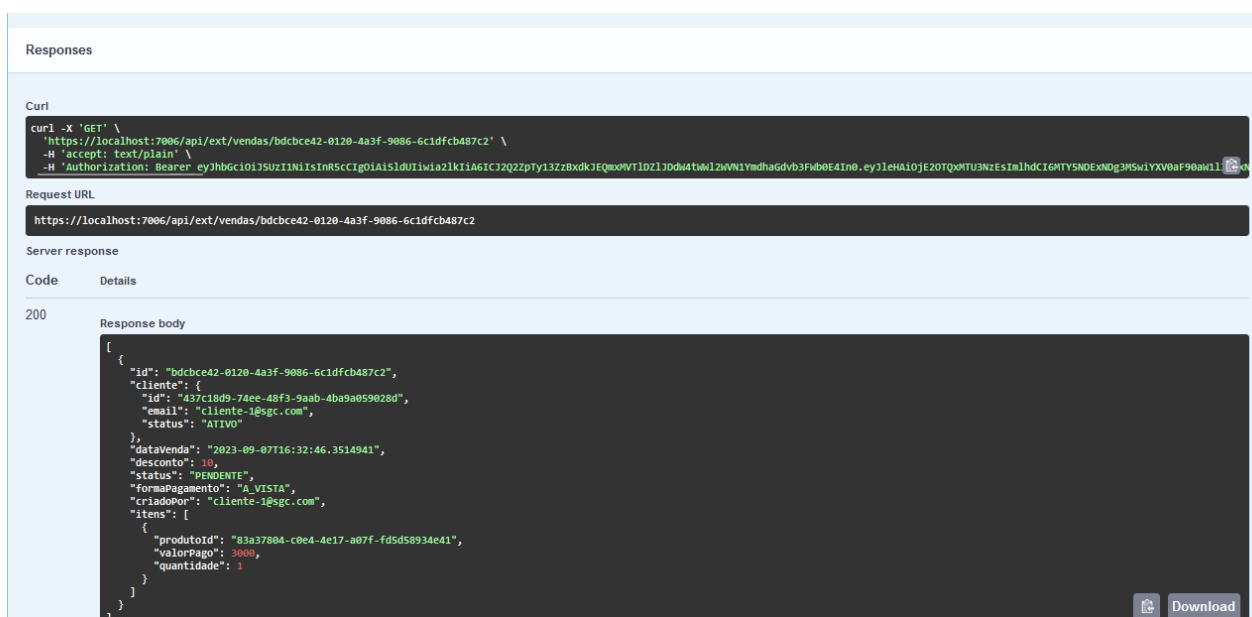


Figura 31: Venda gerada pelo cliente

O cliente envia uma requisição informando que deseja confirmar a sua compra. Essa ação pode ser observada na figura 32. O registro dessa ação é registrada em log, conforme pode ser observado na figura 33.

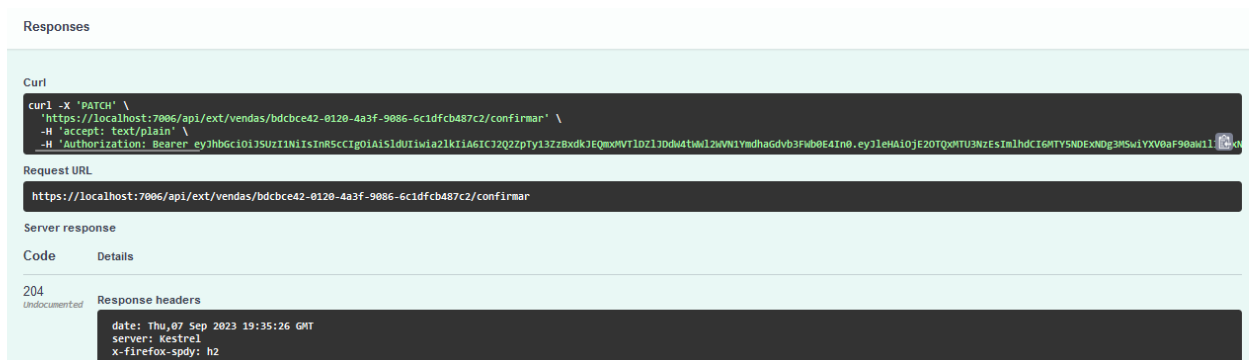


Figura 32: Requisição de confirmação de venda


```
E:\> Programação > Repositories > PUC Minas > SGC - Ecommerce > SistemaDeGerenciamentoComercial > Aplicacao > Aplicacao > Logs > {} logs-20230907.json > ...
1246 {
1247 }
1248 "Timestamp": "2023-09-07T16:35:40.9664987-03:00",
1249 "Level": "Information",
1250 "MessageTemplate": "CommandId: {MessageId} - Inicio baixa de estoque",
1251 "Properties": {
1252   "MessageId": "e919a3b2-cc5b-4642-9782-9d5f069c882a",
1253   "SourceContext": "Produtos.Application.Commands.ProdutoCommandHandler"
1254 }
1255 }
1256 {
1257 "Timestamp": "2023-09-07T16:35:40.9673826-03:00",
1258 "Level": "Information",
1259 "MessageTemplate": "CommandId: {MessageId} - baixa de estoque para produto: {ProdutoId}",
1260 "Properties": {
1261   "MessageId": "e919a3b2-cc5b-4642-9782-9d5f069c882a",
1262   "ProdutoId": "83a37804-c0e4-4e17-a07f-fd5d58934e41",
1263   "SourceContext": "Produtos.Application.Commands.ProdutoCommandHandler"
1264 }
1265 }
1266 {
1267 "Timestamp": "2023-09-07T16:35:40.9689033-03:00",
1268 "Level": "Information",
1269 "MessageTemplate": "CommandId: {MessageId} - Processo de baixa de estoque: {ProdutoId} - Nova Quantidade: {NovaQuantidade}",
1270 "Properties": {
1271   "MessageId": "e919a3b2-cc5b-4642-9782-9d5f069c882a",
1272   "ProdutoId": "83a37804-c0e4-4e17-a07f-fd5d58934e41",
1273   "NovaQuantidade": 11,
1274   "SourceContext": "Produtos.Application.Commands.ProdutoCommandHandler"
1275 }
1276 }
1277 {
1278 "Timestamp": "2023-09-07T16:35:40.9700427-03:00",
1279 "Level": "Information",
1280 "MessageTemplate": "CommandId: {MessageId} - Fim baixa de estoque",
1281 "Properties": {
1282   "MessageId": "e919a3b2-cc5b-4642-9782-9d5f069c882a",
1283   "SourceContext": "Produtos.Application.Commands.ProdutoCommandHandler"
1284 }
1285 }
```

Figura 35: Registros da reserva do produto após confirmação da venda

[illegible]

Figura 36: Registro de venda com status AGUARDANDO PAGAMENTO após o produto ser reservado em estoque

Nessa fase de testes não existe um sistema de pagamentos que já esteja operando com o restante da aplicação. Dessa forma um “Hosted Service” simula o comportamento desse sistema, utilizando uma lógica simples e randômica para definir se o pagamento foi realizado com sucesso ou não.

Após a realização do pagamento, a aplicação recebe a notificação de que ele fora concluído e se foi de fato processado com êxito (figura 37). O status da venda é então alterado mais uma vez

(figura 38) seja para “aprovado”, se o pagamento foi processado com sucesso, ou então “reprovado” (figura 39).

```

> Programação > Repositories > PUC Minas > SGC - Ecommerce > SistemaDeGerenciamentoComercial > Aplicacao > Aplicacao > Logs > logs-2023-09-07
1362 }
1363 {
1364     "Timestamp": "2023-09-07T16:35:46.1960987-03:00",
1365     "Level": "Information",
1366     "MessageTemplate": "Venda {VendaId} pagamento aprovado: {success}",
1367     "Properties": {
1368         "VendaId": "bdcfce42-0120-4a3f-9086-6c1dfcb487c2",
1369         "success": false,
1370         "SourceContext": "AplicacaoGerenciamentoLoja.HostedServices.BaseConsumer"
1371     }
1372 }

```

Figura 37: Mensagem recebida pela aplicação web informando que o pagamento não foi efetuada com sucesso

```
E: > Programação > Repositories > PUC Minas > SGC - Ecommerce > SistemaDeGerenciamentoComercial > Aplicacao > Aplicacao > Logs > logs-20230907.js
1381 }
1382 {
1383     "Timestamp": "2023-09-07T16:35:46.5073560-03:00",
1384     "Level": "Information",
1385     "MessageTemplate": "CommandId: {MessageId} - Venda reprovada: {vendaId}",
1386     "Properties": {
1387         "MessageId": "faacc5db-b56e-4cdb-b45f-3d0c53bd8659",
1388         "vendaId": "bdcfce42-0120-4a3f-9086-6c1dfcb487c2",
1389         "SourceContext": "Vendas.Application.Commands.Handlers.VendaCommandHandler"
1390     }
1391 }
```

Figura 38: Comando enviado pela aplicação web para que o módulo de vendas atualize o status da Venda em questão

[illegible]

Figura 39: Venda com status de reprovada após comunicação de que o pagamento não foi efetuado com sucesso

5 AVALIAÇÃO CRÍTICA DOS RESULTADOS

Considerando os requisitos que foram estabelecidos no início do projeto, a aplicação atende de forma satisfatória a todos eles. Vale ressaltar, no entanto, que nenhuma arquitetura é perfeita e a decisão de adotar determinadas tecnologias ou soluções arquiteturais terá impactos positivos e negativos na composição final do sistema, sendo necessário escolher aqueles que trarão mais benefícios do que prejuízos, dentro do que fora solicitado inicialmente.

Por exemplo, considerando o sistema tratado neste trabalho, temos uma aplicação base desenvolvida utilizando o framework .NET 6 que opera como um monólito. Esta solução, apesar de remover aspectos que adicionariam complexidade ao sistema final, como a necessidade de ter diversos “microssistemas” operando independentemente e que deveriam ser orquestrados para se comunicarem via rede, impõe a necessidade de “deploy” de toda a aplicação sempre que uma correção ou nova funcionalidade for implementada, independente do que seja.

Ainda a respeito do estilo arquitetural adotado, sistemas que apresentam uma arquitetura monolítica muitas vezes caem, mesmo que não-intencionalmente, em um cenário de acoplamento rígido entre diversas partes do sistema. Isso ocorre quando diferentes serviços são utilizados em um mesmo processo, como no exemplo do processamento de uma venda, já apresentado anteriormente: para que a venda seja processada é necessário verificar o status do cliente, processar o pagamento, realizar a baixa do produto em estoque e então atualizar o status da venda.

No entanto, a aplicação web do SGC, mesmo consistindo de uma arquitetura monolítica, foi projetada de modo que os seus diferentes componentes internos não se comunicam diretamente entre si, eliminando qualquer possibilidade de acoplamento entre serviços, mantendo os módulos independentes e que podem ser alterados e testados independentemente.

Todavia, como fora mencionado no início dessa seção, toda solução apresentará vantagens e desvantagens quando implementada em um sistema. A adoção de uma estrutura de módulos independentes que se comunicam de forma assíncrona melhorará a manutenibilidade do sistema, mas o deixará mais complexo para os desenvolvedores que não tiveram contato com o código e não conhecem a estrutura da aplicação, requerendo estudo para sua correta manutenção.

Outro ponto a ser mencionado, a adoção do SQLite como mecanismo de persistência de dados visa disponibilizar o banco de dados em um arquivo que pode ser facilmente copiado e armazenado para backup por administradores de sistemas que não estão familiarizados com os procedimentos mais complexos que são requeridos por outros sistemas de gerenciamento de bancos de dados, como o Microsoft SQLServer e o PostgreSQL, além de ser uma ferramenta que pode ser utilizada gratuitamente.

O SQLite, no entanto, irá se comportar como um limitador de performance a medida que o fluxo de dados na aplicação aumentar, visto que ele não é projetado para esse tipo de operação e funciona a partir de apenas uma máquina (já que consiste exclusivamente de um arquivo de extensão “.db”), não havendo uma forma de escaloná-lo.

Para pequenas empresas e lojas de “e-commerce” que estão iniciando na área, o SQLite como tecnologia de persistência de dados é o bastante. Além disso, quando for necessário substituí-lo, a alteração do código da aplicação será simples, requerendo apenas a alteração das bibliotecas e

configurações (connection strings) utilizadas pelas camadas de persistência de cada um dos módulos.

Essa camada foi projetada de modo que toda alteração necessária no processo de migração de banco de dados fosse restringida a um único arquivo, o *DbContext*, isolando a tecnologia de persistência do restante da aplicação, que a utiliza através de uma interface generalista.

Uma análise semelhante pode ser feita com relação a utilização do Redis como tecnologia responsável pelo papel de message broker dentro do sistema: é simples de implementar e cumpre com o aquilo que é esperado dentro do cenário apresentado (comunicação dos módulos internos em um sistema voltado a pequenas/médias empresas, cujo fluxo de operação não é massivo).

O Redis, no entanto, não apresentará algumas das funcionalidades encontradas em outros sistemas mais populares de comunicação assíncrona, como o RabbitMQ, como a ausência de suporte nativo à persistência das mensagens: Caso a conexão com o Redis seja perdida, as mensagens que estão na fila podem nunca chegar ao seu destino final, requerendo um desenvolvimento a parte do lado da aplicação para lidar com esse tipo de cenário (<https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-redis/>).

A aplicação, por sua vez, utiliza o Redis como solução de mensageria através de uma interface generalista de modo que, desejando substituí-lo pelo RabbitMQ por exemplo, o impacto na aplicação será mínimo.

Podemos observar que muitas das decisões tomadas no desenvolvimento da arquitetura desse sistema busca atender as demandas de pequenas/médias empresas de “e-commerce” e, em caso de aumento do fluxo de usuários e operação, o sistema pode não desempenhar de forma satisfatória, requerendo substituição de algumas das tecnologias adotadas.

Já levando em conta essa necessidade de expansão que pode surgir no futuro, a aplicação foi projetada para que a substituição dessas tecnologias possa ser realizada de forma simples e sem nenhum impacto nas regras de negócio.

6 CONCLUSÃO

O projeto arquitetural apresentado nesse trabalho visa atender as demandas de pequenas e médias empresas de “e-commerce”, não restringindo a aplicação a esse cenário, deixando-a apta para que novos desenvolvimentos e a sua expansão possam ser realizados de forma natural, sem a necessidade de remodelar toda a estrutura do sistema.

Dessa forma, optou-se por uma arquitetura monolítica mais simples, mas utilizando-se de comunicação assíncrona entre os seus componentes para evitar o acoplamento entre os mesmos. As tecnologias adotadas visavam a sua simplicidade de implementação e manutenção, como o Redis como “message broker” e SQLite como solução para persistência de dados, mas que não escalonam bem em performance quando passamos para um cenário de alto fluxo de dados e um número elevado de operações por segundo.

Além disso, o emprego do Keycloak como sistema de gerenciamento de usuários remove essa funcionalidade da aplicação Web, deixando-a responsável exclusivamente pelas operações de negócio (gerenciamento de clientes, de produto, de estoque e de vendas), mas requer que uma configuração extra seja realizada para que ele opere corretamente dentro do sistema, o que adiciona uma camada a mais de complexidade na arquitetura.

O projeto possui ainda vários pontos em que pode ser aprimorado, dentre os quais a implementação de testes unitários, a inclusão de um banco de dados para o sistema de gerenciamento de usuários e desenvolvimento e automação de uma esteira de entrega do software em produção (implementando os conceitos de CI/CD).

Como já fora mencionado anteriormente, não há arquitetura perfeita e a adoção de determinados mecanismos será baseado no cenário, nos requisitos funcionais e não-funcionais, bem como no balanço entre os pontos positivos e negativos que cada solução entregará.

O desenvolvimento desse projeto mostrou como esses pontos positivos e negativos devem ser abordados e como escolher a melhor solução a partir do que fora solicitado. As pesquisas realizadas foram importantes na questão do conhecimento e a oportunidade de implementar diferentes tecnologias e testar diferentes abordagens no desenvolvimento dessa aplicação foi uma experiência singular, desafiadora, mas muito valiosa.

7 REFERENCIAS BIBLIOGRÁFICAS

Ecommerce: The History and Future of Online Shopping. BigCommerce. 2022. Disponível em: <https://www.bigcommerce.com/articles/ecommerce/>. Acesso em: 08 de Abril de 2023.

O que é e-commerce e para que serve? Exame. 2022. Disponível em: <https://exame.com/invest/guia/o-que-e-e-commerce-e-para-que-serve/>. Acesso em: 09 de Abril de 2023.

KEELING, Michael. *Design It! From Programmer to Software Architect*. 1ª ed, O'Reilly Media, 2017.

LAUDON, Kenneth C.; TRAVER, Carol Guercio. *E-commerce*. 13ª ed, Pearson Education, 2017.

RICHARDS, Mark; FORD, Neal. *Fundamentals of Software Architecture: An Engineering Approach* 1ª ed, O'Reilly Media, 2020.

SOMMERVILLE, Ian. *Software Engineering*. 9ª ed, Pearson Education, 2009.

KAZMAN, Rick; KLEIN, Mark; BARBACCIO, Mario; LONGSTAFF, Tom; LIPSON, Howard; CARRIERE, Jeromy. *The Architecture Tradeoff Analysis Method*. July 1998.

What's the Difference Between RabbitMQ and Redis? Disponível em: <https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-redis/>. Acesso em: 07 de Setembro de 2023.

Server Administration Guide: Keycloak Features and Concepts. Disponível em: https://www.keycloak.org/docs/latest/server_admin/. Acesso em: 07 de Setembro de 2023.