UNIVERSIDADE FEDERAL DE ALFENAS

JOÃO PEDRO GIANDOSO MACHADO
IAGO VINICIUS SIQUEIRA RIBEIRO

IMPLEMENTAÇÃO DE UM SISTEMA DE FATURAMENTO COM INTERFACES MÓVEIS E WEB

ALFENAS/MG
2024

JOÃO PEDRO GIANDOSO MACHADO
IAGO VINICIUS SIQUEIRA RIBEIRO

IMPLEMENTAÇÃO DE UM SISTEMA DE FATURAMENTO COM INTERFACES MÓVEIS E WEB

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção de título de Bacharel em Ciência da Computação, pela Universidade Federal de Alfenas.
Orientadora: Profª. Dra. Mariane Moreira de Souza

ALFENAS/MG
2024

Sistema de Bibliotecas da Universidade Federal de Alfenas
Biblioteca Unidade Educacional Santa Clara

Machado, João Pedro Giandoso .
IMPLEMENTAÇÃO DE UM SISTEMA DE FATURAMENTO COM INTERFACES MÓVEIS E WEB / João Pedro Giandoso Machado, Iago Vinicius Siqueira Ribeiro. - Alfenas, MG, 2024.
70 f. : il. -

Orientador(a): Mariane Moreira de Souza.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Federal de Alfenas, Alfenas, MG, 2024.
Bibliografia.

1. Software. 2. Motor de faturamento. 3. Painel administrativo. 4. Aplicativo móvel. I. Siqueira Ribeiro, Iago Vinicius . II. Moreira de Souza, Mariane, orient. III. Título.

Ficha gerada automaticamente com dados fornecidos pelo autor.

JOÃO PEDRO GIANDOSO MACHADO
IAGO VINICIUS SIQUEIRA RIBEIRO

IMPLEMENTAÇÃO DE UM SISTEMA DE FATURAMENTO COM INTERFACES MÓVEIS E WEB

A Presidente da banca examinadora abaixo assina a aprovação do Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção de título de Bacharel em Ciência da Computação, pela Universidade Federal de Alfenas.

Aprovada em: ___ de _____ de 2024.

Profª. Dra. Mariane Moreira de Souza — Assinatura: _____
Universidade Federal de Alfenas

Prof. Dr. Flavio Barbieri Gonzaga — Assinatura: _____
Universidade Federal de Alfenas

Prof. Dr. Rodrigo Martins Pagliares — Assinatura: _____
Universidade Federal de Alfenas

RESUMO

O presente trabalho detalha o desenvolvimento de uma solução tecnológica relacionada a um sistema transacional entre clientes e lojistas. Tal sistema é composto por um aplicativo móvel com webview, um painel administrativo e um motor de faturamento assíncrono com tecnologias de fila para processamento de tarefas e de *cache*. O motor de faturamento foi desenvolvido através de uma API Rest implementada com o FastAPI, consumindo dados através do ORM SQLAlchemy e também de um banco Redis em caso de *cache*, adotando a estratégia de *cache write-through* para acelerar o tempo de resposta de totalizadores importantes. Detalha-se também a implementação de uma fila de processos HTTP, com um modelo de produtor e consumidor capaz de garantir um aceite não bloqueante de requisições com resiliência, utilizando o RabbiMQ como base para o sistema de mensageria. O painel administrativo emprega como arquitetura o MTV associado ao Jinja e ao HTMX para o desenho de telas HTML, sendo utilizado também o FastAPI como servidor web HTTP. Utilizando escolhas tecnológicas semelhantes temos um website criado com interface de aplicações móveis, sendo servido com experiência de aplicativo através das lojas de aplicativos móveis com a utilização de um componente de *webview* utilizando o Flutter. Notou-se durante o processo de desenvolvimento a liberdade e a facilidade promovidas pelas tecnologias utilizadas, tornando o software simples e o processo de desenvolvimento acelerado.

Palavras-Chave: Software; Motor de faturamento; Painel administrativo; Aplicativo móvel.

ABSTRACT

The present work details the development of a technological solution related to a transactional system between customers and retailers. This system consists of a mobile application with webview, an administrative panel, and an asynchronous billing engine with queue technologies for task processing and caching. The billing engine was developed through a REST API implemented with FastAPI, consuming data through the SQLAlchemy ORM and also a Redis database for caching, adopting the write-through cache strategy to speed up the response time of important totals. It also details the implementation of an HTTP process queue, with a producer-consumer model capable of ensuring non-blocking acceptance of requests with resilience, using RabbitMQ as the basis for the messaging system. The administrative panel employs the MTV architecture associated with Jinja and HTMX for HTML screen design, with FastAPI also being used as the HTTP web server. Using similar technological choices, we have a website created with a mobile application interface, served with an app experience through mobile app stores using a webview component with Flutter. It was noted during the development process the freedom and ease promoted by the technologies used, making the software simple and the development process accelerated.

Keywords: Software; billing engine; administrative panel; mobile application.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

ORM	Object Relational Mapping
RF	Requisitos Funcionais
RNF	Requisitos Não Funcionais
DAO	Data access object
MTV	Model-Template-View
MVC	Model-View-Controller
REST	Representational State Transfer
API	Application Programming Interfaces
SQL	Structured Query Language
HTML	HyperText Markup Language
AMQP	Advanced Message Queuing Protocol
JSON	JavaScript Object Notation
URI	Uniform Resource Identifier
DTO	Data Transfer Object
PDV	Ponto de Venda

SUMÁRIO

1 INTRODUÇÃO

A criação de um novo software tem como intuito a resolução de problemas específicos, seja por meio da sistematização de dados a fim de viabilizar a venda de um produto, seja para controlar uma rotina operacional. A necessidade de migração de um software pode surgir no contexto de uma empresa privada por diversos motivos, e um deles é a falta de liberdade de modificação ou ausência do direito legal de sustentar o próprio projeto.

Com base em um problema semelhante enfrentado pela empresa OQTem [1], antiga AbmCash, o presente trabalho apresenta o desenvolvimento de um motor de faturamento, um aplicativo móvel e um painel administrativo, capazes de efetuar o gerenciamento das transações, a composição de faturas e emissão de relatórios de consumo.

Para o desenvolvimento da nova tecnologia o padrão arquitetural MTV foi utilizado com o *framework* FastAPI [2] para definição de rotas de API, e o HTMX [3] associado ao Jinja [4] para fazer o desenho do HTML. A gestão de dados se deu através do *framework* SQLAlchemy [5] e para o aplicativo móvel, além da estrutura de telas que utiliza as tecnologias mencionadas acima, utilizou-se o Flutter [6], ferramenta capaz de renderizar uma *WebView* possibilitando uma distribuição multiplataforma (IOs e Android).

A implementação desse sistema leva em consideração os produtos e componentes de *software* que já estavam presentes em operação na empresa, sendo eles: o banco de dados relacional MySQL [7] e um painel PHP que oferece gestão de clientes.

Este trabalho, portanto, apresenta a implementação dos novos componentes descritos acima, além de detalhar a construção de um sistema de filas capaz de viabilizar o processamento intensivo de chamadas HTTP com o gerenciador de filas RabbitMQ [8]. Nós também apresentamos o desenvolvimento de um mecanismo de cache write-through com o banco de dados em memória Redis [9] para acelerar a resposta de certas rotas que causam grande impacto no banco de dados ou demandam alto poder de processamento do servidor.

2 OBJETIVOS

2.1 OBJETIVO GERAL

O objetivo geral deste trabalho é resolver o problema da empresa através do desenvolvimento de *softwares* capazes de conduzir a operação, sendo eles de um motor de faturamento, um painel administrativo e um aplicativo móvel, aprimorando a funcionalidade e eficiência do sistema.

2.2 OBJETIVOS ESPECÍFICOS

A concretização do objetivo geral se dá por meio dos seguintes objetivos específicos:

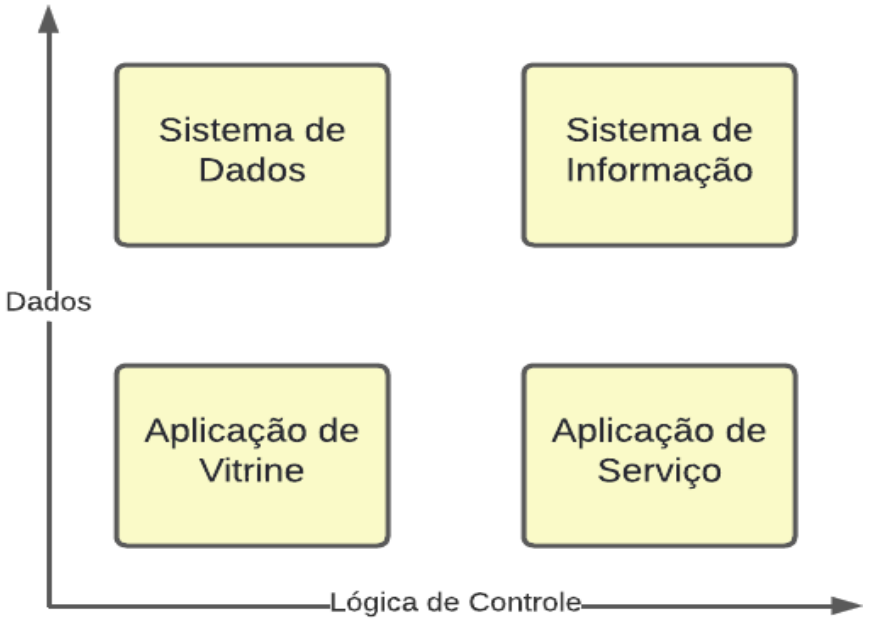
- Permitir retrocompatibilidade com o antigo motor de faturamento;
- Desenvolver e disponibilizar aplicativo móvel com *webview*;
- Desenvolver e disponibilizar um painel administrativo para gestão de clientes de pedidos;
- Implementar processamento assíncrono de transações;
- Implementar e gerenciar *cache* para cálculo de totalizadores.

3 REFERENCIAL TEÓRICO

3.1 APLICAÇÕES WEB

Define-se como Aplicação *Web* um *software* que depende da *web* para sua execução [10], sendo possível variações entre uma simples página *web* até programas sofisticados. Tais aplicações podem ser agrupadas de acordo com sua necessidade de dados e complexidade de controle, como exemplificado na Figura 1 com os elementos descritos em subsequência:

Figura 1 - Classificações de Aplicações Web



Fonte: Autor (2023).

A aplicação de Vitrine é composta por páginas web estáticas, que tendem a não possuir lógica de programação. Seu principal objetivo é a apresentação de imagens e textos.

A Aplicação de Serviço é uma camada orientada a serviços que contém a lógica de programação necessária para implementar o serviço. O *layout* dos dados é, geralmente, uma preocupação secundária.

A Aplicação de Dados consiste em um site que fornece uma interface para navegar e consultar grande quantidade de dados. A ênfase principal em tais aplicativos está nos dados, com uma quantidade mínima de lógica de programação envolvida.

Os Sistemas de informação combinam as Aplicações Orientadas a Serviços e as Aplicações Intensivas em Dados. Nesse modelo existe a preocupação com o fluxo de dados para navegação e a visualização de dados, além de fluxos de controle para as diferentes fases envolvidas na manipulação dos dados [11]. No desenvolvimento deste trabalho discutiremos a implementação de um sistema de informação.

3.2 *FRONT-END* E *BACK-END*

O desenvolvimento *front-end* refere-se à criação e implementação da interface de usuário de aplicações *web* e móveis. Essa área se concentra na experiência do usuário (UX) e na interface do usuário (UI), garantindo que as interações sejam intuitivas, eficientes e esteticamente agradáveis. As linguagens fundamentais para o desenvolvimento *front-end* incluem HTML (Linguagem de Marcação de Hipertexto), CSS (Folhas de Estilo em Cascata) e JavaScript [12], contando também com diversos *frameworks* como o React [13], HTMX [3] e Bootstrap [14]. Aqui utilizaremos o HTMX e o Bootstrap para o desenho de *front-end*.

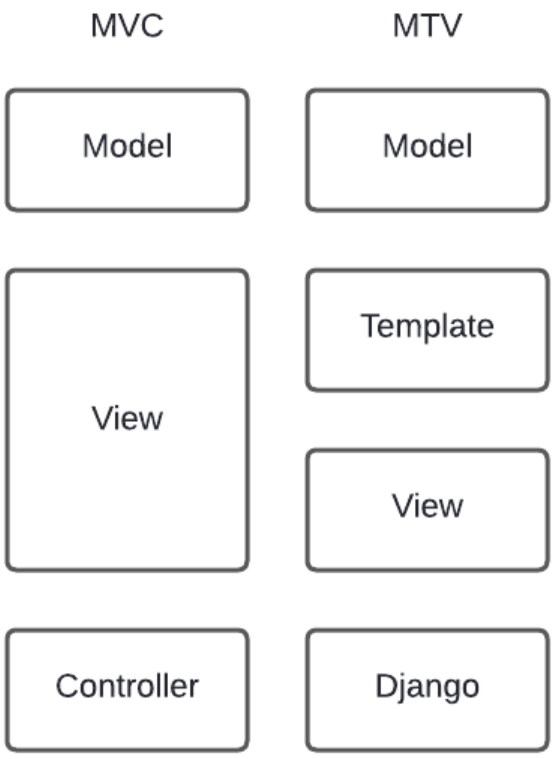
Back-end refere-se à parte “invisível” de um aplicativo ou site. Tal desenvolvimento opera nos bastidores, logo é responsável por gerenciar e armazenar dados e garantir que tudo funcione corretamente ao mesmo tempo que se comunica com o *front-end*. O *front-end*, como dito anteriormente, é a área que os usuários vêem e interagem diretamente, como por exemplo a interface gráfica de um *site*.

Existem muitas linguagens para *back-end*, como Node, Python, Ruby e PHP ou Java, além de incontáveis *frameworks* como Laravel [15], Rails [16], FastApi [2] e Spring [17]. Neste trabalho nós utilizamos o FastApi para os módulos de *back-end*.

3.3 PADRÃO ARQUITETURAL MTV

O padrão Model-Template-View(MTV) foi implementado pelo *framework* Django. Apesar de ser um padrão arquitetural similar ao Model-View-Controller(MVC), nele ocorre uma separação das responsabilidades da camada *view*, enquanto a camada *controller* é manipulada pelo *framework*, como exemplificado na Figura 2:

Figura 2 - Comparativo entre MVC e MTV



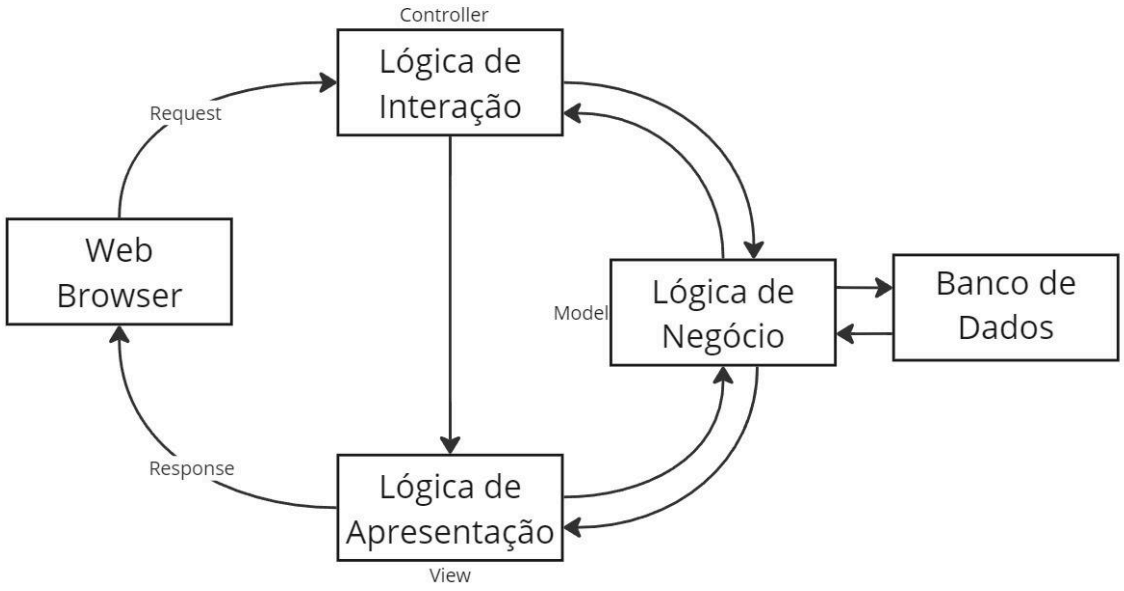
Fonte: Autor (2023)..

Ou seja, o padrão arquitetural MTV possui as seguintes camadas:

- *Model*: contém tudo sobre dados (como acessá-los ou validá-los, além de tratativas específicas);
- *Template*: contém decisões relacionadas a apresentação ou exibição de algo ao usuário;
- *View*: é a camada que contém a lógica de apresentação, definindo quais dados devem ser apresentados.

Com a utilização desse padrão arquitetural, resta ao desenvolvedor a definição da *view* que irá tratar a requisição do usuário — que irá consultar classes de modelo responsáveis pela lógica de acesso à base de dados. Assim, os dados são encaminhados ao *template*, que os formata para o padrão em que serão apresentados na Figura 3. Para todos os módulos apresentados neste trabalho nós utilizamos o padrão arquitetural MTV.

Figura 3 - Fluxo de arquitetura MTV



Fonte: Adaptado de [10].

3.4 REST API E FASTAPI

Application Programming Interface (API) é uma interface que nos permite interagir com aplicações programaticamente [18].

A arquitetura REST é amplamente utilizada para desenvolver API's que são consumidas em diferentes plataformas e ambientes. As API RESTful, também conhecidas como *web* API, consistem em *endpoints* acessados pelos métodos HTTP GET, POST, PUT e DELETE. Cada *endpoint* possui uma URI, ou seja, o endereço pelo qual é invocado. Uma das linguagens utilizadas para a comunicação entre API é o JSON, formato em texto que tem fácil identificação e processamento pelas máquinas [19].

Por muito tempo, conectar e manipular dados diretamente do banco de dados era considerado um desafio para os desenvolvedores, por essa razão a maioria dos profissionais preferem a implementação de API para o desenvolvimento *web* [20]. O FastApi [2], escolha popular para a criação de REST API [18], é um *framework* para Python altamente performático. Ele é mais moderno e também considerado mais rápido do que o seu concorrente, o Flask [21], devido a sua construção feita em torno de paradigmas de programação assíncrona, o que permite tratar mais requisições em paralelo [22].

Na seção 4.5 do presente trabalho detalharemos um módulo que conta com definição de uma API *Rest* utilizando FastApi.

3.5 MAPEAMENTO OBJETO-RELACIONAL E SQLALCHEMY

Do ponto de vista do desenvolvedor, o ORM(Object Relational Mapping) abrange soluções para mapear objetos para dados relacionais, separando questões de persistência em uma camada de abstração oferecida pela ferramenta [23]

O SQLAlchemy é um *framework* conhecido para a linguagem Python, capaz de oferecer o desenvolvimento acelerado graças a recursos como auto-carregamento de tabelas [24]. Uma das principais vantagens do SQLAlchemy é sua capacidade de realizar a introspecção dos atributos da tabela de forma

dinâmica e automática. Isso significa que o código pode se adaptar às estruturas definidas do banco de dados sem a necessidade de definições manuais e extensivas. Sua flexibilidade resulta em um desenvolvimento mais ágil e reduz a possibilidade de erros decorrentes de incompatibilidade com o esquema do banco de dados [24].

O trecho Código 1 mostra todo o código necessário para criação de um objeto ORM de alto nível, dando acesso a todos os recursos de *Query*:

Código 1 - Definição de tabela utilizando SQLAlchemy

```
1.    class Clientes(Base):
2.        __tablename__ = 'clientes'
3.        __table_args__ = {'autoload': True}
```

Fonte: Autor (2023).

Outro benefício significativo do SQLAlchemy é sua capacidade de criar consultas SQL utilizando estruturas de alto nível. Isso torna a interação com o banco de dados mais intuitiva e fácil de implementar, graças aos recursos de geração automática de consultas SQL, ORM e suporte a expressões complexas. O SQLAlchemy proporciona uma experiência de desenvolvimento simplificada e eficiente [25].

Os trechos Código 2, Código 3, Código 4 e Código 5 demonstram um comparativo de inserção e consulta utilizando a linguagem de Queries do SQLAlchemy e o relativo SQL nativo:

Código 2 - Inserção de dados utilizando SQLAlchemy

```
1.  # SQLAlchemy
2.   db.add(
3.      Clientes(
4.         COD_CLIENTE=12345,
5.         SALDO_ATUAL=100.0,
6.         NOME='FULANO DOS SANTOS'
7.      )
8.   )
```

Fonte: Autor (2023).

Código 3 - Inserção de dados utilizando SQL Nativo

```
1.  # SQL
2.  INSERT INTO Clientes (COD_CLIENTE, SALDO_ATUAL, NOME)
3.  VALUES (12345, saldo_atual, 'FULANO DOS SANTOS');
```

Fonte: Autor (2023).

Código 4 - Query com filtro utilizando SQLAlchemy

```
1.  # SQLAlchemy
2.  res = self.db.query(Clientes.NOME) \
3.     .filter(Clientes.CODIGO == '12345') \
4.     .one()
5.  print(f'Nome: {res.NOME}') # FULANO DOS SANTOS
```

Fonte: Autor (2023).

Código 5 - Query com filtro utilizando SQL Nativo

```
1.  # SQL
2.  SELECT * FROM Clientes WHERE CODIGO = '12345'
```

Fonte: Autor (2023).

É válido ressaltar que utilizando o *framework* não é necessário fazer o *parsing* da resposta, assim como demonstrado no bloco de Código 4 na linha 5. No

desenvolvimento deste trabalho nós utilizamos o SQLAlchemy para efetuar toda a comunicação com o banco de dados.

3.6 MOBILE *WEBVIEW*

Um dos principais problemas no desenvolvimento de aplicativos móveis é a alta fragmentação de uma plataforma móvel, além da necessidade de suporte para as várias plataformas disponíveis no mercado. Uma abordagem para evitar esse problema é a utilização de uma *webview* que, de forma sucinta, consiste em um aplicativo móvel capaz de integrar uma página web em sua interface.

Tal abordagem permite que o desenvolvedor obtenha o benefício do aplicativo da Web na programação móvel nativa, pois possibilita que os aplicativos exibam o conteúdo da Web sem a necessidade de um extensivo desenvolvimento por parte do aplicativo móvel. Nesse caso, o desenvolvedor deve manter múltiplas bases de código: uma para a aplicação web e outra para a implementação nativa de cada plataforma móvel que desejar fazer a publicação [26]. Na seção 4.6 deste trabalho detalhamos uma implementação de aplicativo móvel com as tecnologias mencionadas acima.

3.7 MENSAGERIA E RABBITMQ

O sistema de mensageria é uma estratégia adotada por sistemas distribuídos. Ele possibilita a comunicação baseada na publicação de eventos em uma fila de processos, os quais serão consumidos a partir desta. É composto por um produtor e um consumidor, utilizando uma tecnologia de gestão de filas: o RabbitMQ. Este último, por sua vez, trata-se de uma solução gratuita e *open-source* que opera sendo um *broker* de mensagens, implementando o protocolo *Advanced Message Queue Protocol* (AMQP), permitindo a comunicação entre componentes sem preocupações com perda de mensagens. A utilização dessa tecnologia de troca de mensagens baseada em filas permite que os produtores enviem mensagens para

uma fila centralizada, enquanto os consumidores recebem e processam essas mensagens de forma assíncrona [27].

Para utilizar o sistema de mensageria através de um código Python, existe a biblioteca Pika. Ela oferece funções e classes para se conectar e interagir com o RabbitMQ, o que inclui abrir uma conexão com o referido servidor, declarar filas, publicar mensagens e assinar filas para receber mensagens [28].

Como o RabbitMQ é um projeto *open-source*, é necessário fazer o *deploy* do sistema de mensageria por conta própria. Uma maneira simples de alcançar esse resultado é a utilização da imagem Docker oficial fornecida pelo próprio sistema de mensagens RabbitMQ [29]. Neste trabalho, a utilização do sistema de mensageria é um componente fundamental para alcançar o comportamento assíncrono (não bloqueante) e será detalhado na seção 4.5.2.

3.8 CACHE E REDIS

O Redis é um banco em memória, capaz de diminuir o custo das operações de cálculo envolvidas com a operação financeira mantendo o totalizador em memória [30]. Ao utilizar a estratégia de *caching* nos totalizadores é preciso haver estratégias de invalidação de *cache* para operações de escrita, e existem diversas formas para alcançar tal feito. Uma delas é o cache *write-through*: sempre que um item é atualizado na base principal, ele é atualizado simultaneamente no *cache*. O princípio é que, sempre que os dados são calculados, eles são armazenados no *cache* imediatamente após serem obtidos ou calculados a partir da fonte principal, o que garante performance e dados consistentes ao custo de latência durante as operações de escrita [31]. No presente trabalho nós utilizamos o Redis para efetuar o armazenamento de totalizadores que representam um alto custo de processamento.

3.9 HTMX

HTMX é uma biblioteca capaz de oferecer uma interface de chamadas AJAX diretamente do HTML, podendo, assim, substituir o Javascript em alguns cenários. Por meio do uso do *framework*, cada atributo define também o verbo da chamada que será realizada. Seu conteúdo é a URL para emitir uma chamada AJAX, esta que será enviada no momento que o elemento for acionado, como mostra o Código 6:

Código 6 - Exemplo de chamada AJAX usando HTMX

```
1. <div hx-put="/messages">
2.     Put To Messages
3. </div>
```

Fonte: Autor (2024).

No Código 6 podemos observar o atributo *hx-put* na linha 1, ele é oferecido pelo *framework* e é o responsável pelo comportamento AJAX dentro do contexto apresentado. Neste trabalho, o HTMX é um dos principais mecanismos para efetuar interação do tipo AJAX entre *front-end* e *back-end*.

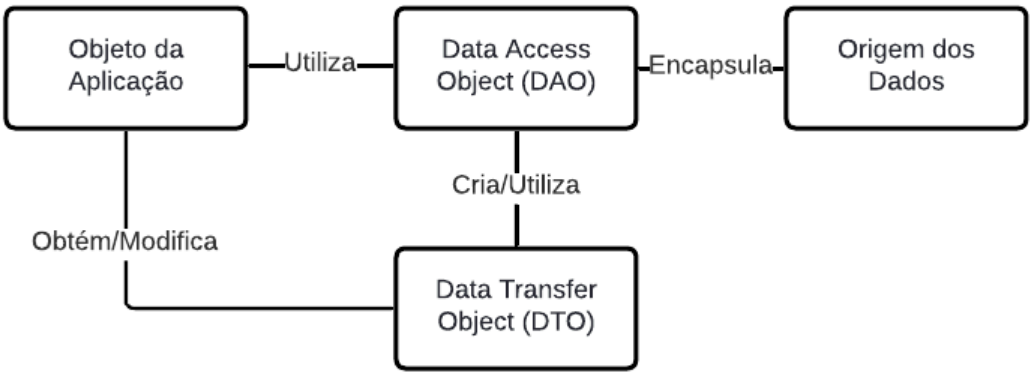
3.10 DAO E DTO

O padrão de *Data Access Object* (DAO) é utilizado para abstrair a forma com que os dados são manipulados na aplicação, ou seja, por meio de uma classe que implementa esse padrão. A aplicação desconhece se os dados estão vindo de um banco de dados, de uma chamada para uma API ou de um arquivo gravado no disco.

Ao utilizar a classe DAO, o usuário recebe ou envia um objeto do tipo DTO (*Data Transfer Object*), isto é, esse objeto contém os dados a serem enviados ou que foram trazidos da origem (banco de dados, arquivo ou outro tipo de fonte) [32]. Exemplificando: supõe-se uma aplicação que tenha dados de produtos persistidos. Buscamos esses produtos implementando uma classe que podemos chamar de ProdutoDAO (classe que utiliza o padrão *DAO*), onde acessamos um método que resgata os produtos da origem de dados. O resultado da busca traz uma lista de

objetos da classe ProdutoDTO (classe que utiliza o padrão *DTO*). A Figura 4 mostra um modelo de classes dos padrões DAO e DTO:

Figura 4 - Diagrama de Classes dos padrões DAO e DTO



Fonte: Autor (2023).

Nós vamos utilizar DAOs e DTOs no presente trabalho para a implementação dos módulos descritos nas seções 4.5 à 4.7.

4 DESENVOLVIMENTO DO SISTEMA DE FATURAMENTO

Dentro do escopo deste trabalho, nós desenvolvemos um motor de transações capaz de controlar um fluxo de transações financeiras, além das interfaces relacionadas a gestão de transações e faturas, contemplando também a emissão de relatórios. Tais interfaces têm o objetivo de viabilizar a utilização e gestão das funcionalidades do motor para clientes nas mais diversas plataformas.

Nos subseções seguintes serão definidos atores, requisitos e trâmites de desenvolvimento.

4.1 ATORES

Os atores envolvidos são:

- Clientes
- Lojistas
- Empregadores
- Administradores

A Tabela 1 apresenta os principais objetivos de cada ator, sendo esses endereçados na primeira coluna com um código numérico para futuras referências aos objetivos enunciados. A empresa detalhou tais requisitos para atender a sua própria operação. O Apêndice B apresenta uma modelagem de casos de uso envolvendo os atores.

Tabela 1 - Atores e seus objetivos

ID	Ator	Objetivo
OBJ-1	Administrador	Cadastrar lojistas e empregadores
OBJ-2		Gerenciar dados de lojistas, empregadores e clientes
OBJ-3		Gerenciar e emitir relatórios de pagamento ao lojista
OBJ-4		Gerenciar cobranças ao empregador
OBJ-5		Visualizar últimas compras e emitir relatórios
OBJ-6	Lojista	Efetuar vendas para clientes
OBJ-7		Visualizar últimas vendas e emitir relatórios
OBJ-8		Gerenciar dados cadastrais
OBJ-9	Cliente	Efetuar compras de lojistas
OBJ-10		Visualizar últimas compras
OBJ-11		Gerenciar dados cadastrais
OBJ-12	Empregador	Cadastrar Clientes
OBJ-13		Gestão de limite e saldo dos clientes afiliados
OBJ-14		Gerenciar dados cadastrais dos clientes afiliados
OBJ-15		Visualizar últimas compras e emitir relatórios
OBJ-16	Cliente e Lojista	Emitir comprovante após transação confirmada

Fonte: Autor (2023).

Os clientes podem efetuar transações na rede, realizando compras e pagamentos utilizando o saldo disponível em sua conta, enquanto os lojistas são responsáveis por vender na rede, aceitando pagamentos dos clientes e garantindo o fluxo de transações. Os empregadores têm a possibilidade de cadastrar seus colaboradores como clientes para consumir na rede, podendo optar por um plano pré-pago ou pós-pago. Os colaboradores, por sua vez, podem utilizar os saldos disponíveis em suas contas para realizar transações na rede. Todo o fluxo transacional além de gestão de dados cadastrais fica disponível para o administrador do sistema.

Essas interações entre os diferentes atores permitem que o sistema atenda as necessidades transacionais, tendo toda a estrutura viabilizada através de aplicativos móveis e páginas web.

4.2 REQUISITOS FUNCIONAIS

Os Requisitos Funcionais (RFs) descrevem as funcionalidades específicas de um software, considerando as tarefas e serviços que ele oferece aos seus usuários. Esses requisitos representam as exigências mínimas necessárias para que o software seja funcional de acordo com o seu planejamento [33].

A Tabela 2 apresenta uma lista dos principais RFs implementados no sistema. Na primeira coluna da tabela são exibidos os códigos dos RFs, já na segunda coluna, os objetivos aos quais eles se referem (de acordo com a Tabela 1). Na coluna "Requisito", são fornecidas as informações específicas de cada RF, e na última coluna é apresentada a sua descrição correspondente.

Tabela 2 - Requisitos Funcionais

Código	Objetivo	Requisito	Descrição
RF-1	OBJ-6 e OBJ-9	Efetuar Transações	Deve ser possível efetuar transações entre cliente e lojista.

RF-2	OBJ-3, OBJ-4, OBJ-6, OBJ-9	Faturar transações	Deve ser possível efetuar manutenção de saldo do cliente e registrar a cobrança ao empregador e o pagamento ao lojista.
RF-3	OBJ-3 e OBJ-7	Pagar Lojistas	Deve ser possível visualizar e emitir relatório do montante de pagamento ao lojista.
RF-4	OBJ-4 e OBJ-15	Receber Pagamento de Empregadores	Deve ser possível visualizar e emitir relatórios para a cobrança de um empregador, afetando o limite disponível de seus colaboradores.
RF-5	OBJ-12 e OBJ-14	Cadastrar clientes afiliados	Deve ser possível cadastrar, definir e redefinir o limite de um cliente afiliado .
RF-6	OBJ-13	Controlar estado de cliente	Deve ser possível bloquear clientes, para caso de desligamento de funcionário.
RF-7	OBJ-7, OBJ-15 e OBJ-5	Emissão de relatórios transacionais e financeiros	Deve ser possível gerar relatórios de extratos de últimas compras e comprovantes fiscais.
RF-8	OBJ-5	Visualizar histórico de transações	Deve ser possível para o Administrador visualizar todas as transações efetuadas.
RF-9	OBJ-10	Visualizar as compras efetuadas	Deve ser possível para o Cliente visualizar as transações efetuadas por ele.

RF-10	OBJ-7	Visualizar as vendas efetuadas	Deve ser possível para o Lojista visualizar as transações efetuadas por ele.
RF-11	OBJ-15	Visualizar histórico de compras dos clientes afiliados	Deve ser possível para o Empregador visualizar as transações efetuadas por todos os seus clientes cadastrados.
RF-12	OBJ-2	Administradores devem poder alterar dados cadastrais e transacionais.	Deve ser possível para os Administradores gerenciarem dados cadastrais e transacionais de Lojistas, Empregadores e Clientes.
RF-13	OBJ-14	Empregadores devem poder alterar dados cadastrais dos clientes afiliados	Deve ser possível os Empregadores gerenciarem os dados cadastrais de seus clientes cadastrados.
RF-14	OBJ-8	Lojistas devem poder gerenciar os próprios dados cadastrais	Deve ser possível que os Lojistas gerenciem seus próprios dados cadastrais.
RF-15	OBJ-11	Gerenciamento de dados	Deve ser possível que os Clientes gerenciem seus próprios dados cadastrais.

Fonte: Autor (2023).

4.3 REQUISITOS NÃO FUNCIONAIS

A complexidade de um software é influenciada tanto pela sua funcionalidade quanto pelos requisitos gerais relacionados ao desenvolvimento do *software*, tais como custo, desempenho, confiabilidade, manutenção, portabilidade e custos operacionais. Esses requisitos são conhecidos como Requisitos Não Funcionais (RNFs) [34].

Enquanto os RFs descrevem o que deve ser feito, os RNFs indicam como deve ser feito, ou seja, qual a abordagem que o software adotará para concretizar o planejado. Por exemplo, eles determinam para qual(is) sistema(s) operacional(is) o serviço será desenvolvido, qual tecnologia de armazenamento será utilizada, como será estabelecida a conexão entre as diferentes partes da plataforma, entre outros aspectos relevantes.

A Tabela 3 apresenta os principais requisitos não funcionais do sistema. Na primeira coluna temos o código de identificação de cada requisito, na segunda coluna sua categoria correspondente e na terceira coluna uma descrição detalhada do requisito em questão.

Tabela 3 - Requisitos não Funcionais

Código	Categoria	Descrição
RNF-1	Desempenho	O saldo dos clientes, uma vez calculados, devem ser armazenados na memória cache. Isso elimina a necessidade de um novo cálculo quando acontecer uma nova consulta.
RNF-2		O fluxo de vendas deve ser isolado, uma operação pode ocorrer independentemente de outras, sem precisar esperar que a anterior termine.
RNF-3		As informações não podem levar mais que 2 segundos para serem carregadas na aplicação *web*

		e no aplicativo móvel.
RNF-4	Portabilidade	A aplicação poderá ser acessada de qualquer plataforma *web*, além de aplicativos disponibilizados para Android e iOS.
RNF-5		As aplicações mobile devem ser construídas em formatos de páginas *web*, portáteis para aplicações mobile *'webview'*.
RNF-6	Segurança	Todo o tráfego de rede deve ser feito utilizando HTTPS, e não HTTP.
RNF-7		A autenticação na API será feita por meio de Json *Web Tokens*.
RNF-8		Todos os dados serão armazenados no banco de dados ou no servidor, nenhum dado será armazenado localmente.
RNF-9	Usabilidade	Ao fazer login o usuário deve manter sessão com utilização de *cookies*.
RNF-10		Deve haver compatibilidade com o sistema legado para as interfaces de usuário e processamento de transações.
RNF-11	Disponibilidade	Para a utilização da aplicação, o dispositivo deverá estar conectado na *internet*.

Fonte: Autor (2023).

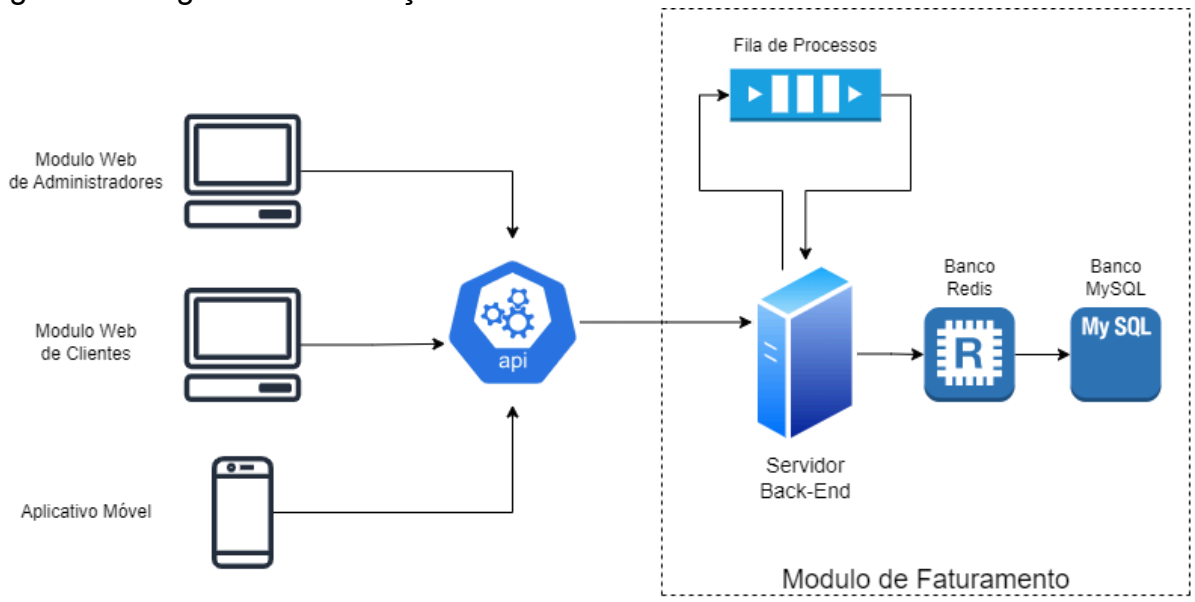
4.4 ARQUITETURA

Para viabilizar um sistema computacional capaz de contemplar todos os requisitos definidos, nós desenvolvemos diferentes aplicações, divididas em quatro componentes de software com responsabilidades distintas, que são:

1. Módulo de faturamento
2. Módulo de aplicativo móvel
3. Módulo de painel *web* para clientes
4. Módulo de painel *web* para administradores

A Figura 5 detalha como tais módulos se comunicam, descrevendo uma arquitetura de alto nível para o sistema.

Figura 5 - Diagrama de interações entre módulos



Fonte: Autor (2023).

4.5 MÓDULO DE FATURAMENTO

O módulo de faturamento é estruturado predominantemente em torno de uma API Rest desenvolvida com o uso do FastAPI. As rotas dessa API são projetadas para prover acesso às regras de negócio fundamentais relacionadas com os requisitos descritos na Tabela 2.

Esse é o componente central de *back-end*, que tem importância crucial no processamento de transações, realização de cálculos de saldo, validação de operações efetuadas pelos outros módulos e autenticação dos usuários. Além dos fluxos mencionados anteriormente, tal módulo também lida com o consumo de dados para telas e relatórios, o que envolve operações ao banco MySQL.

4.5.1 Operações de Banco de Dados

O principal banco de dados da aplicação é um banco MySQL, sendo este um dos componentes de *software* que vieram da provedora terceirizada. A definição de seu modelo está fora do escopo deste projeto.

Existem diversas entidades e relacionamentos definidos no banco em questão. A ferramenta SQLAlchemy foi utilizada para efetuar a integração e definição de classes de alto nível que promovem a interação com cada entidade necessária para a operação do motor de faturamento. O uso do *framework* fica encapsulado dentro de um DAO, que consiste em uma classe Python fazendo o gerenciamento das operações de criação, edição e atualização para uma determinada tabela no banco. Um exemplo da implementação de um DAO pode ser observado no treco de código 7.

Código 7 - Exemplo de interface de um DAO

```
1.    from sqlalchemy_db import Session
2.    from dto import ClienteDTO
3.    from db import Cliente
4.
5.    class ClienteDAO:
6.       def __init__(self):
7.          self.db = Session()
8.
9.       def busca_cliente(self, codigo) -> ClienteDTO:
10.         d = self.db.query(Cliente) \
11.            .filter(Cliente.codigo == codigo) \
12.            .one()
13.
14.         return ClienteDTO(d)
15.
16.      def cria_cliente(self, cliente: ClienteDTO):
17.         ...
18.
19.      def atualiza_cliente(self, cliente: ClienteDTO):
20.         cliente.commit()
21.         self.db.commit()
```

Fonte: Autor (2023).

O DAO irá utilizar uma instância de DTO que pode ser implementada em conjunto com o *framework* SQLAlchemy, de modo a oferecer os campos presentes no resultado da introspecção de tabela, ou seja, todos os campos originais definidos na tabela. Um exemplo simplificado da implementação original está no trecho de Código 8. Supõe-se que a entidade de dados é representada apenas por uma tabela, sendo possível tornar isso mais robusto para aceitar modelos de dados com *Joins*.

Código 8 - Exemplo de interface de um DTO

```
1.    import sqlalchemy
2.
3.    class ClienteDTO:
4.       nome: str
5.       codigo: int
6.
7.       def __init__(self, query_result):
8.          self._table = query_result.__class__
9.          self._mapper = sqlalchemy.inspect(self._table)
10.            for k in self._mapper.attrs:
11.               v = getattr(query_result, k)
12.               attr_name = str(k).split('.')[1]
13.               setattr(self, attr_name, v)
14.
15.         def commit(self):
16.            for k in self._mapper.attrs:
17.               attr_name = str(k).split('.')[1]
18.               dto_value = getattr(self, attr_name)
19.               setattr(
20.                  self._original_data,
21.                  attr_name,
22.                  dto_value
23.               )
```

Fonte: Autor (2023).

Nós utilizamos uma abordagem genérica para a implementação dos DTOs, como pode ser observado no construtor do Código 8, utilizamos um método do SQLAlchemy(*inspect*) para efetuar a população dos atributos do objeto com base nos atributos da tabela em tempo de execução. De modo opcional, podemos definir no DTO os campos que se deseja trabalhar, como podemos ver nas linhas 4 e 5 do trecho de código 8. Isso melhora a interação com a tipagem no Python, além de possibilitar que IDEs analisem o código sugerindo o atributo definido, contudo ainda

é possível acessar o valor dos atributos não definidos no DTO e também utiliza-los em queries.

Além disso, o encapsulamento da responsabilidade de atualização da entidade de dados do SQLAlchemy foi feito dentro do DTO. Um exemplo disso está no Código 8 na linha 15. Essa abordagem também é dinâmica e utiliza os atributos da tabela no MySQL capturados através do *inspect*, atualizando as alterações efetuadas no objeto python diretamente no objeto do SQLAlchemy, possibilitando a persistência dos dados alterados no DTO de forma simplificada.

O *commit* é utilizado no método *atualiza_cliente*, definido na linha 19 do trecho de Código 7, juntamente com o *commit* da sessão do SQLAlchemy.

4.5.2 Fila de Tarefas

A Fila de Tarefas gera vantagens quando utilizada em fluxos de trabalho que exige um processamento mais intensivo no servidor. Nós integramos a fila em operações que demandam múltiplas interações com o banco de dados como nas liberações de saldo, ou que exigem processamento computacional elevado,como no faturamento de transações.

Utilizamos o RabbitMQ para implementação da fila, incorporando ao sistema uma fila de tarefas HTTP simples através de chamadas efetuadas no método de callback do *consumidor,* com um modelo de mensagens contendo um JSON formatado como *byte string*. O esquema estabelecido para as mensagens exige que a mensagem possua duas chaves essenciais: “rota” e “*payload*”, que irão estabelecer uma interface de chamadas HTTP.

Quando uma mensagem é recebida pelo consumidor, a função de *callback* é executada. Ela é definida no trecho de Código 10 na linha 15, fazendo uma chamada HTTP para o endereço especificado na “rota” e transmitindo o “*payload*” da mensagem no corpo da solicitação. O consumidor não irá enviar outra mensagem enquanto a primeira não for totalmente processada, além de enfileirar a mensagem novamente em caso de falha.

O mecanismo em questão é demonstrado pelos excertos dos Códigos 9 e 10. Estes servem para ilustrar como se dá a implementação do processo de geração e recepção de mensagens. Para simplificar a compreensão, os exemplos omitirão a autenticação do serviço de *back-end* e farão uso exclusivo do método POST para realizar as solicitações.

A estratégia de processamento de transações com base na estratégia apresentada acima é vital para garantir que o sistema atenda o RNF-2.

Código 9 - Definição de uma classe produtora com RabbitMQ

```
1.  import json
2.  import pika
3.
4.   class RabbitMQ:
5.      def __init__(self):
6.         credentials = pika.PlainCredentials(...)
7.         self.connection = pika.BlockingConnection(
8.            pika.ConnectionParameters(
9.               credentials=credentials
10.        ))
11.     self.channel = self.connection.channel()
12.     self.channel.queue_declare('RABBITMQ_QUEUE')
13.
14.  def publish(self, rota, payload):
15.     message = json.dumps(
16.        {
17.           'rota': rota,
18.           'payload': payload
19.        }
20.     )
21.
22.     properties=pika.BasicProperties(delivery_mode=2)
23.
24.     self.channel.basic_publish(
25.        exchange='',
26.        routing_key='RABBITMQ_QUEUE',
27.        body=message.encode('utf-8'),
28.        properties=properties
29.     )
30.
31.  def close(self):
32.     self.connection.close()
```

Fonte: Autor (2023).

Código 10 - Definição de uma instância de consumidor do RabbitMQ

```
1.    import json
2.    import requests
3.    import pika
4.
5.
6.    credentials = pika.PlainCredentials(...)
7.    parameters = pika.ConnectionParameters(
8.       credentials=credentials
9.    )
10.
11.   connection = pika.SelectConnection(...)
12.   channel = connection.channel()
13.   channel.queue_declare(queue='RABBITMQ_QUEUE')
14.
15.   def callback_function(ch, method, properties, body):
16.      try:
17.         body = json.loads(body)
18.         rota = body['rota']
19.         body = dict(body['payload'])
20.         requests.post(url+rota, json=body)
21.         ch.basic_ack(delivery_tag=method.delivery_tag)
22.      except Exception as e:
23.         ch.basic_reject( ... )
24.
25.   channel.basic_consume(
26.      queue='RABBITMQ_QUEUE',
27.      auto_ack=False,
28.      on_message_callback=callback_function
29.   )
30.   channel.basic_qos(prefetch_count=1)
31.   channel.start_consuming()
```

Fonte: Autor (2023).

4.5.3 Cache write-through para totalizadores

Mesmo garantindo o processamento de muitas requisições com a fila de tarefas, ainda existem rotinas dentro do sistema que precisam ser entregues ao cliente de forma ágil e síncrona. Um exemplo é o cálculo de saldos ou limites de clientes, em que registros de múltiplas tabelas devem ser operados matematicamente para que possam ser entregues ao cliente ou utilizados em processos de validação.

Para lidar com essa problemática o uso da estratégia de cache *write through* com o Redis foi adotado. Com a utilização desse mecanismo, garantimos eficiência e agilidade para os fluxos mencionados. A escolha visou a redução do tempo gasto em operações recorrentes em que, caso os parâmetros de uma operação não se alterem, o seu resultado permanecerá inalterado. Isso permite que em acessos subsequentes, os dados possam ser recuperados sem a necessidade de nova computação, resultando em uma significativa economia de tempo e recursos computacionais, tal estratégia é importante para garantir o RNF-3.

O trecho de Código 11 apresenta um modelo simplificado da função de obtenção de saldo, excluindo, para facilitar a compreensão, a interação com o banco de dados e o cálculo efetivo do saldo. É importante reconhecer que a operação omitida na linha 10 compreende a coleta integral dos débitos e créditos associados a um cliente específico, com os débitos sendo subtraídos dos créditos no processo.

Código 11 - Consulta de saldo com cache em memória

```
1.    from my_redis import RedisClient
2.
3.    def consulta_saldo(self, codigo):
4.       redis_client = RedisClient()
5.       saldo_em_memoria = redis_client.hget(codigo,
   'saldo')
6.
7.       # Checa se já existe o saldo em memória
8.       if saldo_em_memoria is None:
9.          # Efetua cálculo do saldo com dados do banco
10.         saldo_calculado = …
11.
12.         # Persiste o resultado do cálculo no redis
13.         redis_client.hset(codigo, 'saldo', saldo)
14.         redis_client.expire(codigo, 1800)
15.         saldo_atual = redis_client.hget(codigo, 'saldo')
16.         return saldo_calculado
17.      else: # Já existe saldo calculado em memória
18.         return saldo_em_memoria
```

Fonte: Autor (2023).

4.6 MÓDULO WEB PARA ADMINISTRADORES

O módulo *Web* é um dos componentes visuais do sistema, seguindo o padrão de arquitetura MTV, adotado graças à sua característica de isolamento da responsabilidade da camada *controller* do padrão MVC para o *framework web* FastApi, nos permitindo focar no desenvolvimento da solução sem tantas burocracias técnicas.

Nós utilizamos o *framework* FastAPI, o mecanismo de template Jinja, HTML, CSS, além do *framework* HTMX para efetuar chamadas AJAX sem a necessidade de arquivos JS na dinâmica de construção de páginas.

Esse módulo não contém lógica de *back-end* ou acesso a banco de dados. Tais funcionalidades são conduzidas por meio de requisições ao módulo de faturamento, que disponibiliza os dados através de uma interface de API Rest. O módulo tem a capacidade de construir telas, tabelas e estruturar arquivos CSV para a exportação de dados, gerenciando o login de administradores do sistema e gerenciamento de sessão de usuário.

A fim de demonstrar o processo de construção de telas utilizando AJAX, detalharemos o desenho de uma tabela simples, com parâmetros obtidos do usuário através de entradas em um formulário HTML. Para melhor entendimento, algumas estruturas de HTML serão omitidas.

Código 12 - HTML com chamada via HTMX

```
1.    <form hx-get="/api/get-lojas" hx-target="#target">
2.       <input id="cidade" name="cidade" required>
3.       <button type="submit">Enviar</button>
4.    </form>
5.    <div id="target"></div>
```

Fonte: Autor (2023).

O template realiza uma chamada HTTP não bloqueante para */api/get-lojas*. Essa rota deve retornar código HTML, que por sua vez será inserido no elemento *#target*. Um exemplo simplificado da dinâmica de construção da resposta utilizando a arquitetura MTV pode ser visualizado no Código 13.

Código 13 - Exemplo de resposta com MTV

```
1.    # View - Python
2.    @api.get("/api/get-lojas")
3.    @jinja.template('lojas.j2')
4.    def view_ajax_lojas(cidade: str):
5.       m = Model()
6.       m.lojas = v.filtra_lojas_por_cidade(cidade)
7.       return m.build()
8.
9.    # Model - Python
10.   class Model
11.      def filtra_lojas_por_cidade(cidade: str) ->
   List[dict]:
12.         return self.backend.get_lojistas(cidade)
13.
14.   # Template - Jinja2
15.   <table>
16.      <!-- Table Header -->
17.      {% for loja in lojas %}
18.         <tr>
19.            <td>
20.               <a href="/lojista/{{ loja.CODIGO }}">
21.                  {{ loja.NOME }}
22.               </a>
23.            </td>
24.         </tr>
25.      {% endfor %}
26.   </table>
```

Fonte: Autor (2023).

No momento de clique no botão de *submit* do formulário(linha 3 do Código 12), a chamada HTTP atinge a *view*(linha 1 do Código 13), que por sua vez receberá como parâmetro o conteúdo da entrada do formulário. A *view* irá utilizar o modelo(linha 9 do Código 13) responsável pelos dados daquela entidade para

construir o HTML de resposta. O *template*(linha 14 do Código 13) Jinja utiliza os dados do modelo para gerar código html de forma dinâmica. Para evidenciar essa interação podemos observar a linha 6 do trecho de Código 13, em que vemos o atributo lojas definido pela *view*. Isso viabiliza que o *template* utilize esses dados durante a construção da resposta. Nas linhas 17, 20 e 21 também do Código 13 podemos observar a utilização desses dados para a construção de uma tabela dentro do HTML.

4.7 MÓDULO DE APLICATIVO MÓVEL

Para o módulo de Aplicativo Móvel foram adotadas as mesmas ferramentas e princípios estruturais do módulo *Web*. A principal diferença é que a presente aplicação tem o objetivo de ser distribuída e utilizada por aplicativos móveis. Dessa forma, além da camada *web*, em que as telas são renderizadas utilizando o padrão MTV para construção de páginas HTML e disponibilizadas através de um domínio na rede, devemos também criar um projeto que incorpore e renderize tais páginas em um dispositivo móvel.

Para este projeto, o Flutter foi empregado primariamente para incorporar uma *WebView*, que é um componente que permite visualizar conteúdo web dentro de um aplicativo nativo. A *WebView* carrega as páginas HTML desenvolvidas seguindo o padrão MTV, que são otimizadas para dispositivos móveis. Isso é feito através do uso do plugin *webview_flutter*, que oferece uma maneira conveniente e eficiente de carregar conteúdo *web*.

O código 14 ilustra a estrutura principal do módulo de aplicativo móvel, demonstrando a inicialização da aplicação com o Flutter, a gestão de estados e permissões e a integração da *WebView*:

Código 14 - Widget de *Webview* com Flutter

```
1.    class MyWebView extends StatelessWidget {
2.       @override
3.       Widget build(BuildContext context) {
4.          return Scaffold(
5.             appBar: AppBar(title: Text('My WebView')),
6.             body: WebView(
7.                initialUrl: 'https://mydomain.com',
8.                javascriptMode: JavascriptMode.unrestricted,
9.             ),
10.         );
11.      }
12.   }
```

Fonte: Autor (2023).

Para garantir uma experiência de usuário robusta e segura no módulo de Aplicativo Móvel, é essencial uma gestão de sessão e permissões. Esses mecanismos são responsáveis por manter a aplicação responsiva e interativa, assegurando que o conteúdo seja apresentado e que os recursos sejam acessados de maneira segura e com o consentimento do usuário. No exemplo de código fornecido acima, optou-se por uma representação sintetizada, focando na integração básica da *WebView* com o Flutter. Para manter o foco na estrutura principal do módulo do aplicativo móvel, o excerto de código 14 foi simplificado, omitindo tais detalhes. Isso foi feito para evitar a complexidade desnecessária que poderia dificultar o entendimento do funcionamento básico da *WebView* no Flutter. No entanto, é importante notar que, em uma aplicação real, o código seria mais complexo e incluiria a implementação detalhada destes aspectos, além de controle para notificações e permissões.

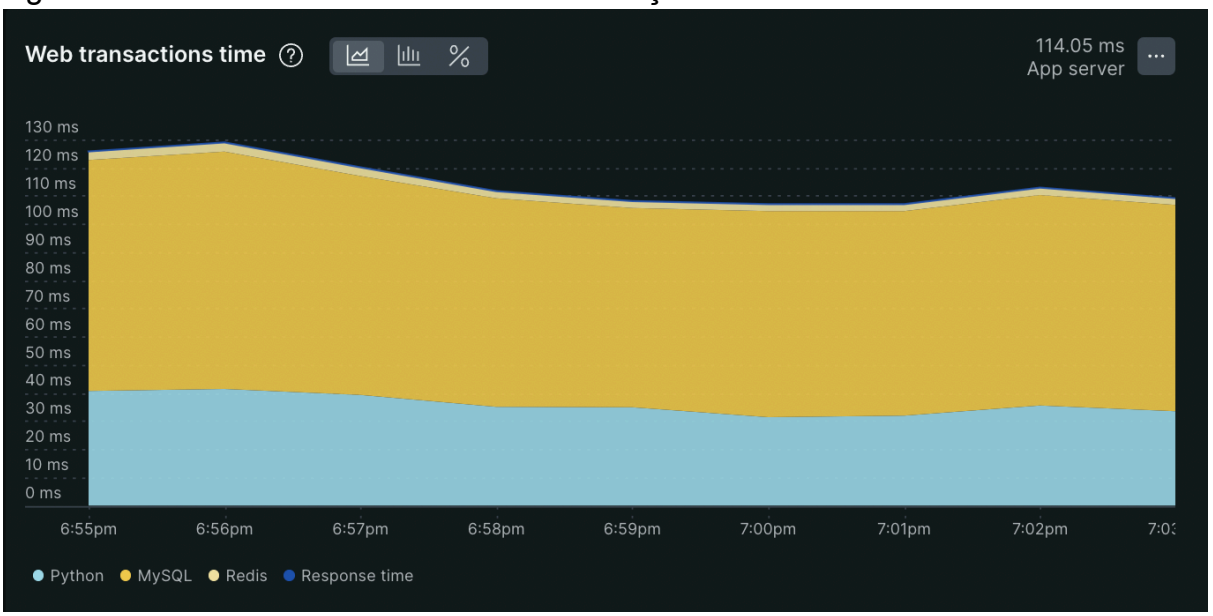
Com essa abordagem, a experiência do usuário é equivalente a de um aplicativo móvel nativo, com a flexibilidade de atualizações rápidas e diretas do conteúdo web — o que é uma vantagem significativa para o gerenciamento do aplicativo.

5 RESULTADOS

5.1 MÉTRICAS DA FILA DE TAREFAS

Utilizando a ferramenta de coleta de métricas New Relic [35], o sistema foi medido em funcionamento e seu comportamento analisado em diferentes cenários, no intuito de entender melhor as vantagens do uso da fila de tarefas. O gráfico da Figura 6 demonstra o servidor em um cenário hipotético, recebendo 30 transações por minuto de forma constante.

Figura 6 - Gráfico com fluxo estável de transações



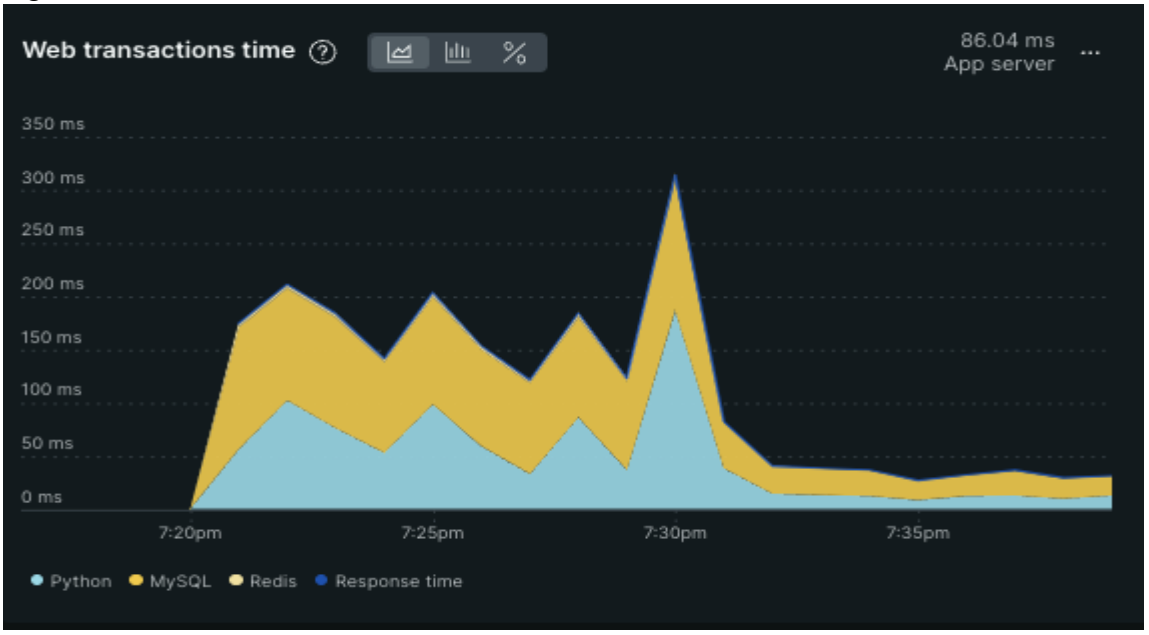
Fonte: Autor (2023).

Ao visualizar a Figura 6, fica perceptível que a operação é custosa no banco de dados, como demonstrada pela porção amarela do gráfico. Além disso, também consome uma breve porção de tempo no redis. Isso se dá pelo fluxo de verificação de saldo antes da validação da venda e escrita de um novo saldo após a conclusão da mesma.

Para os próximos gráficos, foi considerado um cenário mais complexo: 1000 transações são registradas simultaneamente, mantendo um fluxo constante de 15

transações por minuto. No gráfico a seguir (Figura 7) podemos observar que o tempo de resposta do processamento aumenta (representados pelos picos do gráfico na Figura 7), mas o servidor é capaz de compensar o fluxo inicial intenso, normalizando o excesso ao longo do tempo.

Figura 7 - Gráfico com fluxo excessivo utilizando a fila de tarefas



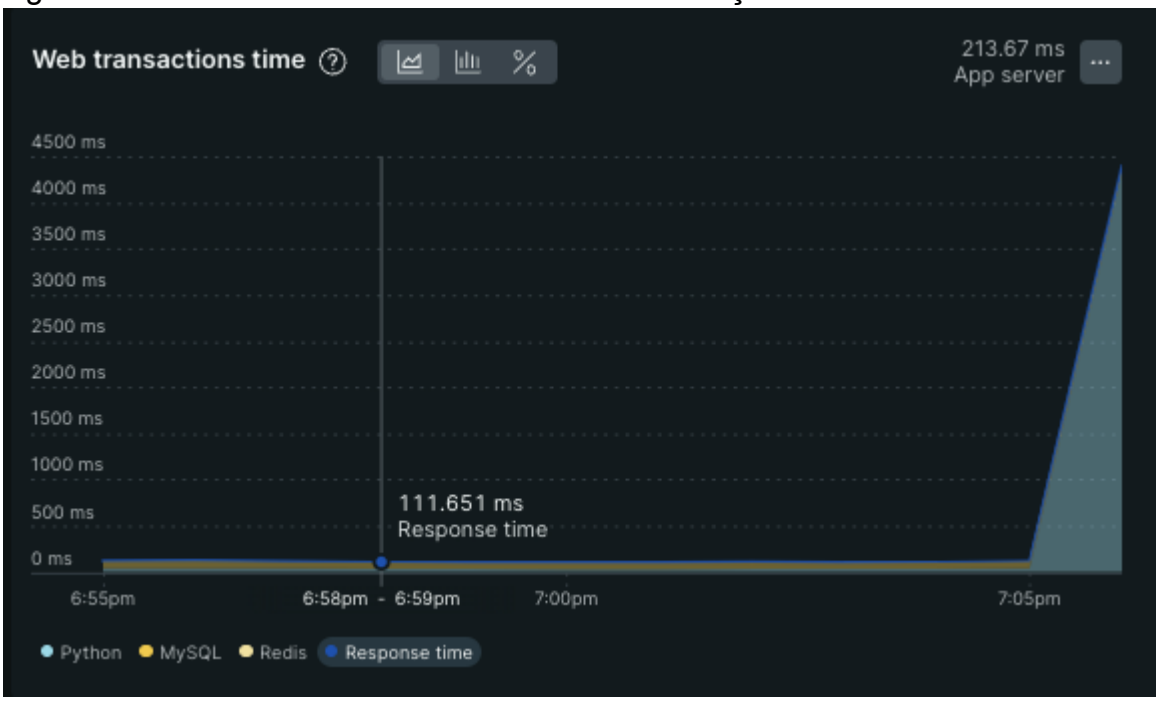
Fonte: Autor (2023).

Inicialmente, o tempo de resposta do processamento aumenta significativamente, criando picos no gráfico. Isso ocorre porque a taxa de chegada das transações (λ) é muito superior à capacidade de processamento do servidor (μ), resultando em um aumento no tempo de espera na fila. Em termos de teoria das filas, quando λ excede μ, o tempo médio de espera aumenta, causando um aumento correspondente no tempo de resposta total[35]. Esses picos refletem a incapacidade inicial do sistema de processar rapidamente a alta carga de trabalho. Contudo, à medida que o servidor processa continuamente as transações pendentes, a fila começa a diminuir, e o tempo de resposta gradualmente retorna a níveis normais . Esse comportamento demonstra a capacidade do servidor de compensar o fluxo inicial intenso e estabilizar o sistema, processando todas as transações sem negações ou erros.

Considerando um cenário similar, porém sem a utilização da fila de processos, rapidamente entramos em escassez de recursos, levando a uma falha na operação pelo não aceite de transações ou falha no processamento. O gráfico abaixo (Figura 8) mostra o fluxo de 30 transações por minuto, e subitamente 1000 transações simultâneas, levando rapidamente a *timeouts* e falhas no banco de dados, com um *crash* no sistema.

É importante enfatizar que tal cenário é tendencioso, sendo este exemplo capaz de evidenciar as vantagens trazidas pela implementação da estratégia de processamento de transações através da filas de tarefas.

Figura 8 - Gráfico com fluxo excessivo sem a utilização da fila de tarefas

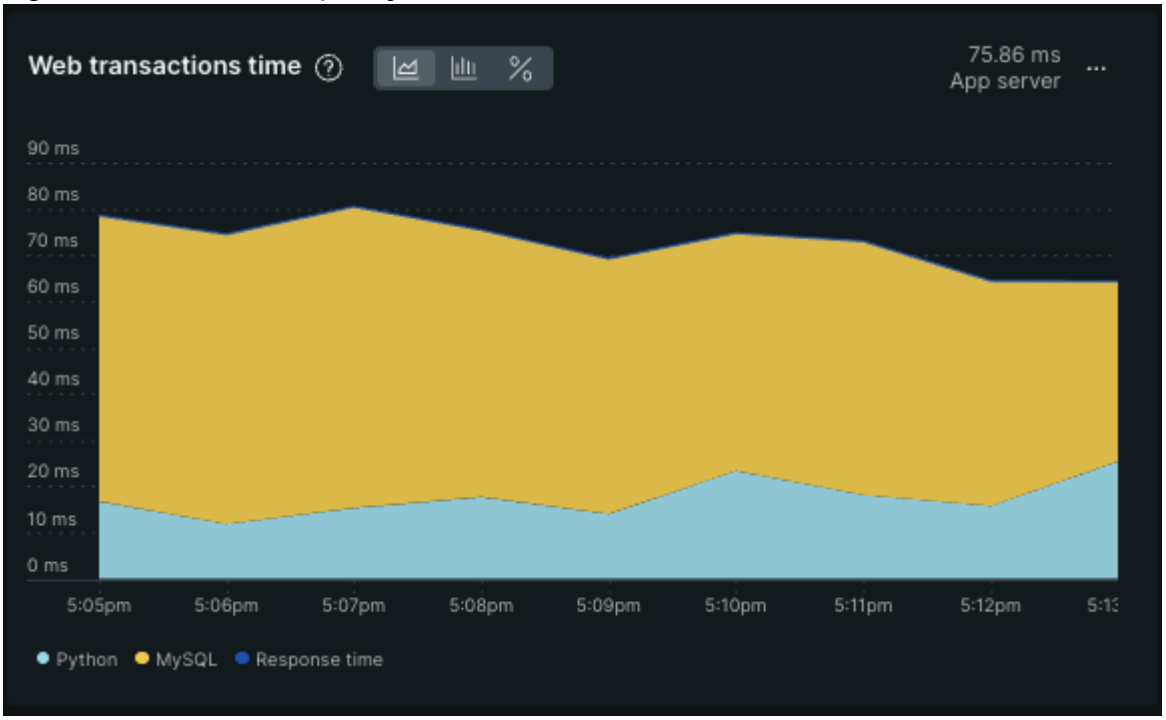


Fonte: Autor (2023).

5.2 MÉTRICAS DO CACHE WRITE-THROUGH

Utilizando ainda o New Relic [36], observamos o comportamento de um grupo de 10 clientes que efetuam a captura do saldo em uma frequência de 3 chamadas por minuto. Esse cenário acontece sem a utilização da estratégia de cache:

Figura 9 - Gráfico de operações de cálculo sem uso de cache

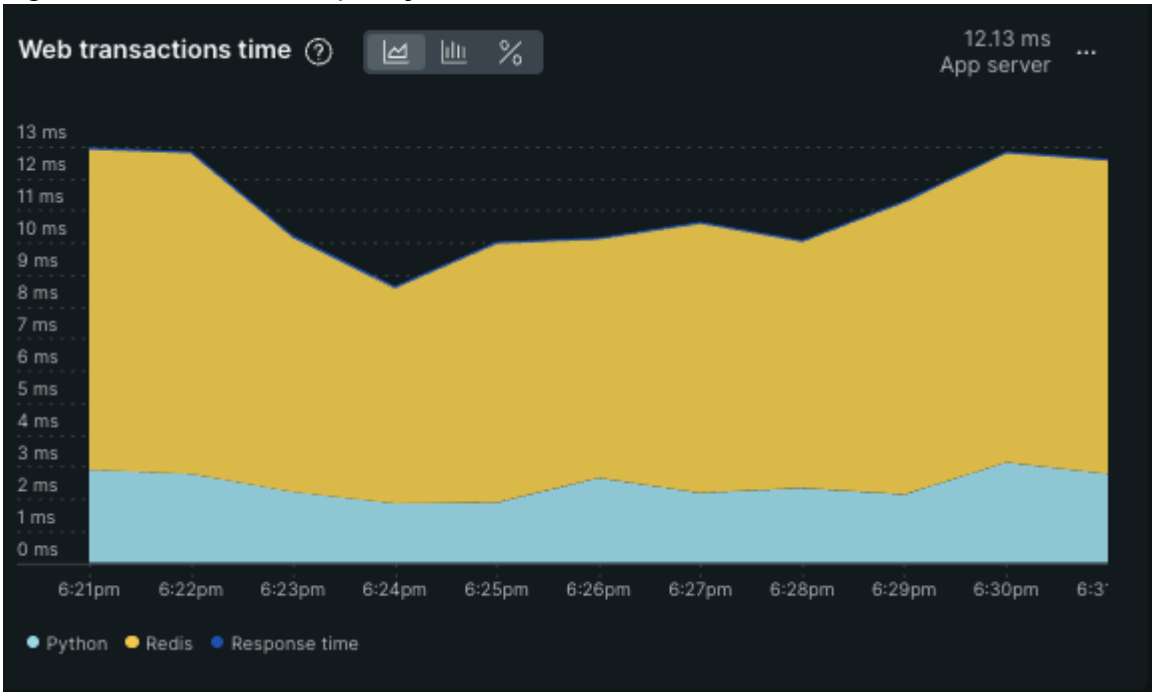


Fonte: Autor (2023).

É possível constatar que a operação possui um alto custo de banco de dados, de forma que a obtenção de uma resposta envolve múltiplas consultas na base, além da realização de operações matemáticas com os resultados.

Vamos observar agora o mesmo cenário com a utilização do cache *write-through*, dessa vez supondo o cálculo prévio dos totalizadores de todos os clientes envolvidos:

Figura 10 - Gráfico de operações de cálculo com *hit* em todas as chaves

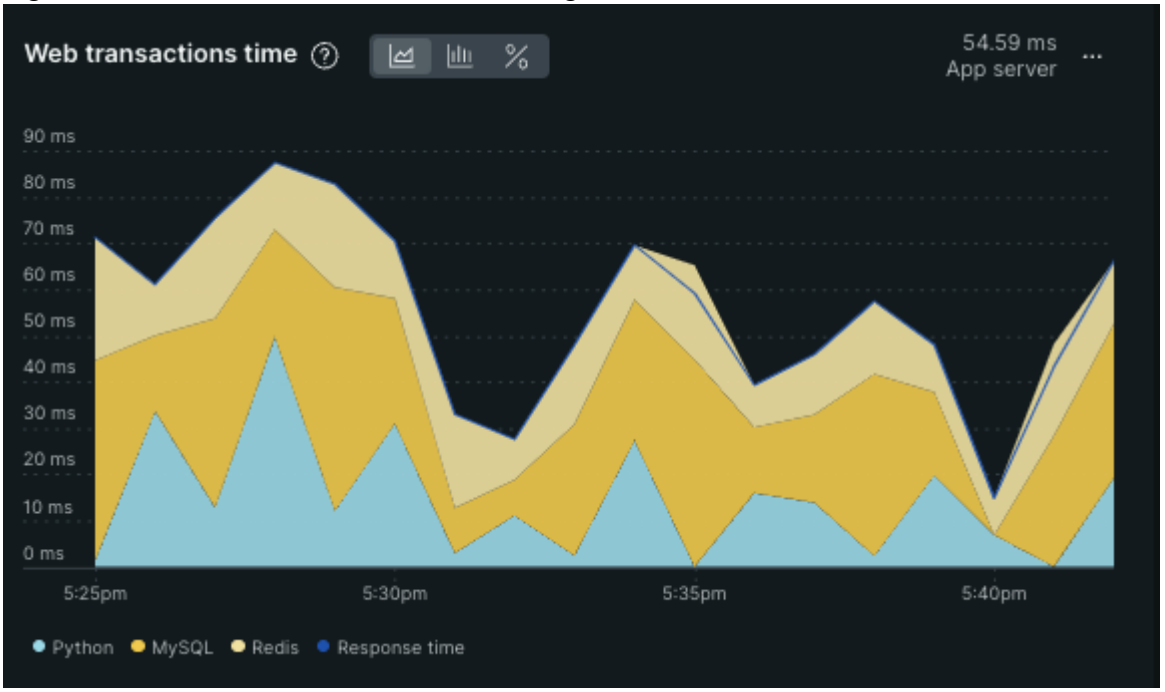


Fonte: Autor (2023).

A partir da utilização do redis é possível ver que as chamadas tiveram uma grande redução no tempo de resposta, retornando com média de 12 ms. É importante entender que, com a utilização da estratégia de cache, devemos considerar que para uma determinada operação existem dois casos: o totalizador salvo em cache ou não. No cenário descrito na Figura 10 todos os totalizadores estavam salvos em cache. Um cenário mais realista é apresentado no gráfico da Figura 11, no qual é possível visualizar um fluxo de consumo de saldo de múltiplos clientes que possuem a chave salva no banco Redis ou não.

Isso gera a presença tanto de picos que incluem todo o tempo de processamento no MySQL, somando ainda o tempo de gravação do totalizador no Redis, quanto de momentos onde não existe a utilização do banco relacional, existindo, nesse caso, um tempo de resposta extremamente reduzido.

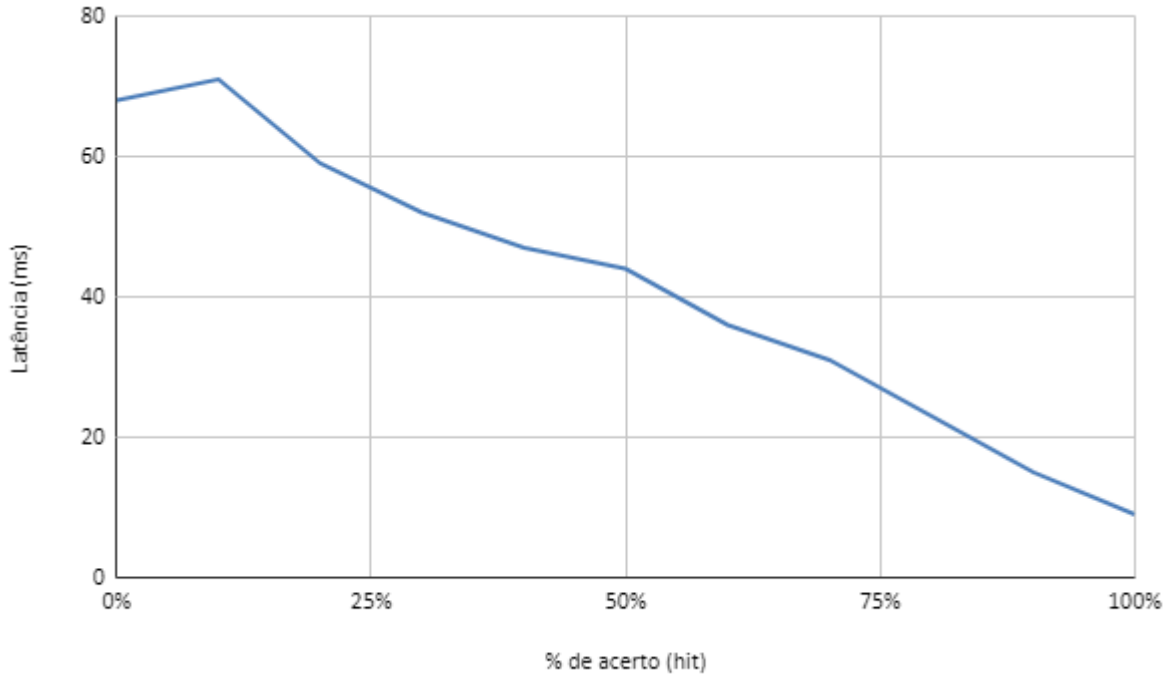
Figura 11 - Gráfico de cálculo com *hit* algumas das chaves



Fonte: Autor (2023).

Para entender melhor o comportamento do tempo de resposta, fizemos uma análise para visualizar a progressão do comportamento de um servidor respondendo com taxas de acerto variadas, de 0% à 100%, de modo a obter um gráfico que mostra a variação entre os cenários apresentados nas Figuras 9 e 10. O Apêndice A apresenta os gráficos obtidos durante a coleta dos tempos médios de resposta em cada cenário.

Figura 12 - Tempo de resposta do cache por percentual de acerto



Fonte: Autor (2024).

Com a adoção do Redis devemos considerar a escolha de um tempo de expiração para as chaves no momento da operação de gravação. A definição desse valor dentro de um fluxo de trabalho envolve amplo estudo das rotinas de uso dos clientes, ponderando entre o tempo em que determinado totalizador continuará sendo válido, o número de operações de escrita associados a esse totalizador e a frequência que essas operações acontecem no sistema.

Tal processo é complexo, pois a adoção de tempos muito altos provoca um aumento de remoções manuais da chave no Redis, enquanto um tempo muito baixo aumenta a quantidade de operações no banco principal.

Além disso, a otimização para diferentes operações possui uma escolha de tempo de expiração de chaves distinto. Um exemplo é que o totalizador referente ao valor de uma fatura fechada tende a ser válido por mais de 30 dias, enquanto o saldo de um cliente pode ser invalidado múltiplas vezes em um único dia.

5.3 FLUXOGRAMAS e INTERFACE GRÁFICA DO SISTEMA

Esta seção tem objetivo de definir os fluxos apresentados nas interfaces gráficas necessárias para compor o sistema transacional, atendendo os RFs definidos na tabela 2.
Com a intenção de tornar o entendimento das interfaces enxuto e conciso, vamos apresentar os fluxos em 4 categorias capazes de atender todos os RFs, sendo estas:

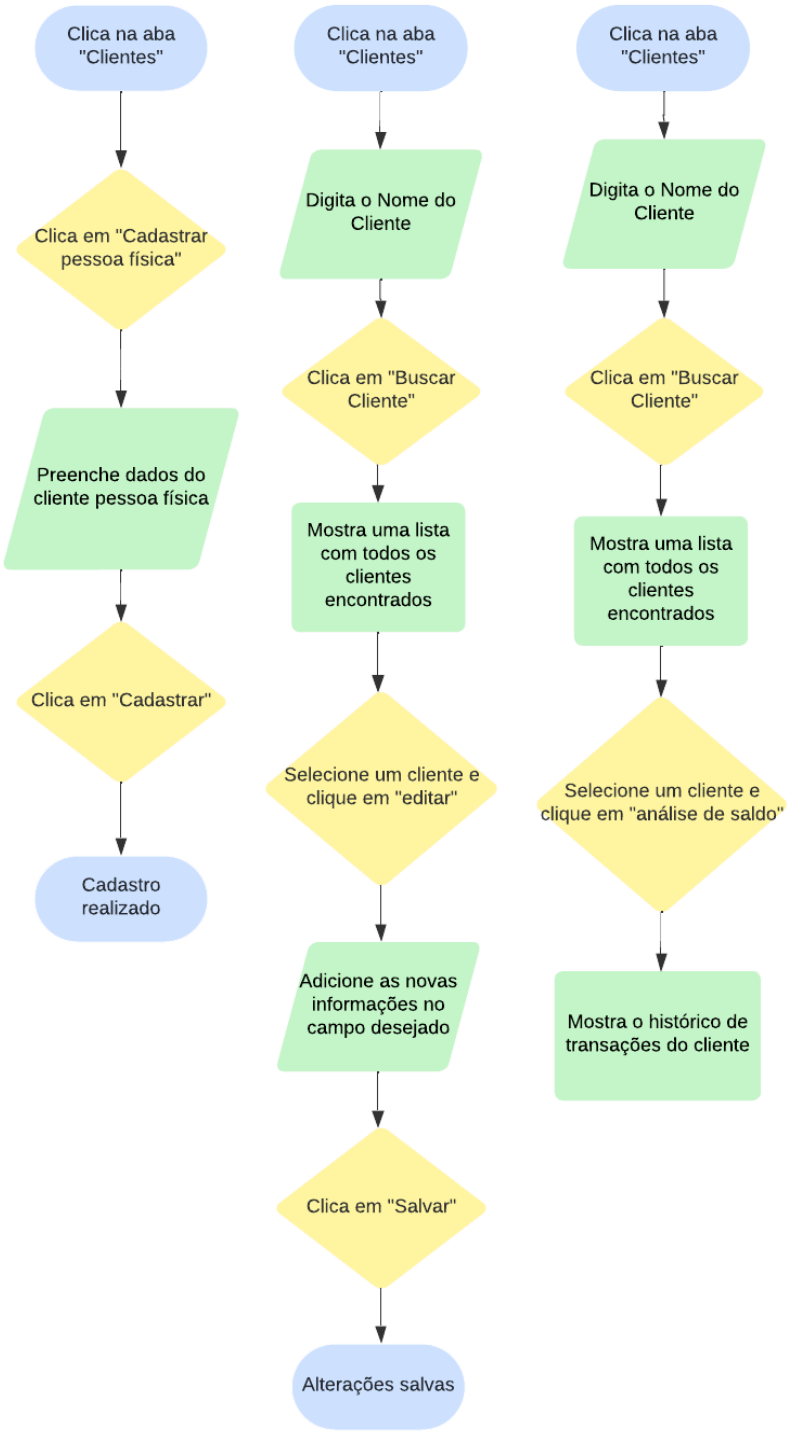
- Clientes
- Contas a Receber
- Pagamento a Lojistas
- Transações

5.3.1 Fluxos de Clientes

Para demonstrar as rotinas envolvidas no gerenciamento de clientes, vamos apresentar 3 fluxos através dos fluxogramas da Figura 13:

- Cadastrar pessoa
- Buscar e editar cliente
- Análise de saldo de cliente

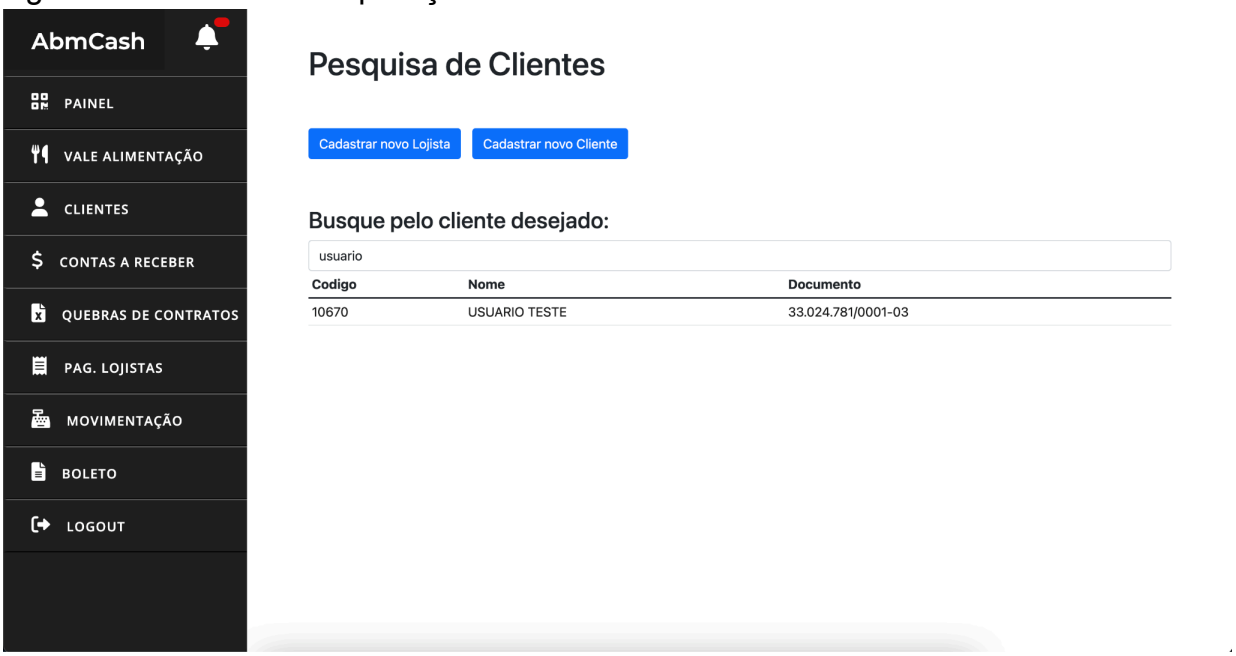
Figura 13 - Fluxograma da rotina de Clientes



Fonte: Autor (2024).

A Figura 14 representa a visão do administrador quando este se encontra no momento referenciado na Figura 12.

Figura 14 - Interface da aplicação em Clientes



Fonte: Autor (2024).

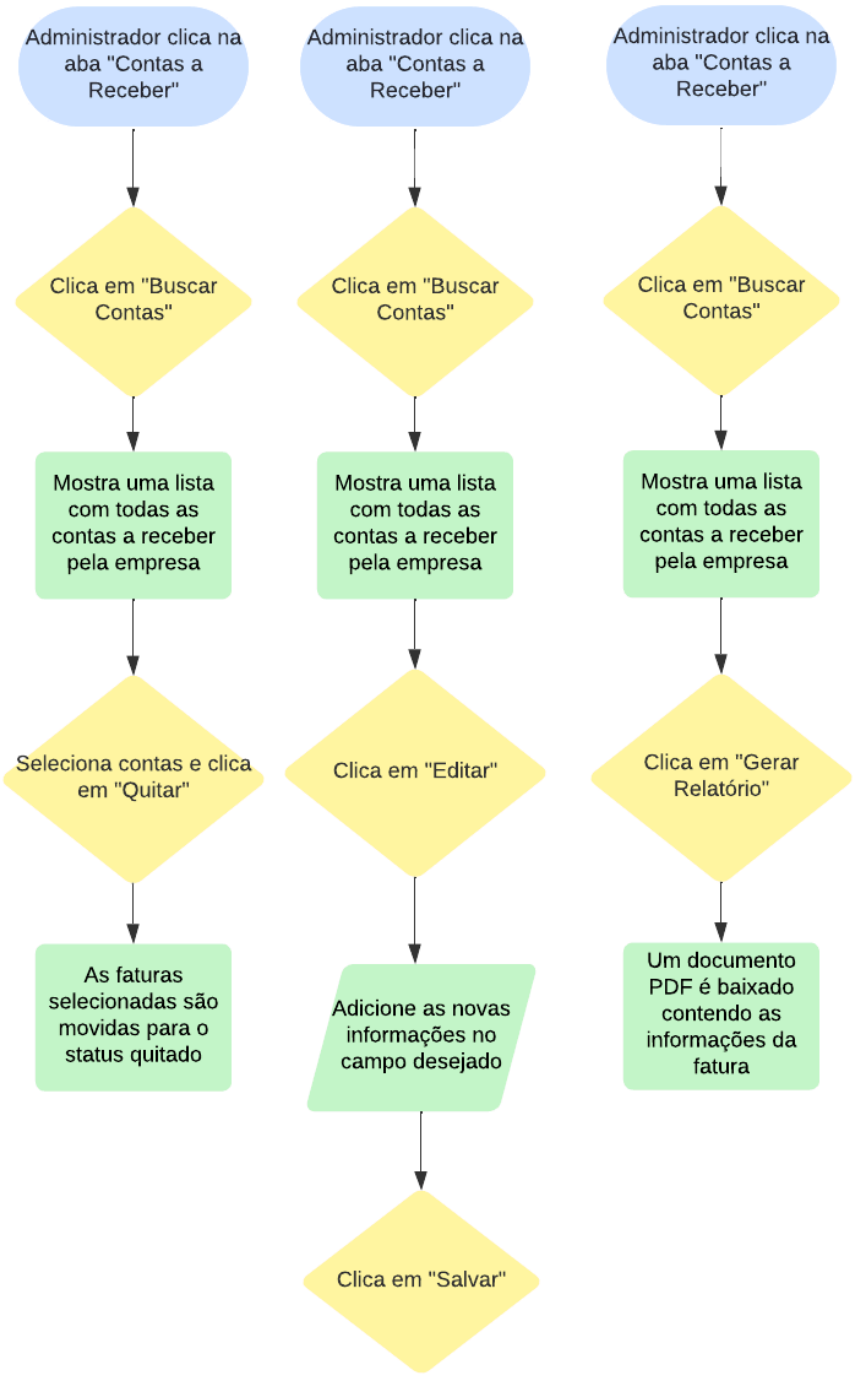
A interface da Figura 13 disponibiliza os fluxos necessários para cumprir os seguintes RFs: RF-5, RF-6, RF-7, RF-8, RF-12 e RF-13. Alguns fluxos definidos no fluxograma não podem ser visualizados na Figura 12, pois seriam necessários um ou mais cliques e, por objetividade, são omitidos.

5.3.2 Fluxos de Contas a Receber

Para demonstrar as rotinas envolvidas no gerenciamento de contas a receber, vamos apresentar 3 fluxos através dos fluxogramas da Figura 15:

- Buscar e quitar contas
- Editar cliente
- Emitir Relatório

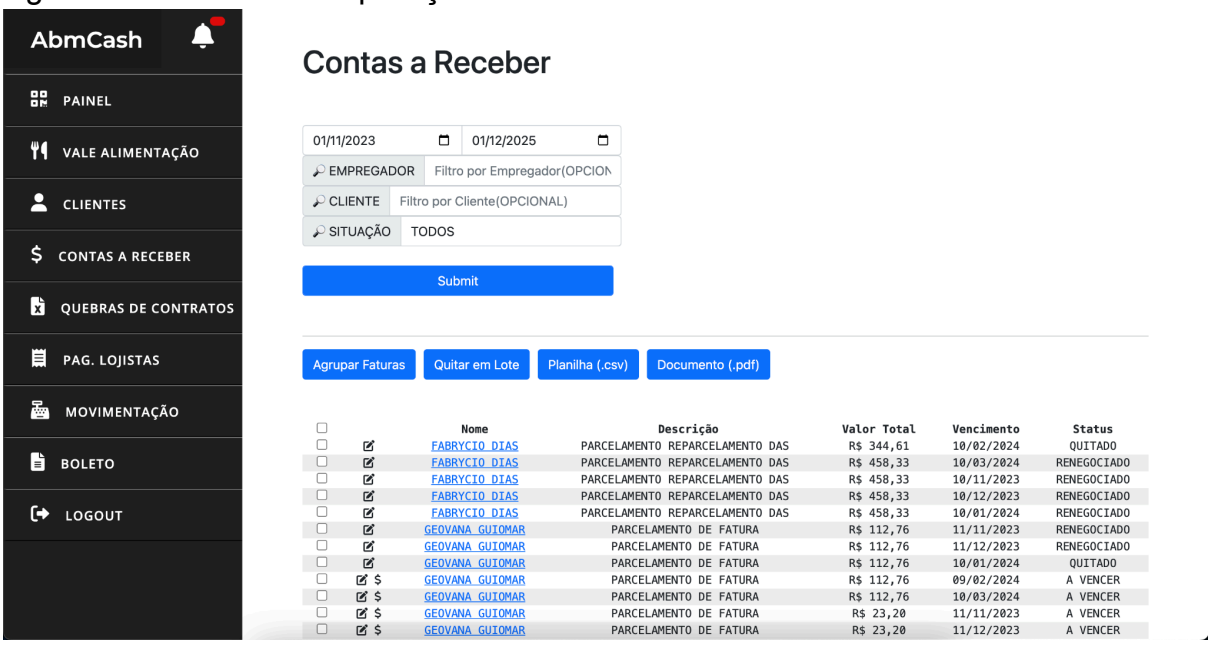
Figura 15 - Interface da aplicação em Contas a Receber



Fonte: Autor (2024).

A Figura 16 representa a visão do administrador quando este se encontra no momento referenciado na Figura 15.

Figura 16 - Interface da aplicação com rotina de Contas a Receber



Fonte: Autor (2024).

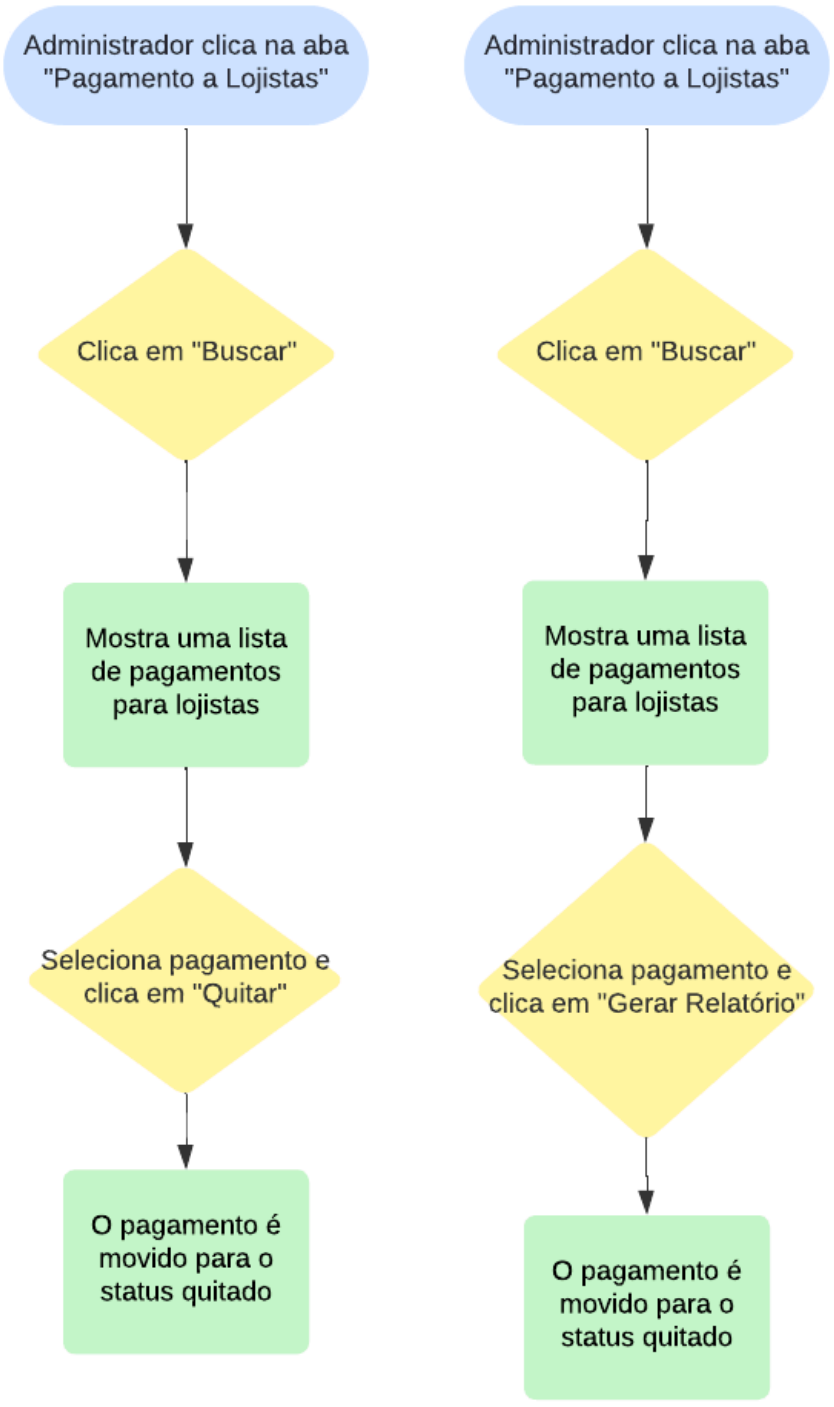
A interface da Figura 16 disponibiliza os fluxos necessários para cumprir os requisitos funcionais RF-2, RF-4 e RF-7. Aqui certos fluxos definidos no fluxograma não podem ser visualizados, pois seriam necessários um ou mais cliques e, por objetividade, são omitidos.

5.3.3 Fluxos de Pagamentos a lojistas

Para demonstrar as rotinas envolvidas no gerenciamento de Pagamentos a lojistas, vamos apresentar 2 fluxos através dos fluxogramas da Figura 17:

- Buscar e quitar contas a lojistas
- Emitir relatórios

Figura 17 - Fluxograma da rotina de gerenciamento de Pagamentos a Lojistas



Fonte: Autor (2024).

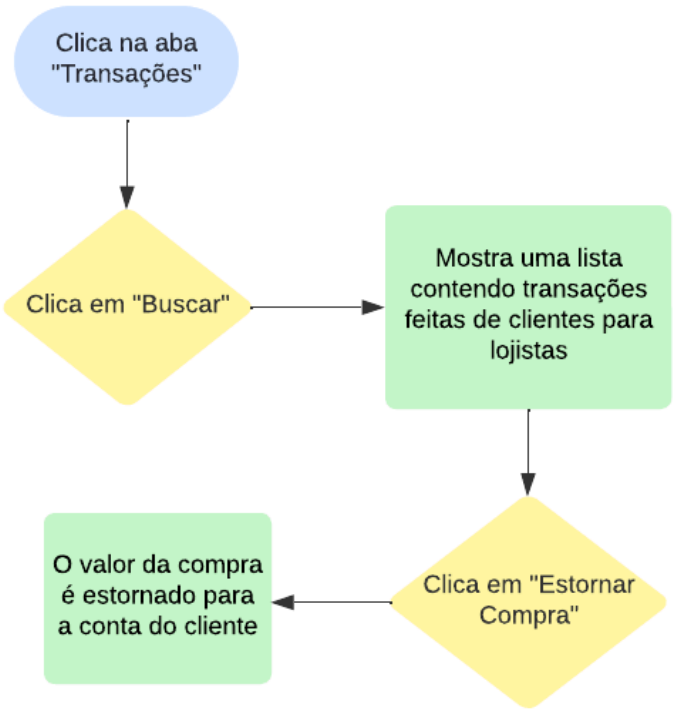
A interface da Figura 17 disponibiliza os fluxos necessários para cumprir os RFs RF-2, RF-3, RF-7, RF-9, RF-10 e RF-11. A imagem que contém a interface referente à Figura 17 foi omitida com a intenção de simplificar o entendimento dos RFs, compondo duas ou mais telas em um único fluxograma.

5.3.4 Fluxos de Transações

Para demonstrar as rotinas envolvidas nas transações, temos um fluxograma na Figura 18:

- Buscar e estornar transações

Figura 18 - Fluxograma da rotina de gerenciamento de Movimentações



Fonte: Autor (2024).

A interface da Figura 18 disponibiliza os fluxos necessários para cumprir os RFs RF-7, RF-8, RF-11. Da mesma forma, postergou-se a imagem que compreende a interface referente a Figura 18 para melhor entendimento de seus RFs, compondo duas ou mais telas em um único fluxograma.

6 CONCLUSÃO

O objetivo do presente trabalho é a resolução da necessidade de uma empresa através da introdução de um novo software. Em termos de desenvolvimentos concretos, obteve-se um aplicativo móvel baseado em *webview* e foi introduzido um motor de faturamento, mantendo a retrocompatibilidade com a antiga interface web. Além disso, uma nova interface web administrativa que proporciona a gestão das operações financeiras.

Notou-se durante o processo de desenvolvimento a liberdade e a facilidade promovidas pelas tecnologias utilizadas, tornando o software simples e o processo de desenvolvimento acelerado.

Para trabalhos futuros, propõe-se a inclusão de transações entre clientes e lojistas, utilizando um fluxo em que o cliente seja capaz de efetuar a solicitação de transações via leitura de um *QR Code*, além também de um PDV que emite alarmes sonoros que notificam uma nova transação ao lojista com capacidade de impressão automática de comprovantes. Além desses novos módulos, também propõe-se a integração do motor de faturamento dentro de um aplicativo de *food service*, possibilitando também aos clientes a utilização de seus saldos em ambiente virtual.

REFERÊNCIAS

1 OQTEM. OQTem official web site. Disponível em https://oqtem.com Acesso em 25/06/2024.

2 FASTAPI. **FastAPI framework, high performance, easy to learn, fast to code, ready for production**. Disponível em https://fastapi.tiangolo.com/ Acesso em 23/05/2024.

3 HTMX. **HTMX official web site**. Disponível em https://htmx.org/ Acesso em 23/05/2024.

4 JINJA. **Jinja official web site**. Disponível em https://jinja.palletsprojects.com/en/3.1.x/ Acesso em 23/05/2024.

5 SQLALCHEMY. **The Database Toolkit for Python**. Disponível em https://www.sqlalchemy.org/ Acesso em 23/05/2024.

6 FLUTTER. **Flutter official web site**. Disponível em https://flutter.dev/ Acesso em 23/05/2024.

7 MYSQL. **MySQL official web site**. Disponível em https://www.mysql.com/ Acesso em 23/05/2024.

8 RABBITMQ. **One broker to queue them all**. Disponível em https://www.rabbitmq.com/ Acesso em 23/05/2024.

9 REDIS. **Redis official web site**. Disponível em https://redis.io/ Acesso em 23/05/2024.

10 CONTE, T.; TRAVASSOS, G. H.; MENDES, E. **Revisão Sistemática sobre Processos de Desenvolvimento para Aplicações Web.** Relatório Técnico ESE/PESC–COPPE/UFRJ, 2005.

11 ZIEMER, S. **An architecture for web applications**. essay in dif 8914 distributed information systems. [s.l: s.n.].

12 XING, Y.; HUANG, J.; LAI, Y. **Research and analysis of the front-end frameworks and libraries in E-business development.** Proceedings of the 2019 11th International Conference on Computer and Automation Engineering. Anais...New York, NY, USA: ACM, 2019.

13 REACT. **React official web site**. Disponível em https://react.dev/ Acesso em 23/05/2024.

14 BOOTSTRAP. **Bootstrap official web site**. Disponível em https://getbootstrap.com/ Acesso em 23/05/2024.

15 LARAVEL. **Laravel official web site**. Disponível em https://laravel.com/ Acesso em 23/05/2024.

16 RUBY. **Ruby on Rails official web site**. Disponível em https://rubyonrails.org/ Acesso em 23/05/2024.

17 SPRING. **Java Spring official web site**. Disponível em https://spring.io/ Acesso em 23/05/2024.

18 HARO PERALTA, J. **Microservice APIs: Using python, flask, FastAPI, OpenAPI and more.** Nova Iorque, NY, USA: Manning Publications, 2023.

19 EHSAN, A. et al. RESTful API testing methodologies: Rationale, challenges, and solution directions. **Applied sciences** (Basel, Switzerland), v. 12, n. 9, p. 4369, 2022.

20 TRIKHATRI, S. **A Project Report on Creating Back-End For Online Business Site Utilizing FastAPI**. Siquim, 2022.

21 FLASK. **Flask official web site**. Disponível em https://flask.palletsprojects.com/en/3.0.x/ Acesso em 23/05/2024.

22 NILSSON, E.; DEMIR, D. **Performance comparison of REST vs GraphQL in different web environments : Node.js and Python.** [s.l: s.n.].

23 TORRES, A. et al. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. **Information and software technology**, v. 82, p. 1–18, 2017.

24 DOLGERT, A.; GIBBONS, L.; KUZNETSOV, V. **Rapid web development using AJAX and Python.** Journal of physics. Conference series, v. 119, n. 4, p. 042011, 2008.

25 FREDSTAM, M.; JOHANSSON, G. **Comparing database management systems with SQLAlchemy : A quantitative study on database management systems.** [s.l: s.n.].

26 ADINUGROHO, T. Y.; REINA; GAUTAMA, J. B. Review of multi-platform mobile application development using WebView: Learning management system on mobile platform. **Procedia computer science**, v. 59, p. 291–297, 2015.

27 ĆATOVIĆ, A.; BUZAĐIJA, N.; LEMES, S. **Microservice development using RabbitMQ message broker** . Science, Engineering and Technology, [S. l.], v. 2, n. 1, p. 30–37, 2022.

28 PYPI. **Pika python AMQP Client Library**. Disponível em https://pypi.org/project/pika/ Acesso em 23 maio 2024.

29 DOCKER HUB. **Imagem Docker oficial RabbitMQ**. Disponível em https://hub.docker.com/_/rabbitmq Acesso em 23/05/2024.

30 ZULFA, M. I.; FADLI, A.; WARDHANA, A. W. Application caching strategy based on in-memory using Redis server to accelerate relational data access. **Jurnal teknologi dan sistem komputer**, v. 8, n. 2, p. 157–163, 2020.

31 JOUPPI, N. P. Cache write policies and performance. **Computer architecture news**, v. 21, n. 2, p. 191–201, 1993.

32 SARDAGNA, M.; VAHLDICK, A. **Aplicação do Padrão Data Access Object (DAO) em Projetos Desenvolvidos com Delphi**. Departamento de Sistemas e Computação Universidade Regional de Blumenau.

33 VAZQUEZ, C. E.; SIMÕES, G. S. **Engenharia de Requisitos**: software orientado ao negócio. [s.l.] Brasport, 2016.

34 CHUNG, L., NIXON, B. A., YU, E. e MYLOPOULOS, J.; **Non-Functional Requirements in Software Engineering**, 1 ed. Springer US, 2000.

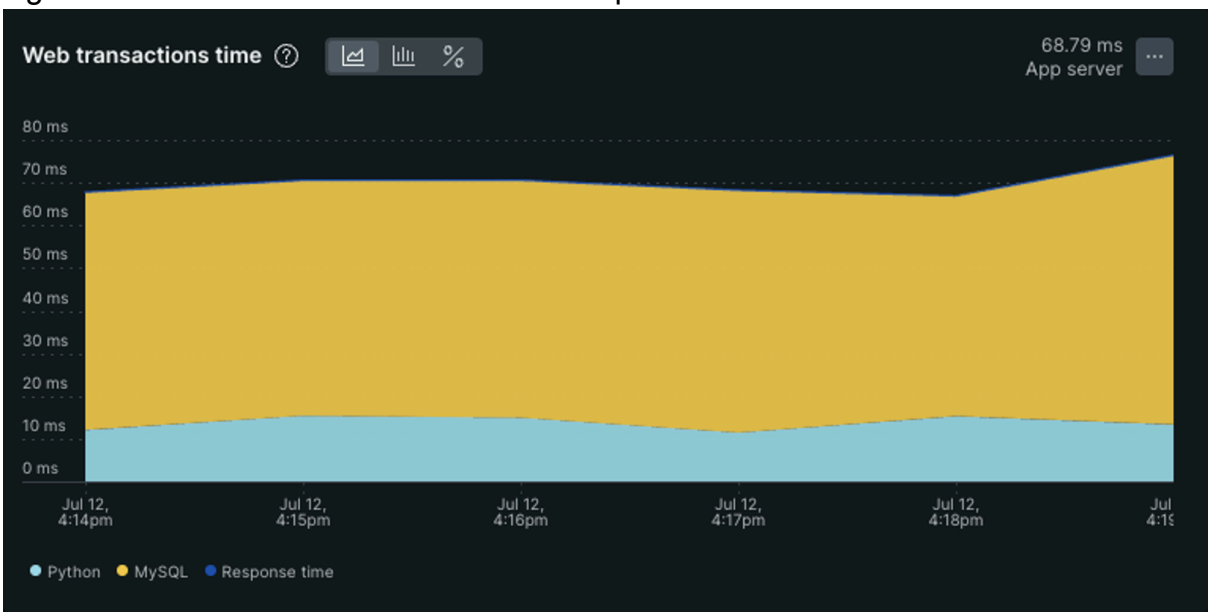
35 ANDRADE, E. L. Problemas de Congestionamento das Filas. In: ANDRADE, E. L. **Introdução à Pesquisa Operacional: Métodos e modelos para análise de decisões**. Ed. 4. Rio de Janeiro : LTC, 2009. Cap. 6, p. 104- 120.

36 NEW RELIC. **New relic official web site**. Disponível em https://newrelic.com/ Acesso em 23/05/2024.

APÊNDICE A - Gráficos do Impacto do Cache no Tempo de Resposta do Sistema

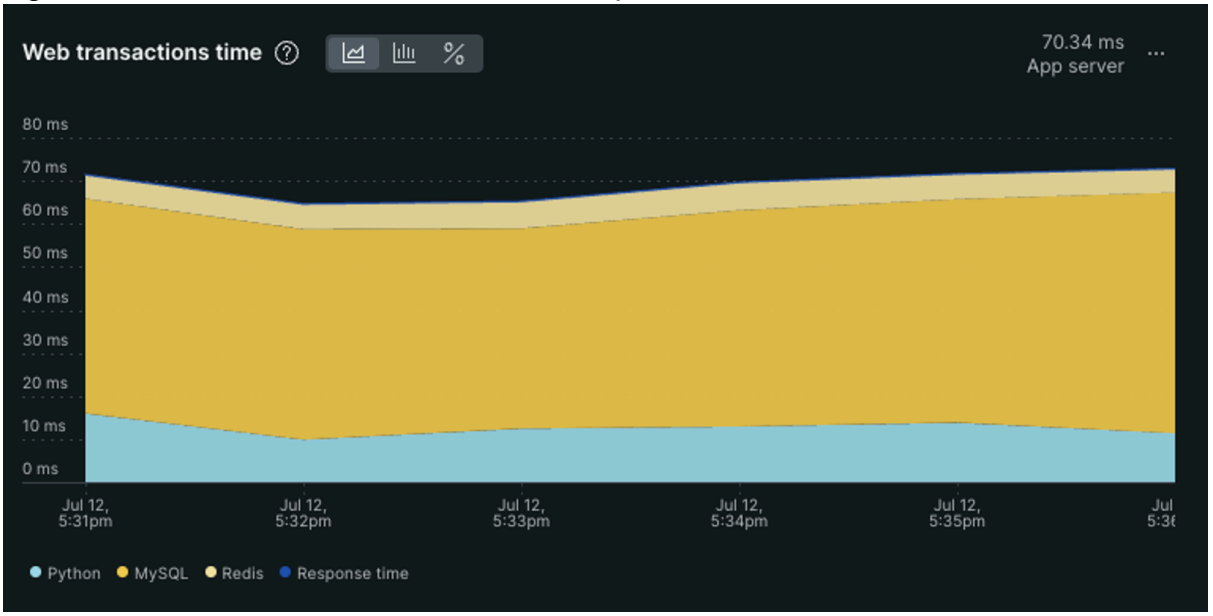
Com o objetivo de obter o tempo médio de resposta do sistema com a utilização do cache, fez-se a coleta de métricas com o New Relic [36] em diferentes cenários de consumo de saldo com um percentual diferente de acerto no cache. Tal percentual varia desde 0% de acerto chegando até 100% de acerto em saltos de 10%, os gráficos das Figuras 19 até 29 demonstram os resultados de cada caso.

Figura 19 - Métricas de consumo de cache para 0% de acerto



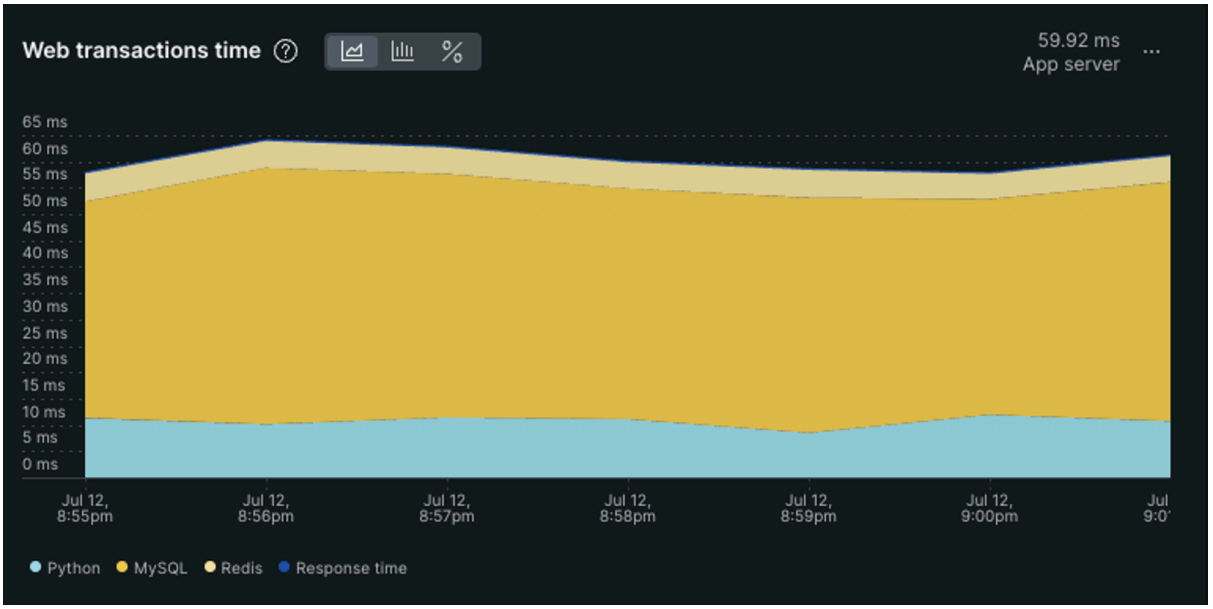
Fonte: Autor (2024).

Figura 20 - Métricas de consumo de cache para 10% de acerto



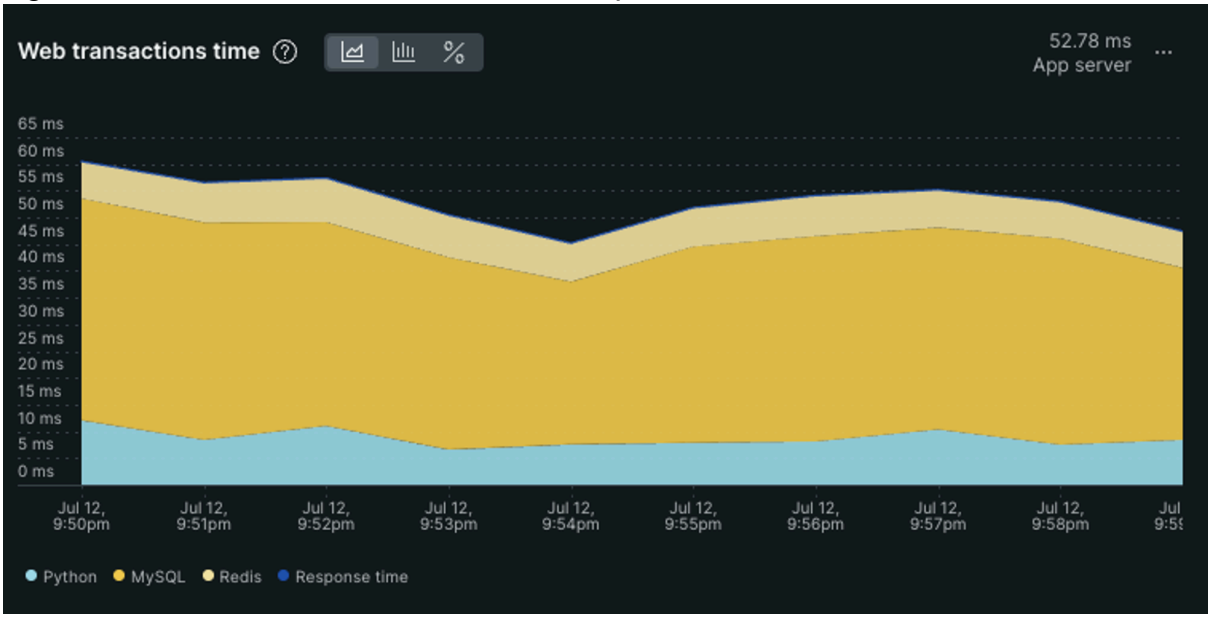
Fonte: Autor (2024).

Figura 21 - Métricas de consumo de cache para 20% de acerto



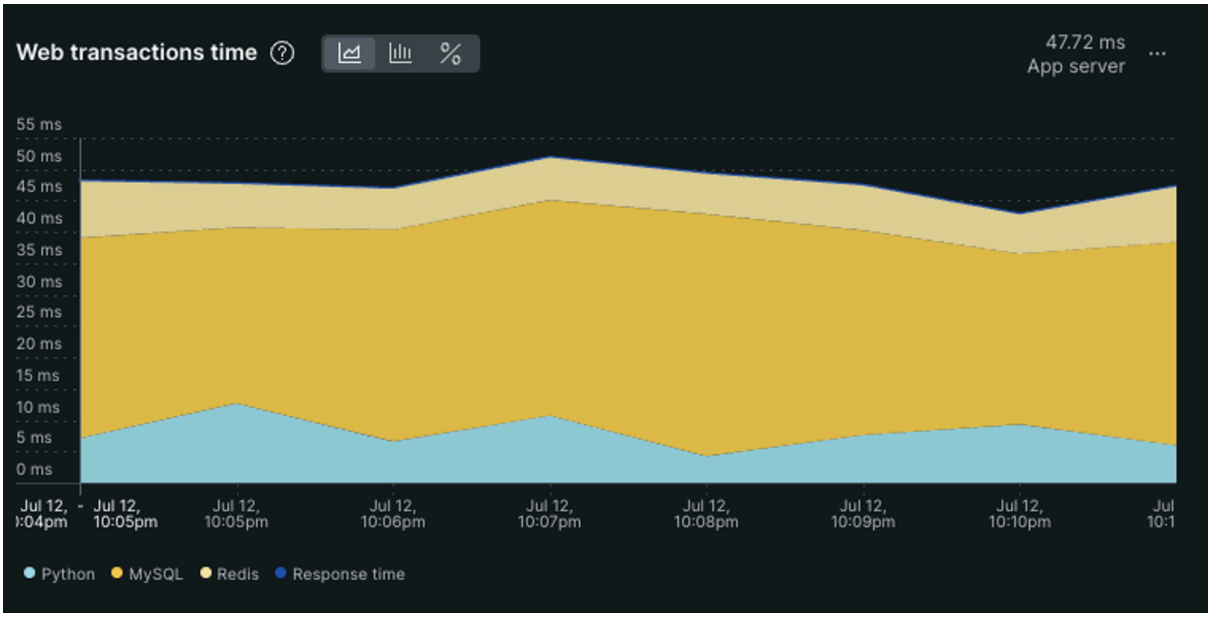
Fonte: Autor (2024).

Figura 22 - Métricas de consumo de cache para 30% de acerto



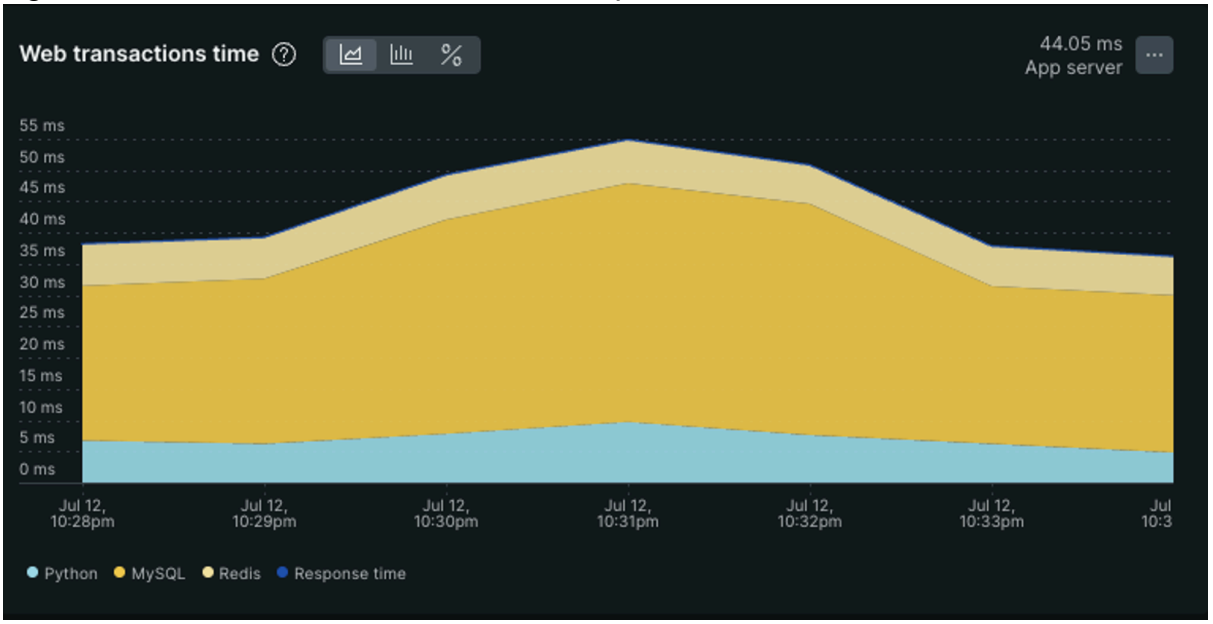
Fonte: Autor (2024).

Figura 23 - Métricas de consumo de cache para 40% de acerto



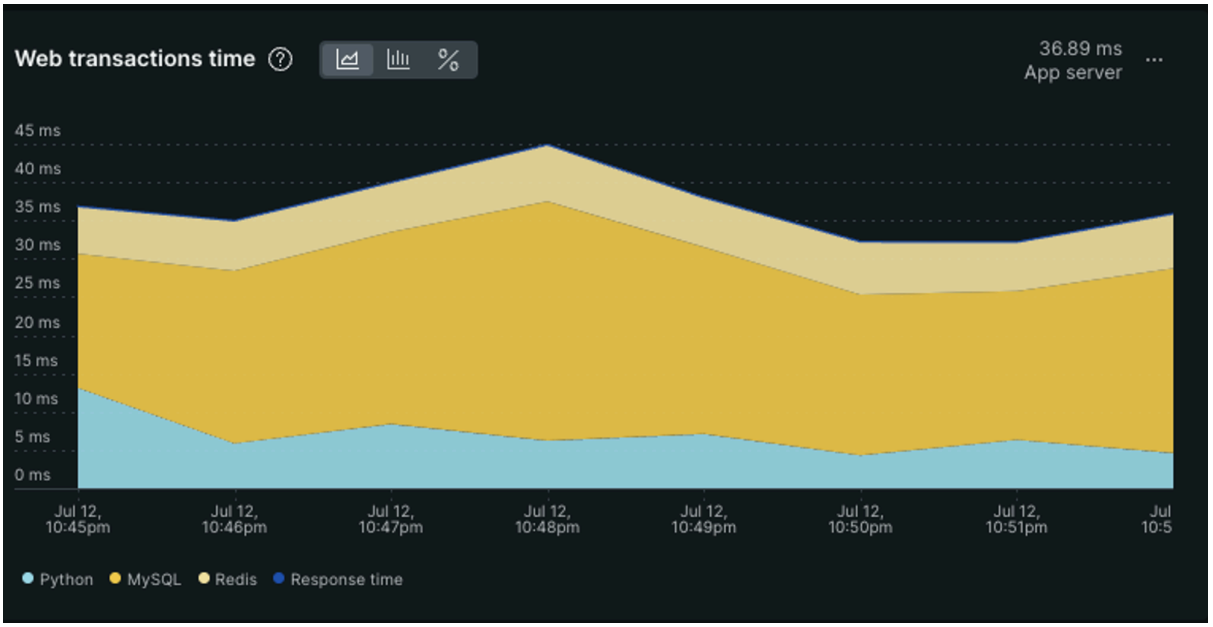
Fonte: Autor (2024).

Figura 24 - Métricas de consumo de cache para 50% de acerto



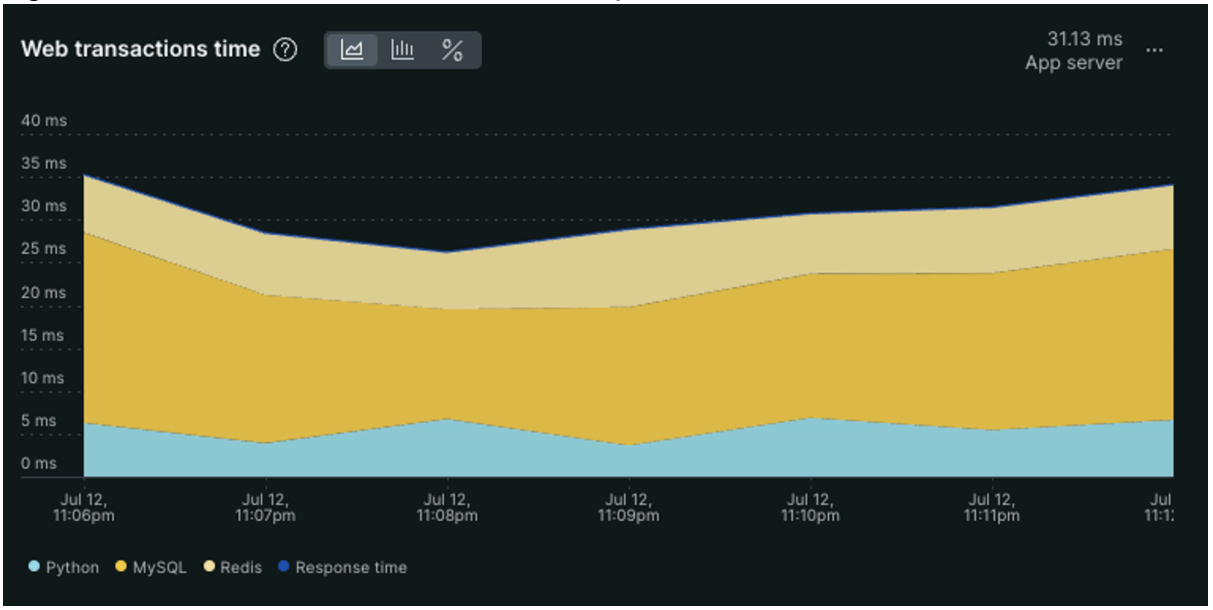
Fonte: Autor (2024).

Figura 25 - Métricas de consumo de cache para 60% de acerto



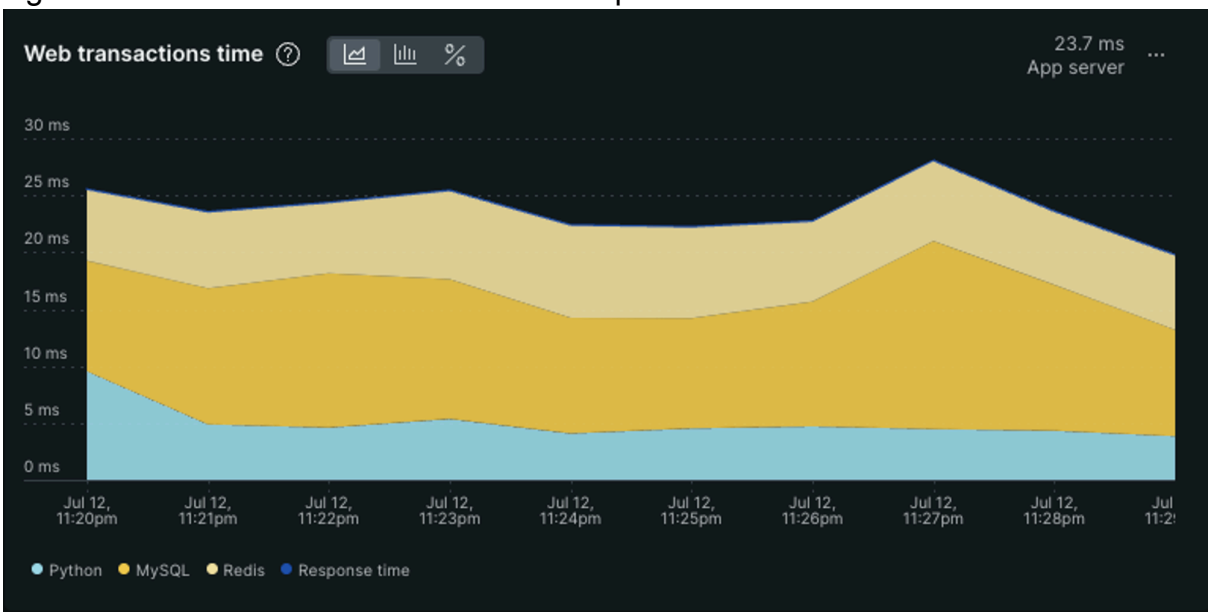
Fonte: Autor (2024).

Figura 26 - Métricas de consumo de cache para 70% de acerto



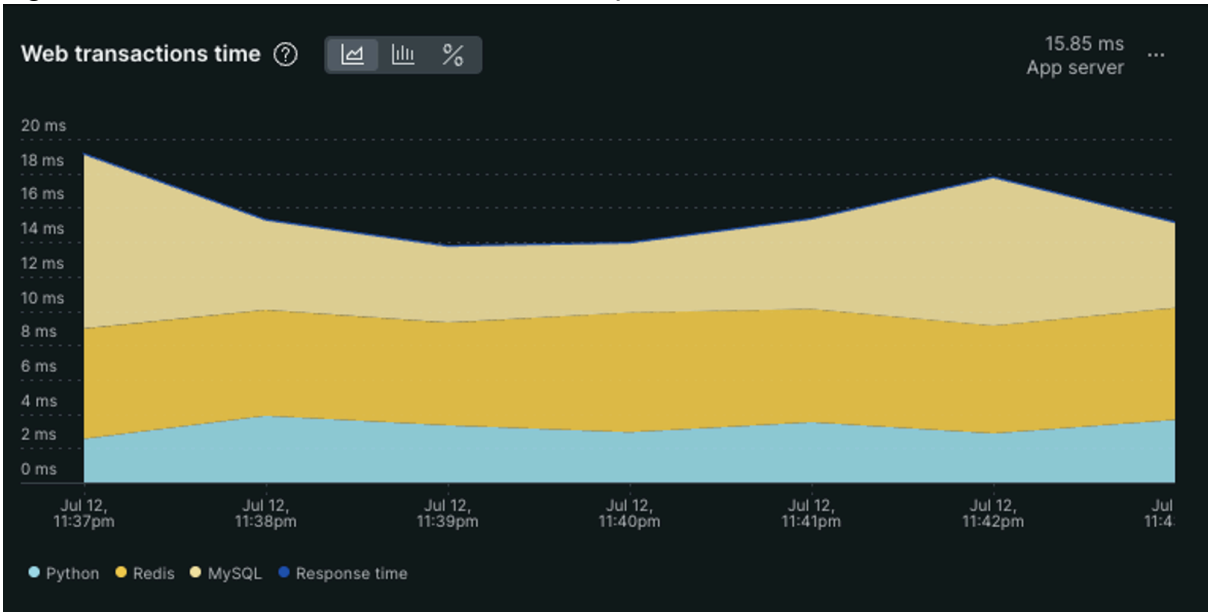
Fonte: Autor (2024).

Figura 27 - Métricas de consumo de cache para 80% de acerto



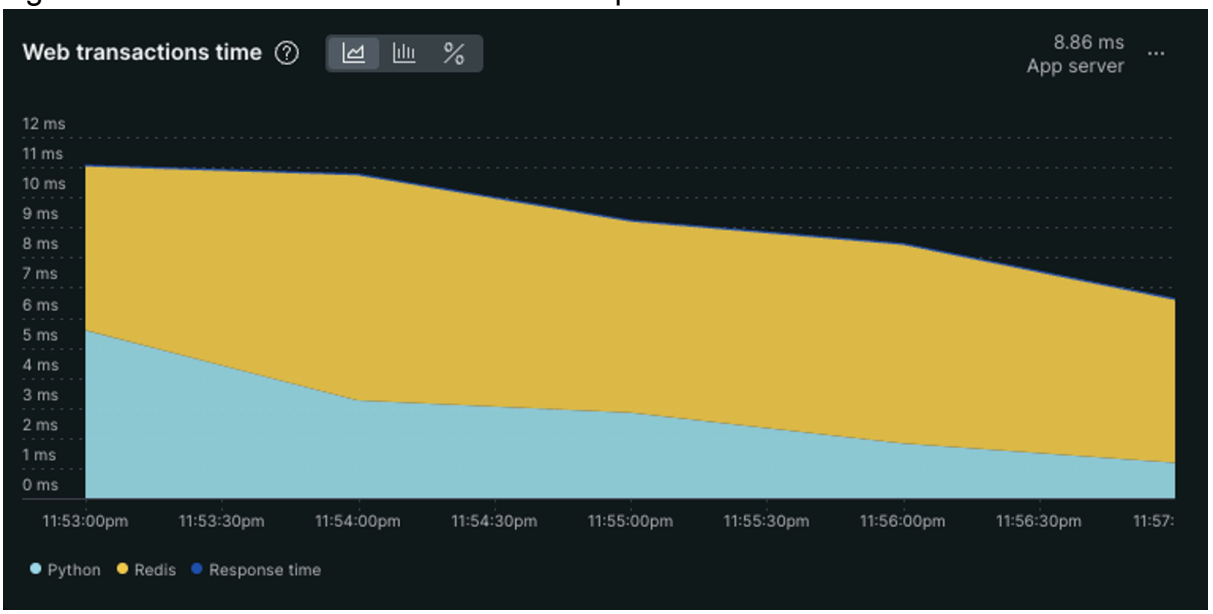
Fonte: Autor (2024).

Figura 28 - Métricas de consumo de cache para 90% de acerto



Fonte: Autor (2024).

Figura 29 - Métricas de consumo de cache para 100% de acerto

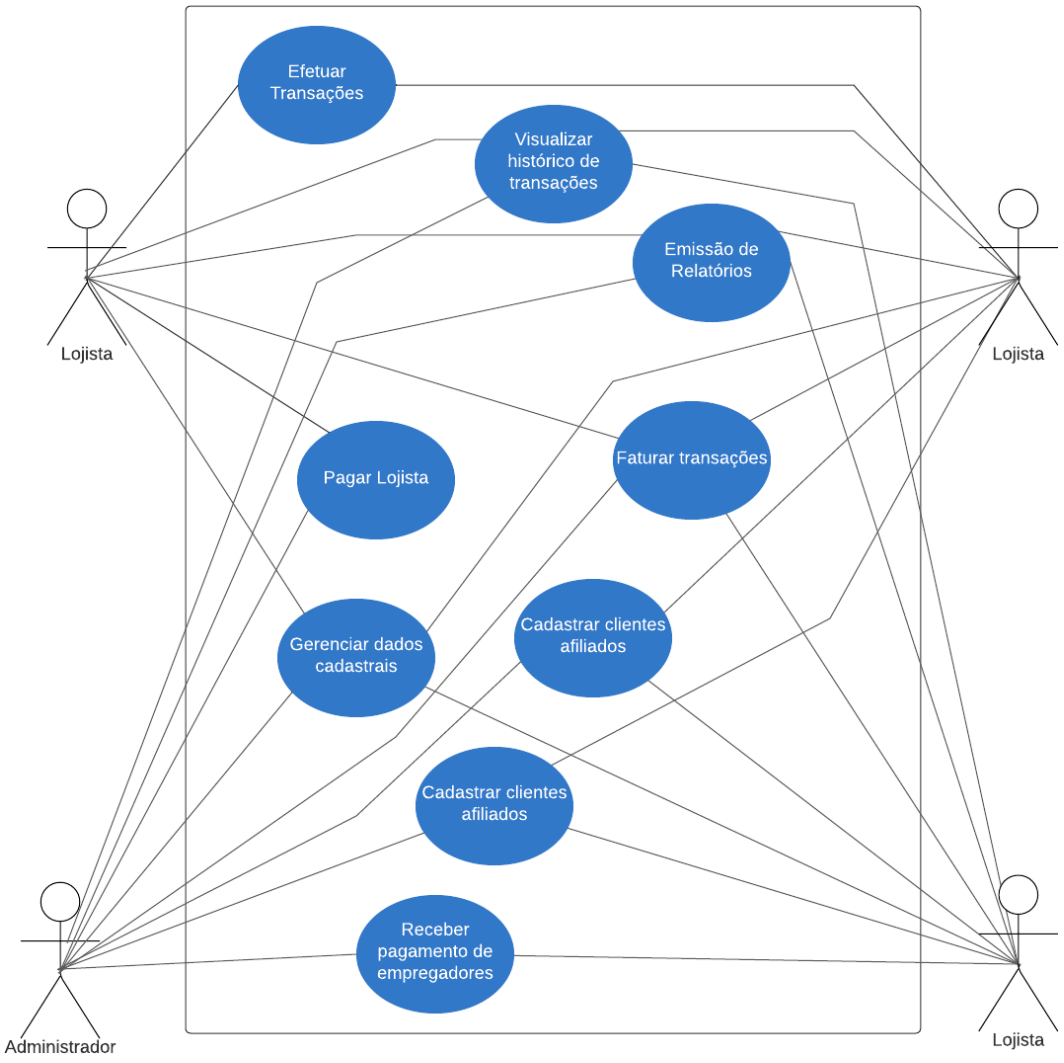


Fonte: Autor (2024).

APÊNDICE B - Modelagem dos Casos de Uso

Com base na definição de requisitos da Tabela 2, é possível fazer a modelagem de um diagrama de casos de uso ilustrando tais requisitos. Esse diagrama pode ser visualizado na Figura 30.

Figura 30 - Diagrama de Casos de Uso



Fonte: Autor (2024).