**UNIVERSIDADE DE CAXIAS DO SUL**
**ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS**

**AUGUSTO FERNANDO KLEIN**

# DESENVOLVIMENTO DE PROTÓTIPO PARA GESTÃO DE REPLICAÇÃO ENTRE BASES DE DADOS REMOTAS

**BENTO GONÇALVES**

**2024**

**AUGUSTO FERNANDO KLEIN**

# DESENVOLVIMENTO DE PROTÓTIPO PARA GESTÃO DE REPLICAÇÃO ENTRE BASES DE DADOS REMOTAS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Profª. Dra. Helena Graziottin Ribeiro

**BENTO GONÇALVES**

**2024**

**AUGUSTO FERNANDO KLEIN**

# DESENVOLVIMENTO DE PROTÓTIPO PARA GESTÃO DE REPLICAÇÃO ENTRE BASES DE DADOS REMOTAS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

**Aprovado em 00/00/2024**

**BANCA EXAMINADORA**

---

Profª. Dra. Helena Graziottin Ribeiro
Universidade de Caxias do Sul - UCS

---

Prof. Dr. Daniel Luis Notari
Universidade de Caxias do Sul - UCS

---

Prof. Me. Alexandre E. Krohn Nascimento
Universidade de Caxias do Sul - UCS

# AGRADECIMENTOS

Primeiramente, inicio os agradecimentos aos meus pais, Flávio e Alice, que foram fundamentais durante minha jornada acadêmica e de vida. Foram eles que me ajudaram a construir meu caráter e minha índole, proporcionando o mais importante para que eu pudesse conquistar meus sonhos e objetivos.

Ao meu irmão Gustavo, que sempre me apoiou nas minhas escolhas de vida. Nos momentos em que me sentia receoso ao tomar decisões, foi ele quem me mostrou que o momento certo é o agora. Muito obrigado por ser esse irmão maravilhoso, que sempre esteve ao meu lado e me apoiou.

À minha namorada Rafaela, que foi minha base para que eu pudesse seguir meus sonhos e seguir em frente. Ela foi e continua sendo minha companheira de vida, sempre soube me apoiar nos momentos mais importantes e me ajudou a chegar até aqui. Sem dúvida, essa conquista é tanto minha quanto sua. Te amo muito.

À minha orientadora de pesquisa, Profª. Dra. Helena Graziottin Ribeiro, que não mediu esforços para me auxiliar no desenvolvimento deste trabalho. Sem dúvida, seu apoio nos encontros semanais, inclusive fora do horário, fez toda a diferença para a entrega de um trabalho mais completo e enriquecedor.

Agradeço aos meus colegas da universidade, com quem a troca de conhecimento foi sempre rica e proveitosa. Com vocês, estudar se tornou mais leve, com trocas de experiências e auxílio mútuo que nos ajudaram a nos desenvolver aula após aula.

Agradeço também a todos os professores da universidade, que me ajudaram a me tornar um amante do conhecimento e um profissional mais qualificado. Além da troca de conhecimento, sou grato pelos ótimos momentos em sala de aula, os quais levarei com muito carinho ao longo da minha jornada de eterno aprendiz.

Agradeço à universidade por me proporcionar o acesso ao conhecimento, por meio de sua estrutura, salas de aula, laboratórios e professores incríveis. No vestibular, escolhi a universidade por priorizar a busca pelo conhecimento e, hoje, concluo que fiz a escolha certa, tanto para meu desenvolvimento pessoal quanto profissional.

Sou muito grato à Tramontina por me oferecer todo o suporte necessário para o desenvolvimento deste trabalho. A ideia do projeto surgiu internamente, motivada por uma necessidade de aprimorar os processos de replicação entre diferentes bases de dados. Espero que, com o desenvolvimento deste trabalho, eu possa agregar valor ao negócio e torná-lo mais competitivo e sustentável no mercado.

Agradeço também aos meus colegas de trabalho, que me apoiaram e me ajudaram

de alguma forma na escolha e desenvolvimento do TCC. Momentos especiais de troca de conhecimento e discussões produtivas sem dúvida contribuíram para o meu desenvolvimento e fizeram toda a diferença para que eu chegasse até aqui.

Gostaria de expressar meu profundo agradecimento a todos vocês que, de alguma forma, me ajudaram a transformar meu sonho em realidade. Muito obrigado!

*"O conhecimento torna a alma jovem e diminui a armagura da velhice. Colhe, pois, a sabedoria. Armazena suavidade para o amanhã."*

***Leonardo da Vinci***

# RESUMO

A disponibilidade de dados é um tema muito relevante, especialmente no que diz respeito à sua alta qualidade para diversas aplicações, que frequentemente exigem acesso imediato. Em sistemas distribuídos, os dados podem estar localizados em locais diferentes de onde as aplicações são executadas, o que torna necessário o desenvolvimento de técnicas eficientes de gestão para garantir a disponibilidade e o acesso a esses dados. Este trabalho realiza uma pesquisa sobre as características dos Sistemas de Gerenciamento de Banco de Dados (SGBD) relacionais e distribuídos, com foco principal nas técnicas nativas de replicação presentes nos SGBDs mais utilizados no mercado. O Informix é um SGBD relacional que suporta replicação de dados, mas sob a gestão de um Database Administrator (DBA). O objetivo do trabalho é desenvolver um protótipo de *middleware* acessado em ambiente WEB, dividido em *backend* e *frontend*, integrado a um Enterprise Resource Planning (ERP) do Informix, para servir como uma ferramenta complementar de centralização das manutenções que envolvem as estruturas de replicação em uma empresa. O protótipo disponibiliza uma interface de visualização dos dados para determinados desenvolvedores responsáveis de cada equipe de trabalho, com o objetivo de acompanhar os processos. O protótipo desenvolvido já está sendo utilizado na empresa e atualmente está em processo de homologação.

**Palavras-chave**: Banco de dados. Replicação. Enterprise Replication. Gestão de replicação.

# ABSTRACT

Data availability is a highly relevant topic, especially regarding its high quality for various applications, which often require immediate access. In distributed systems, data may be located in places different from where applications are executed, making it necessary to develop efficient management techniques to ensure the availability and access to such data. This work conducts research on the characteristics of relational and distributed Database Management Systems (DBMS), focusing primarily on the native replication techniques present in the most widely used DBMSs on the market. Informix is a relational DBMS that supports data replication but under the management of a Database Administrator (DBA). The objective of this work is to develop a middleware prototype accessed in a WEB environment, divided into backend and frontend, integrated with an Enterprise Resource Planning (ERP) system of Informix, to serve as a complementary tool for centralizing maintenance involving replication structures within a company. The prototype provides a data visualization interface for specific developers responsible for each work team, aiming to monitor the processes. The developed prototype is already being used in the company and is currently undergoing approval testing.

**Keywords**: Database. Replication. Enterprise Replication. Replication Management.

# LISTA DE FIGURAS

# LISTA DE QUADROS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| **2PC** | 2 Phase Commit |
| **ACID** | Atomicidade, Consistência, Isolamento e Durabilidade |
| **AAA** | Arrange, Act, Assert |
| **CLI** | Common Language Infrastructure |
| **CLR** | Common Language Runtime |
| **CQRS** | Command Query Responsibility Segregation |
| **CLU** | Command-line Utility |
| **DBA** | Database Administrator |
| **DBMS** | Database Management Systems |
| **ER** | Enterprise Replication |
| **ERP** | Enterprise Resource Planning |
| **GTID** | Global Transaction Identifiers |
| **HTML** | HyperText Markup Language |
| **HTTP** | Hypertext Transfer Protocol |
| **IBM** | International Business Machine Corporation |
| **JSON** | JavaScript Object Notation |
| **MBR** | Mixed Base Replication |
| **MVC** | Model-View-Controller |
| **RBR** | Row Based Replication |
| **REST** | Representation State Transfer |
| **SBR** | Statement Based Replication |
| **SGBD** | Sistemas de Gerenciamento de Banco de Dados |
| **SPL** | Stored Procedure Language |
| **SQL** | Structured Query Language |
| **URI** | Uniform Resource Identifier |
| **WAN** | Wide-area Network |
| **WWW** | World Wide Web |
| **XML** | Extensible Markup Language |

# SUMÁRIO

# 1 INTRODUÇÃO

A disponibilidade de dados é uma necessidade essencial para os sistemas atuais, com grande interconectividade proporcionada pela Internet e fácil acesso a diversos sistemas distribuídos que oferecem recursos de disponibilidade e até alta disponibilidade. Ao utilizar métodos de distribuição de dados em ambientes empresariais, a depender de seu uso e da organização da arquitetura, eles podem ter grande influência no sucesso ou fracasso do negócio (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

O acesso à informação é um fator de vantagem em um mercado competitivo. Ou seja, oferecer facilidade e praticidade no acesso aos dados para o cliente ou usuário pode ser considerado uma estratégia para atraí-lo a usar a plataforma, como uma forma de satisfazer suas necessidades. Quando for necessário utilizar os dados na plataforma, estes precisam estar disponíveis em um grau de proximidade elevado para a execução da aplicação, pois quanto mais próximos os dados estiverem, melhor será o nível de resposta. Para que esse processo ocorra e o acesso aos dados em sistemas distribuídos — que têm sua execução e dados distribuídos em diferentes servidores —, uma estratégia que pode ser adotada na implementação é a utilização de técnicas de replicação (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

A replicação de dados envolve a criação de cópias de informações que serão sincronizadas em múltiplas bases de dados, dispostas em diferentes locais geográficos. O processo de replicação visa aumentar a tolerância a falhas e melhorar o desempenho do sistema, tornando o acesso às informações mais ágil por meio de bases mais próximas. Entre os desafios na implementação dessa arquitetura, podemos considerar a consistência e a disponibilidade dos dados em todas as bases, levando em conta as possíveis falhas que possam ocorrer na rede (MOIZ *et al.*, 2011).

Considerando os benefícios de utilizar a replicação, muitos sistemas distribuídos contam com ferramentas complementares desenvolvidas para funcionar como *middleware*, baseadas em estruturas distintas integradas ao sistema atual. Dessa forma, mecanismos de replicação têm sido disponibilizados como ferramenta padrão em muitos SGBDs de mercado, como Oracle, PostgreSQL, SQL Server, MySQL, Informix, entre outros, com o objetivo de garantir processos nativos, simplificando os fluxos internos do sistema (MOIZ *et al.*, 2011).

Embora a gestão da replicação seja um recurso dos SGBDs, pode haver necessidades específicas de gestão de replicação em aplicações que utilizam os dados, as quais precisam ser desenvolvidas em coordenação com o SGBD distribuído. O presente trabalho visa explorar os conceitos e mecanismos de funcionamento do SGBD Informix, com suas ferramentas de replicação destinadas a sistemas distribuídos, propondo a integração de uma solução para dar maior autonomia na gestão das estruturas distribuídas pelos líderes de equipes da área de tecnologia da informação (IBM Corporation, 2010).

## 1.1 PROBLEMA

O problema identificado para este trabalho concentra-se em prover uma gestão ativa de uma estrutura de replicação composta por diversos servidores geograficamente distribuídos, na qual os DBAs de um ambiente corporativo de uma empresa localizada na Serra Gaúcha precisam garantir que as informações estejam disponíveis para que as aplicações possam acessá-las corretamente. Dessa forma, o presente trabalho se baseia na resolução do seguinte problema: é possível criar um protótipo que auxilie na manutenção e visibilidade das estruturas de replicação por parte dos DBAs?

## 1.2 OBJETIVOS

O presente trabalho está organizado em um objetivo geral e alguns objetivos específicos.

### 1.2.1 Objetivo geral

O objetivo deste trabalho é projetar e desenvolver um protótipo de gestão de replicação vinculado ao SGBD distribuído Informix que auxilie no monitoramento e manutenção das estruturas de replicação por parte dos DBAs.

### 1.2.2 Objetivos específicos

1. Documentar o funcionamento de diferentes metodologias de replicação em diferentes SGBDs.
2. Apresentar uma proposta de gestão da replicação para diminuir o gargalo no atendimento aos usuários.
3. Desenvolver um protótipo de uma aplicação para realizar a gestão do sistema de replicação.

## 1.3 ESTRUTURA DO TRABALHO

O documento foi estruturado em seções, onde cada uma é responsável por exemplificar o entendimento como um todo.

O presente trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta a estrutura teórica de operação e funcionamento dos SGBDs distribuídos.
- O Capítulo 3 aborda as principais características de replicação de alguns SGBDs distribuídos.

- O Capítulo 4 descreve a estrutura de organização e desenvolvimento do protótipo proposto para o trabalho.
- O Capítulo 5 detalha o desenvolvimento do protótipo.
- O Capítulo 6 fornece um estudo de caso de uso do protótipo e o detalhamento dos testes realizados.
- O Capítulo 7 apresenta as conclusões do trabalho.

# 2 ESTRUTURA DE SISTEMAS DE BANCO DE DADOS

Os sistemas de banco de dados (SGBDs) são ferramentas essenciais para aplicações, garantindo organização e acesso adequado aos conjuntos de dados manipulados. Este capítulo apresenta algumas características da estrutura dos SGBDs, abordando tanto as arquiteturas centralizada e distribuída quanto os recursos necessários para sua implementação.

## 2.1 ARQUITETURA DE BANCO DE DADOS

Ao desenvolver uma aplicação, uma parte fundamental do processo é a definição e a estruturação de um SGBD, que gerenciará grandes volumes de dados, além de fornecer mecanismos de manipulação, controle e segurança no acesso.

> "Um SGBD é uma coleção de dados inter-relacionados e um conjunto de programas para acessar esses dados. A coleção de dados, normalmente conhecida como banco de dados, contém informações relevantes para uma empresa. O principal objetivo de um SGBD é proporcionar uma forma de armazenar e recuperar informações de um banco de dados de maneira conveniente e eficiente."(SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

A arquitetura básica dos SGBDs é centralizada, ou seja, todos os processos são concentrados em uma única máquina, que gerencia as requisições. Com a crescente utilização de aplicações distribuídas, a arquitetura foi adaptada para suportar o processamento e o acesso distribuídos. Isso resultou no desenvolvimento de bancos de dados paralelos e distribuídos, que dividem dados e processamento entre vários nós da rede, permitindo o acesso às informações de forma distribuída (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

### 2.1.1 Bancos Centralizados

Um SGBD centralizado opera em um único sistema, que gerencia todas as requisições ao banco de dados. Esse tipo de banco pode ser de dois tipos de acesso: monousuário e multiusuário, com diferentes abordagens dependendo da metodologia adotada.

No modo monousuário, o SGBD é utilizado em dispositivos que não exigem alto grau de concorrência nos dados, como em smartphones, onde apenas um usuário acessa os dados. Em contraste, no modo multiusuário, diversos usuários acessam o banco simultaneamente, o que pode gerar concorrência entre os processos, exigindo controle rigoroso de acesso, como transações no servidor para garantir a integridade dos dados.

### 2.1.2 Arquitetura de Sistemas de Servidores

O servidor é a máquina na qual o SGBD executa para atender às requisições dos usuários. Existem dois tipos principais de servidores: servidores de transações e servidores de dados. A seguir, apresentam-se as características e o funcionamento de cada tipo.

#### 2.1.2.1 Características das Transações

Uma transação de banco de dados é um conjunto de operações que constitui uma operação mais complexa, e ela possui garantias das propriedades Atomicidade, Consistência, Isolamento e Durabilidade (ACID).

A propriedade de atomicidade garante que todas as operações de uma transação sejam executadas ou, em caso de falha, todas as modificações sejam desfeitas, mantendo a integridade do banco de dados.

A consistência assegura que, após a execução de uma transação, a base de dados permaneça em um estado válido, realizando checagens de precisão nas variáveis que armazenam as informações.

O isolamento permite que transações concorrentes acessem os dados sem interferir umas nas outras, de modo que cada transação perceba o acesso aos dados como exclusivo. Caso uma transação precise acessar um dado utilizado por outra, ela aguardará a conclusão da primeira.

A durabilidade garante que, mesmo em caso de falhas como perda de energia, os dados modificados por transações sejam gravados de forma permanente em uma memória não volátil (LOTFY *et al.*, 2014).

Uma transação é composta por uma ou várias operações que modificam o estado dos dados no SGBD. Ela é projetada para garantir que todas as operações sejam executadas em conjunto, preservando a integridade dos dados, mesmo em caso de falhas.

Em sistemas com alto grau de concorrência, é possível implementar controle de bloqueios para gerenciar o acesso simultâneo a estruturas de dados. Este mecanismo é importante em ambientes multiusuários, evitando inconsistências durante o acesso concorrente.

No entanto, a utilização de bloqueios pode gerar *deadlocks*, situações em que duas ou mais transações ficam bloqueadas, esperando que a outra libere os dados. Quando isso ocorre, o SGBD deve resolver o *deadlock* revertendo parcialmente ou totalmente uma das transações para permitir a continuidade das outras.

Durante a execução de transações, informações de rápido acesso são armazenadas em buffers de memória compartilhada, como o plano de consulta e o *buffer* de *log*, que pode conter dados ainda não sincronizados com o SGBD. Para garantir que os dados finais não se percam, um processo escritor é responsável por transferir esses dados para o armazenamento estável.

Em caso de falhas, a recuperação das transações é realizada a partir do ponto de execução utilizando o *log* armazenado na memória estável. Além do *log*, a técnica de *checkpoints* é utilizada para marcar no arquivo de log os pontos em que as transações finalizadas foram gravadas no banco. Essa técnica é aplicada por SGBDs que não gravam os dados imediatamente após a finalização de uma transação, utilizando os registros de *log* para recuperar os dados que não foram gravados devido a falhas.

Essas informações foram extraídas da referência (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

#### 2.1.2.2 Servidores de Transação

O servidor de transação é responsável por garantir a conclusão de uma transação, utilizando diversos processos auxiliares para assegurar sua execução de forma eficaz. Na Figura 1 é apresentada uma ilustração que exemplifica a metodologia de execução do serviço, destacando os componentes envolvidos na finalização das transações:

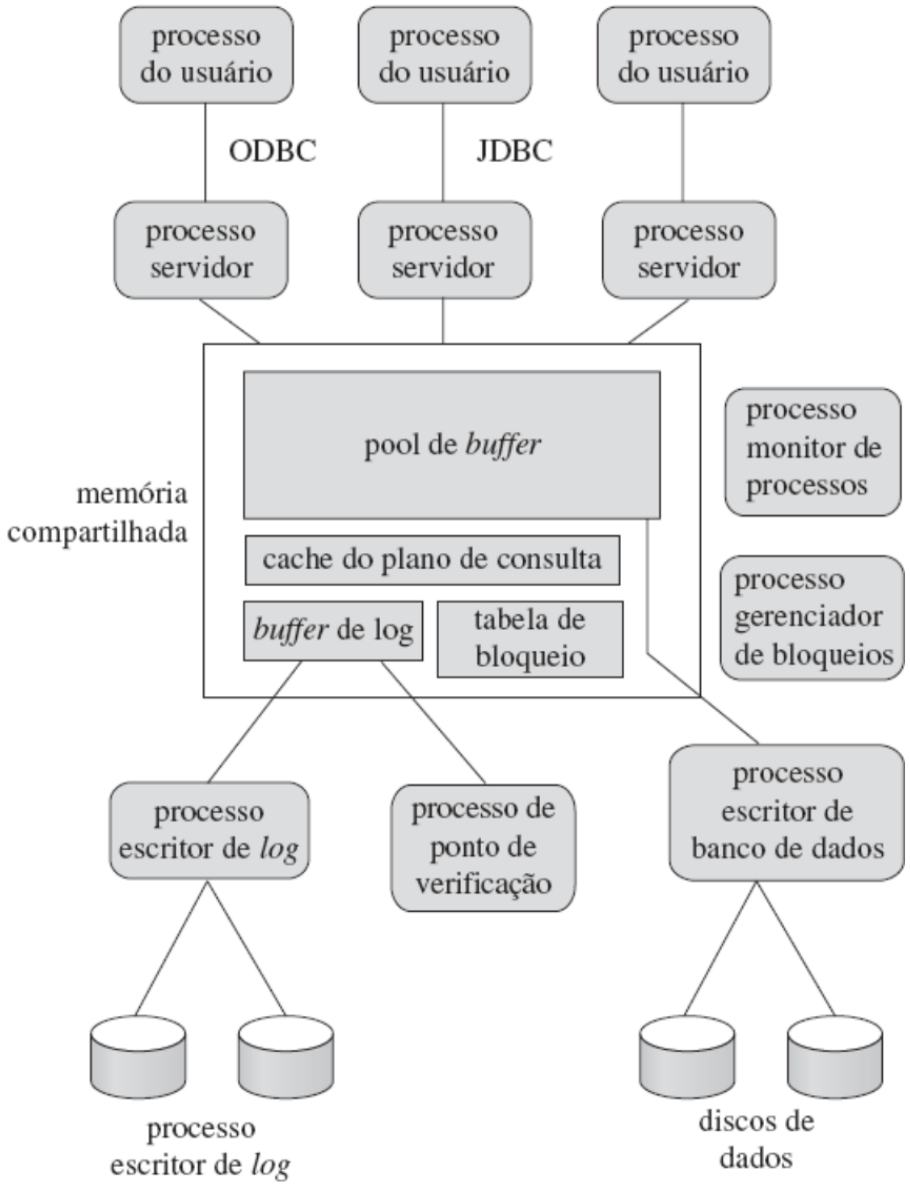
- **Processo gerenciador de bloqueios:** Utilização de bloqueios é uma das técnicas que pode ser implementada no SGBD para garantir o controle de concorrência (acesso multiusuário sobre um conjunto de dados);
- **Processo escritor de banco de dados:** envio de blocos de *buffer* modificados de forma contínua para o disco: envio de registros de log do *buffer* para o armazenamento estável;
- **Processo escrito de log:** envio de registros de log do *buffer* de registro de log para o armazenamento estável;
- **Processo de ponto de verificação:** realização de *checkpoints* periódicos;
- **Processo monitor de processos:** realiza a gestão da execução dos processos, realizando a restauração ou parada;

Na seção da memória compartilhada os seguintes itens são compartilhados:

- **Pool de *buffer*:** fornecimento de páginas de memória de trabalho;
- **Tabela de bloqueio:** tabela que faz a gestão do bloqueio dos processos;
- ***Buffer* de log:** registros em aguardo para serem enviados para o armazenamento estável;
- **Planos de consulta em *cache*:** *cache* das consultas que poderão ser reutilizáveis;

No mecanismo da memória compartilhada, todos os processos podem acessar as informações contidas na memória. Dessa forma, pode haver a ocorrência de mais de um processo

Figura 1 – Exemplo do serviço de transação fazendo o uso da memória compartilhada.




Fonte: (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

acessando o mesmo dado para realizar alguma manutenção. Para garantir que determinado dado seja selecionado por um processo por vez, são utilizados bloqueios, que são funções do sistema operacional que utilizam instruções atômicas testar-e-marcar (*test-and-set*) ou comparar-e-trocar (*compare-and-swap*). Dessa forma, as instruções atômicas são utilizadas para definir uma tranca (bloqueio) no SGBD de curta duração, tendo seu controle realizado mediante uma tabela específica de bloqueios.

A abordagem de uso de semáforos foi proposta inicialmente por E. W. Dijkstra para realizar o controle de sinais de uso futuro, onde o sinal 0 é considerado estado *down* e o valor 1 como *up*. Em sistemas operacionais, semáforos são utilizados basicamente para realizar o controle de concorrência entre processos, onde o mesmo determina quais processos devem

aguardar a execução enquanto os que estão em execução finalizem. Quando o processo possui o estado *down*, ele deve aguardar até que o sistema sinalize que possa prosseguir, tendo sua condição alterada para *up* (WOODHULL, 2008).

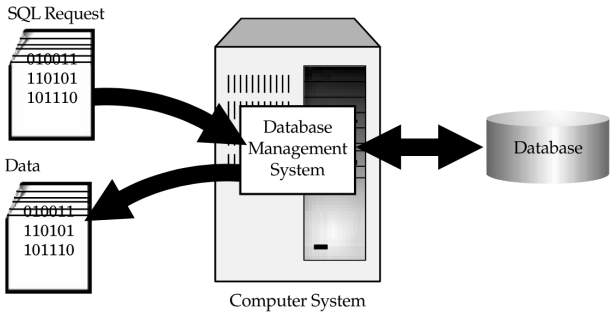
#### 2.1.2.3 Servidores de Dados

Os servidores de dados têm como objetivo central a utilização do seu poder computacional para armazenar informações eventualmente utilizadas para a realização de cálculos, evitando a busca das informações no servidor principal e, então, realizando todos os cálculos necessários. A *Structured Query Language (SQL)* nem sempre é utilizada diretamente, mas sim através do uso de APIs para o retorno dos dados via estruturas como Extensible Markup Language (XML) ou JavaScript Object Notation (JSON) (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

A linguagem SQL foi desenvolvida com o objetivo de ser utilizada exclusivamente para comunicação com SGBDs relacionais, sendo possível acessar e retornar determinadas estruturas de dados utilizando comandos de *select* e também alterar o estado das informações fazendo o uso de comandos de *insert*, *update* e *delete* (WEINBERG; GROFF; OPPEL, 2010).

Conforme exemplificada na imagem Figura 2, ao executar um comando na linguagem SQL o mesmo é enviado para o servidor de dados que se comunica com o SGBD e realiza o retorno dos dados.

Figura 2 – Figura de exemplificação do funcionamento da linguagem SQL.




Fonte: (WEINBERG; GROFF; OPPEL, 2010).

## 2.2 BANCOS DE DADOS PARALELOS E DISTRIBUÍDOS

Com o crescente uso da internet, houve um aumento expressivo do desenvolvimento de aplicações para a World Wide Web (WWW), havendo a necessidade de um ambiente adequado em processamento para as mesmas poderem ser hospedadas e necessitando de bancos de dados robustos, que em alguns casos poderiam atingir petabytes de armazenamento e *data centers* com centenas de processadores com diversos núcleos. Outro fator que impulsionou o uso de estruturas robustas foi o barateamento do custo de desenvolvimento de equipamentos de sistema de informação, que facilitou a aquisição de equipamentos para desenvolvimento de um ambiente

capaz de adequar aplicações de tal poder computacional, permitindo execuções paralelas e distribuídas. Dessa forma, foi necessário o planejamento de estratégias que visassem utilizar da melhor maneira possível o poder computacional contido, diminuindo o tempo de resposta, maximizando a quantidade de processamento e tornando o acesso a dados em bases remotas facilitado pela disponibilidade das informações.

### 2.2.1 Bancos Paralelos

Os bancos paralelos têm como objetivo central a divisão do processamento de informações em diversas máquinas disponíveis na rede, maximizando o poder computacional para garantir a entrega de velocidade adequada para as aplicações.

> "Existem duas medidas principais de desempenho de um sistema de banco de dados: (1) vazão (*throughput*), o número de tarefas que podem ser completadas em determinado intervalo de tempo, e (2) tempo de resposta, a quantidade de tempo necessária para completar uma única tarefa desde o momento que ela foi submetida. Um sistema que processa grande número de pequenas transações pode melhorar o *throughput* processando muitas transações em paralelo. Um sistema que processa grandes transações pode melhorar o tempo de resposta e, também, o *throughput* realizando, em paralelo, sub-tarefas de cada transação."(SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

O principal benefício do uso de um sistema paralelo é o ganho de agilidade através da divisão dos processos com os nós localizados internamente na rede, aproveitando para serem executadas um maior número de transações em menor intervalo de tempo. Em momentos que é discutida a performance da utilização de bancos paralelos, existem duas questões importantes: uma delas é em relação ao ganho de velocidade obtido pela execução de determinadas tarefas em menor tempo e um maior grau de paralelismo. A segunda é denominada ganho de escala, onde é exigido a execução de tarefas com um maior grau de complexidade e paralelismo.

### 2.2.2 Bancos Distribuídos

Sistemas de bancos de dados distribuídos são definidos como sítios localizados em diferentes pontos geograficamente, onde os sistemas que os gerenciam se comunicam tanto via rede privada como pela internet, não fazendo o compartilhamento de recursos computacionais entre as bases. O uso desse tipo de estrutura ocorre em arquiteturas onde cada sítio faz o papel de um nó na rede, que ao mesmo tempo é essencial para o seu funcionamento, e caso ocorra uma falha, o sistema pode operar normalmente realizando a distribuição das cargas com os demais nós. Dessa forma, a principal vantagem da utilização de um banco de dados distribuído é a disponibilização das informações que estão contidas em diferentes sítios, onde os mesmos possuem um alto grau de autonomia para controle dos dados contidos localmente.

Os bancos de dados distribuídos são divididos em duas categorias importantes: homogêneos e heterogêneos. Os bancos distribuídos homogêneos utilizam o mesmo software de

gerenciamento de banco de dados, onde os nós da rede cooperam para a execução das operações internas. Já o banco de dados heterogêneo refere-se a sistemas que executam software de diferentes bancos de dados, onde os sítios podem oferecer baixo grau de cooperação entre os nós da rede.

A comunicação entre sítios de um sistema distribuído ocorre mediante uma *Wide-area Network (WAN)*, onde normalmente a largura de banda pode ser compartilhada por várias aplicações, podendo ocasionar a diminuição da banda. Ao utilizar uma rede WAN para a conexão entre pontos localizados em diferentes locais geográficos, a latência pode ocorrer em determinados momentos, onde as causas podem ser perda de desempenho e enfileiramento de roteadores, ocasionando lentidão na entrega de mensagens para diversos destinos.

As informações contidas nessa subseção foram acessadas pela referência (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

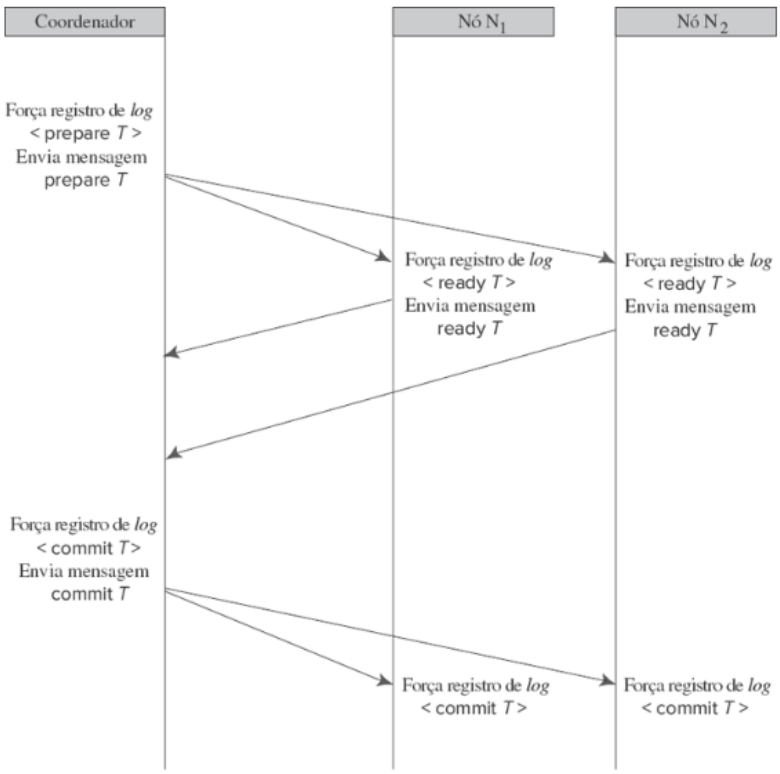
### 2.2.3 Processamento de Transações Distribuídas

O uso de transações em bancos de dados paralelos e distribuídos tem como preocupação principal a manutenção da atomicidade dos dados, onde uma determinada transação pode ser executada tendo como base informações localizadas em diferentes máquinas na rede. Assim, é necessária a utilização de um protocolo que garanta um *commit* unificado de um conjunto de informações que façam parte de uma mesma transação, e o protocolo mais utilizado para esse caso é o 2 Phase Commit (2PC). Para executar esse protocolo, a configuração mais comum é a definição de uma das máquinas de rede participante da transação como coordenador da execução da transação distribuída (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

O protocolo 2PC, conforme demostrado na Figura 3, tem como base a sincronização dos *commits* (validação) das partes (sub transações) de uma transação distribuída. ele gerencia a operação de uma transação com diversos participantes. Na execução de uma determinada transação, os nodos participantes enviam o status da execução ao coordenador, e dependendo da mensagem o mesmo define se deve dar andamento da operação ou cancelar. Após receber o status o coordenador envia uma mensagem de preparação <*prepare* T> para todos os nós, e é devolvida uma mensagem de confirmação <*ready* T> para dar andamento na operação. Caso o coordenador receba uma mensagem de *abort* ou não receba qualquer mensagem dos participantes é realizado o cancelamento da operação. Caso o coordenador tenha recebido todas as confirmações é disparada uma nova mensagem de *commit* para a operação ser finalizada.

Todas as operações descritas acima para o funcionamento do protocolo 2PC são armazenadas em um ou mais arquivos de *log*(mantidos nos diferentes sites de execução da transação distribuída), onde o mesmo é importante para a resolução de possíveis falhas durante a execução da operação tanto dos nodos participantes quando do coordenador. Em situações onde o coordenador falhe toda a operação do protocolo pode ficar comprometida, pois os nodos

Figura 3 – Figura exemplificando o funcionamento do protocolo 2PC.




Fonte: Verificado em (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

secundários não teriam as políticas necessárias para poder agir entre as transações. Para essas situações existem procedimentos que visam mitigar a situação, como, por exemplo, atribui à outra transação a responsabilidade do *commit* final.

As informações contidas nessa subseção foram acessadas pela referência (SILBERSCHATZ; KORTH; SUDARSHAN, 2020).

## 2.3 APLICAÇÕES DISTRIBUÍDAS E REPLICAÇÃO

A replicação é uma das formas de distribuição de dados presente em sistemas distribuídos. Aplicações e sistemas distribuídos são construídos mais recentemente a partir da composição de serviços ou de microsserviços.

### 2.3.1 Replicação

O termo replicação é utilizado para definir tecnologias capazes de realizar a cópia e distribuição de dados entre diferentes SGBDs, mantendo a consistência dos dados. O seu uso se dá principalmente em mitigar possíveis problemas que possam ocorrer entre uma determinada base de dados, criando uma cópia da base de dados para *backup*. Outro fator que favorece a sua utilização é a necessidade do dado estar disponível em mais de uma fonte (EZECHIEL; AGARWAL;

KAUSHIK, 2020).

As duas principais formas de distribuição dos dados são a fragmentação e a replicação. Fragmentar é dividir uma ou mais tabelas (ou conjunto de dados) por linhas ou por colunas, e cada fragmento pode ser colocado em um sítio diferente. Replicar é duplicar, fazer cópias, de uma ou mais tabelas (ou conjunto de dados) (EZECHIEL; AGARWAL; KAUSHIK, 2020).

Atualmente o recurso de replicação de dados tem se tornado um mecanismo essencial de uso nos bancos de dados distribuídos, havendo a necessidade crescente da informação estar disponível em mais de uma fonte por motivos estruturais da aplicação ou como *backup*. Há dois tipos de replicação, síncrona e assíncrona, onde a operação varia conforme a sua utilização (MOIZ *et al.*, 2011).

> "A replicação é um conjunto de tecnologias para copiar e distribuir dados e objetos de banco de dados de um banco de dados para outro e depois sincronizar entre bancos de dados para manter a consistência. [...] A sincronização de dados é o processo que visa estabelecer a consistência entre os dados de uma fonte para um destino e vice-versa, e realizar a contínua harmonização dos dados ao longo do tempo. Tecnologias de sincronização são projetadas para manter um único conjunto de dados entre dois ou mais dispositivos, copiando automaticamente as alterações entre as bases"(Traduzido via plataforma Google Tradutor)(EZECHIEL; AGARWAL; KAUSHIK, 2020)

Com a execução da operação de replicação existe a ocorrência de duplicidade da informação, assim necessitando o uso de uma operação atômica controlada através de uma transação. Caso houver a necessidade de replicação para mais de um sítio, a mesma deve ser distribuída.

Os dois tipos de replicação visam a sincronização das informações entre as bases de dados, onde cada um dos tipos possui características únicas que fazem o seu uso necessário e específico. (MOIZ *et al.*, 2011).

### 2.3.2 Replicação síncrona

A replicação síncrona realiza a replicação dos dados de forma que todos os nodos que fazem parte da rede recebam a informação antes da realização do fechamento da transação, assim, obrigando que todos os participantes da rede estejam disponíveis no momento da execução da operação.

Podem ser identificados dois tipos de replicação dentro do contexto síncrono:

- ***Read-Any* e *Write-All*:** operação de leitura prevalece contra a de escrita, onde ambas devem ser aptas em todos os nodos que fazem parte da rede;
- ***Voting*:** operação garante o acesso e atualização do dado denominado mais recente na base;

As informações sobre a replicação síncrona foram acessados através da referência (EZECHIEL; AGARWAL; KAUSHIK, 2020).

### 2.3.3 Replicação assíncrona

A replicação assíncrona possui a característica de não haver a necessidade dos participantes da rede estarem disponíveis no momento da execução da operação. Dessa forma, os dados são organizados em uma estrutura específica para serem replicados aos destinos quando eles estivem disponíveis para receber as informações.

Dentro da estrutura assíncrona existem dois tipos de contextos:

- ***Primary Site*:** a replicação das informações sempre terão origem do nodo denominado como principal, onde as réplicas somente realizarão a leitura dos dados replicados;
- ***Peer-to-Peer*:** as diferentes réplicas podem ser definidas como geradoras da replicação, sendo necessário a utilização de um protocolo de controle de *deadlocks* para garantir a consistência das informações;

As informações sobre a replicação assíncrona foram acessados através da referência (EZECHIEL; AGARWAL; KAUSHIK, 2020).

## 2.4 CONSTRUÇÃO DE APLICAÇÕES DISTRIBUÍDAS

Aplicações distribuídas são processos que são executados em diversas máquinas com uma comunicação via internet, com objetivo de distribuir o processamento de determinados serviços. Dessa forma, uma aplicação poderá fazer o uso de diferentes tipos de serviços escalando a execução.

Há diferentes tipos de arquiteturas para construção de sistemas distribuídos com formas diversas de abstração na estrutura, como arquitetura de 2 camadas (cliente-servidor), arquitetura de 3 camadas (dados, negócios e interface), integração de componentes, integração de objetos, integração de serviços, entre outras. Nesse trabalho o foco será em uma arquitetura em três camadas.

> "MVC é um padrão de design de software construído em torno da interconexão de três tipos principais de componentes: Modelo, Visualização e Controlador, muitas vezes com um forte foco em paradigmas de software de Programação Orientada a Objetos (OOP). MVC é uma estrutura para construção de aplicações web usando um design MVC. É a arquitetura de desenvolvimento de software mais importante hoje em dia. Essa arquitetura gerenciava automaticamente o código e ajudava o programador a desenvolver aplicações Web bem gerenciadas."(Traduzido via plataforma Google Tradutor)(THAKUR; PANDEY, 2019)

A camada de apresentação será composta pelos componentes de interação com o usuário, havendo o contato com páginas e seções para requisição de informações. Nessa camada serão desenvolvidas regras específicas que dizem respeito ao cumprimento em informar determinados valores e seus formatos, para então serem utilizadas como base para novas requisições de informações. Com a execução de requisições por parte da página de apresentação, serão acessados *endpoints* que estarão disponíveis na camada de lógica do negócio. Esses *endpoints* estarão expostos por classes controladoras que farão todas as validações necessárias para então retornar os dados solicitados. Após a requisição ter todos os dados validados pela camada de negócio, serão acessados procedimentos de acesso ao SGBD, executando consultas SQL para seleção ou manutenção de determinadas estruturas.

### 2.4.1 Middleware

Ao fazer o uso de diferentes bases de dados distribuídas, a disponibilidade do ambiente é um tema fundamental para manter a execução de processos e tarefas em um sistema interligado. A utilização de *middlewares* agrega facilidades no controle e execução de processos, auxiliando na necessidade de execuções intermediárias entre cliente e servidor.

> "*Middleware* é um produto de software que atua como um intermediário entre aplicativos e fornece serviços para aplicativos de software e o sistema operacional. Ele serve como a cola de software que permite que sistemas distintos trabalhem juntos."(Traduzido via plataforma Google Tradutor)(VERMA, 2022)

No desenvolvimento de um *middleware*, é necessário definir uma arquitetura para embasar o desenvolvimento dos processos que venham a definir a sua utilização. Em processos que necessitam uma alta disponibilidade de conexão, é comum encontrar mecanismos que implementam a tolerância a falhas, onde o *middleware* é desenvolvido para prover um alto grau de resiliência quando se trata em manter a conexão ativa. Caso contrário, uma pequena falha de conexão pode levar a uma parada repentina de um processo importante dentro do sistema. (LI *et al.*, 2015)

# 3 REPLICAÇÃO NOS SGBDS

Na presente seção serão apresentadas algumas das formas como os diferentes SGBDs fazem o uso da replicação em sua estrutura, com o objetivo de comparar o seu funcionamento e processos. O conceito de replicação pode ser verificado em diferentes produtos de banco de dados, onde, dependendo do produto, são identificadas metodologias e processos distintos, mas a utilização se mantém equivalente: a replicação das informações contidas em sistemas distribuídos.

Atualmente, com a crescente demanda pelo uso de sistemas distribuídos no ambiente corporativo, as ferramentas de replicação se tornaram parte do produto do banco de dados. O uso delas garante um maior nível de segurança e disponibilidade para eventuais desastres que possam vir a ocorrer, além de manter um *backup* dos dados (MOIZ *et al.*, 2011).

Serão apresentadas a seguir as principais características de gerenciamento da replicação dos SGBDs Informix, MySQL e PostgreSQL.

## 3.1 IBM INFORMIX ENTERPRISE REPLICATION

*Enterprise Replication (ER)* é uma ferramenta de replicação de informações entre bancos de dados distribuídos Informix, de propriedade exclusiva da empresa International Business Machine Corporation (IBM), a qual utiliza métodos assíncronos para controle da replicação das informações entre os nodos da rede. Sua operação consiste na gestão da replicação de informações em modo assíncrono entre um nodo principal e os secundários vinculados a ele. A replicação ocorre via captura de transações pela leitura do *logical log* do banco de dados para, então, realizar a sincronização com base na definição das regras (IBM, 2022).

### 3.1.1 Modo de operação do sistema de replicação

A ferramenta ER tem como base principal de funcionamento a replicação dos dados via modo assíncrono, não sendo necessário que todos os participantes secundários que compõem a rede estejam visíveis no momento da execução da operação. Esse tipo de abordagem de replicação garante um maior nível de resiliência por não depender da disponibilidade dos participantes. Na ocorrência de algum nodo estar indisponível, no momento em que a conexão for restabelecida, as informações são sincronizadas com base na captura do *logical log*.

> "Para manter um histórico de transações e alterações no servidor de banco de dados desde o último backup do espaço de armazenamento, o servidor de banco de dados gera registros de log. O servidor de banco de dados armazena os registros de log no log lógico, um arquivo circular composto por três ou mais arquivos de log lógico. O log é chamado de lógico porque os registros de log

> representam operações lógicas do servidor de banco de dados, em oposição às operações físicas. A qualquer momento, a combinação de um backup de espaço de armazenamento com um backup de log lógico contém uma cópia completa dos dados do servidor de banco de dados."(Traduzido via plataforma Google Tradutor) (IBM, 2023).

O uso do *logical log* para realizar a gestão da replicação, ao invés do uso das transações síncronas, traz vantagens, por utilizar um mecanismo que está disponível no próprio banco de dados. Dessa forma, não ocorre a sobrecarga do sistema, que não precisa utilizar os recursos consumidos por outros processos.

Por haver a possibilidade de falhas de conexão com os nodos de destino da replicação das informações, a tecnologia garante a atualização dos dados em qualquer ponto capturado pelo *logical log*. No momento de uma ocorrência de falha, as operações pendentes são adicionadas em uma fila, para que, quando a conexão for retomada, os dados sejam replicados a partir do ponto de parada, garantindo dessa forma a integridade das informações e das transações.

Com a necessidade de realizar a replicação das informações que estavam em fila, podem ocorrer colisões de acesso a dados, caso uma mesma linha precise ser alterada por duas transações diferentes. Para resolver o impasse, é necessário informar ao ER qual deve ser a ação a ser adotada conforme o tipo de conflito. A resolução é informada no momento do cadastro da regra de replicação pelo desenvolvedor ou responsável pelo grupo de trabalho. O Quadro 1 exemplifica os tipos de ocorrências de conflitos e a aplicação das regras de resolução.

Quadro 1 – Regras de resolução dos conflitos entre transações

| **Tipo de ocorrência de conflitos** | **Aplicação das regras de resolução** |
|---|---|
| Ignorar | Não atendimento de qualquer regra. |
| Time stamp | A transação com o *time stamp* mais recente é replicado. |
| Rotinas de Stored Procedure Language(SPL) | Aplica as rotinas definidas via SPL para determinar o que será feito em caso de conflito. |
| Time stamp com SPL | Na ocorrência de *time stamp* é invocado a rotina definida via SPL. |
| Exclusão dos vencedores | Operações de *DELETE* e *INSERT* possuem maior grau de prioridade que *UPDATE*, caso contrário o registro com *time stamp* mais recente prevalece. |
| Sempre aplica | Não realiza o tratamento de conflitos. |

Fonte: (IBM Corporation, 2010).

No processo de replicação, podem ocorrer diferenças de valores, tornando necessária a execução de uma sincronização completa em uma tabela específica ou na estrutura que participa da replicação, com o objetivo de deixar as bases equivalentes. Esse tipo de operação pode ser executado utilizando comandos de reparo, que devem ser executados pelos DBAs.

A ferramenta possibilita a criação de uma estrutura dinâmica de atualização, onde pode-se escolher como os participantes deverão se comportar. Esse tipo de estado pode ser definido como *one-to-many*, onde o nodo primário realiza a replicação para diversos nodos secundários da rede, que apenas farão a leitura das informações. Também existe a composição *many-to-one*,

onde diversos nodos primários enviam a informação para um único ponto, centralizando o envio dos dados. Com a possibilidade de criar uma rede de replicação dinâmica, é possível desenvolver uma topologia de rede que pode influenciar significativamente o método de replicação escolhido.

### 3.1.2 Gestão das Regras de Replicação

Uma determinada regra de replicação é denominada *replicate*, onde é definido como a replicação deve ocorrer, os participantes que farão parte da replicação, a frequência da replicação e como lidar com conflitos, caso ocorram.

Para realizar qualquer operação no sistema do ER, é necessário utilizar os comandos do *Command-line Utility (CLU)* via *prompt* UNIX ou sistema operacional Windows conectado diretamente a um servidor ER. Ao utilizar o comando *cdr*, ele segue um padrão de ordem de execução:

1. **Comando**: comando e suas variações;
2. **Opções**: opções a serem executadas com o comando, onde sempre serão iniciadas com um simples sinal negativo(-) ou dum duplo sinal negativo(–);
3. **Destino**: bases de dados alvo da execução;
4. **Demais objetos**: demais objetos que serão afetados;

Com base nas regras de ordem de execução dos comandos, no Quadro 2 são descritos possíveis comandos que fazem a manutenção de estruturas de replicação.

Quadro 2 – Algumas especificações de comandos fazendo o uso do comando *cdr*

| Comando | Especificação |
|---|---|
| *cdr define replicate* | comando com objetivo de definir uma determinada regra de replicação. |
| *cdr modify replicate* | comando com objetivo de definir uma determinada regra de replicação |
| *cdr check replicate* | comando com objetivo de checar a replicação entre determinados participantes |

Fonte: (IBM Corporation, 2010)

Além dos comandos apresentados na seção anterior existem diversos outros que auxiliam na gestão e manutenção da estrutura de replicação. Como verificado nos exemplos anteriores, cada comando possui sua sintaxe específica para execução, e por consequência, responsabilidades atribuídas.

Para realizar a configuração de parâmetros e variáveis de ambiente existe o comando *onconfig*, onde o ambiente de execução do ER funcionará com base das informações fornecidas. No Quadro 3 são exemplificados alguns comandos possíveis de serem executados utilizando o *onconfig* para definir parâmetros para a operação do ambiente.

Quadro 3 – Algumas especificações de comandos fazendo o uso de *onconfig*

| Commando | Sintaxe | Aplicação das regras de resolução |
|---|---|---|
| onconfig.std value | *CDR_APPLY min_threads max_threads* | Especifica o número mínimo e máximo de threads |
| onconfig.std value | *units dbs_validos* | Especifica o espaço do banco de dados para criação |
| onconfig.std value | *units segundos* | Especifica o tempo em segundos para o componente aguarde até o bloqueio da base de dados se finalizada |

Fonte: (IBM Corporation, 2010)

### 3.1.3 Replicação dos dados

Antes que a replicação possa ser utilizada, é importante definir qual será o nodo raiz da replicação, que ficará no servidor central, e as demais máquinas que farão parte da arquitetura da rede, que serão os nodos folhas. Após a definição dos participantes, a replicação pode ser dividida em três fases importantes: captura dos dados, transporte e execução da replicação dos dados (IBM Corporation, 2010).

Conforme disposta na Figura 4 a replicação de um determinado dado deve ocorrer primeiramente em uma máquina que faz parte da arquitetura. Após isso, a informação é colocada no *logical log*. Existe um componente responsável por monitorar o *logical log*, observando se existem novos registros para serem processados, denominado *snoopy*. Após identificar os registros, eles são agrupados para serem adicionados a uma fila de execução, e então transmitidos para as demais máquinas na rede. Caso a máquina tenha a conexão disponível, o registro da fila é transmitido para uma nova fila de recebimento no sistema ER destino. Após o processo de recebimento, o registro é sincronizado com a base de destino, e caso haja algum conflito, ele é resolvido instantaneamente, conforme as regras previamente definidas. Por último, o resultado da execução do registro é adicionado a uma fila de confirmação e transmitido ao remetente para ser armazenado em um *spool* de execuções.

Para auxiliar no monitoramento das estruturas, o Informix disponibiliza comandos que podem ser executados diretamente na linha de comando, sendo um dos mais utilizados o *onstat*. Esse comando pode ser executado com acréscimos para monitorar seções específicas, como o estado de threads, catálogos, estatísticas de estruturas de dados específicas e de rede.

Para a manutenção dos métodos de replicação dos dados, são utilizadas tabelas próprias do Informix. Cada tabela é responsável pela gestão de uma determinada estrutura de dados que faz parte da replicação. A seguir, são apresentados alguns exemplos de tabelas que são utilizadas para esse propósito:

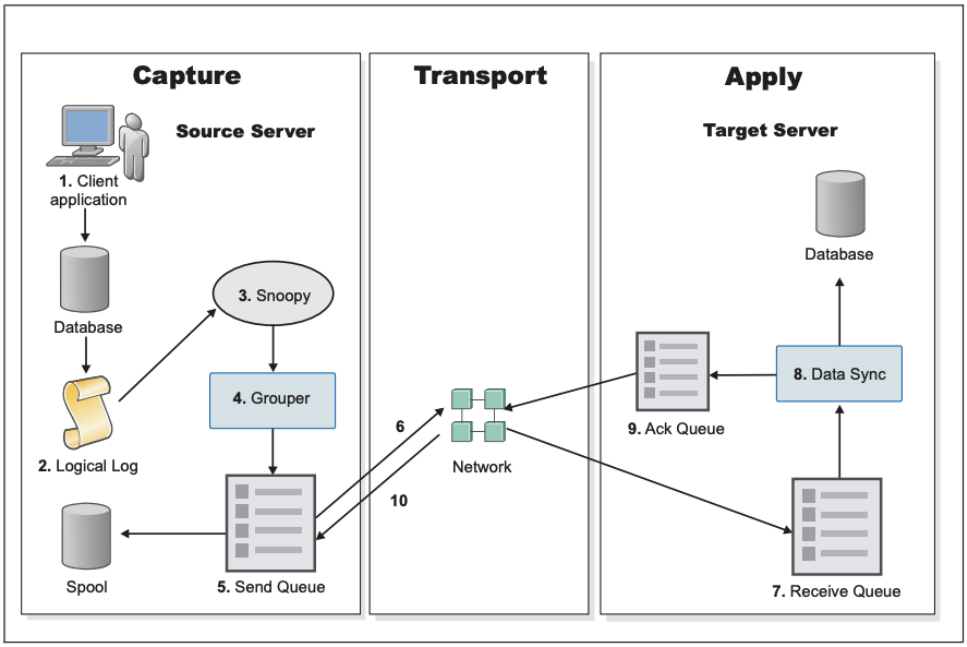
As informações mencionadas na subseção foram extraídas no manual do banco de dados Informix ER (IBM Corporation, 2010).

Quadro 4 – Exemplo de tabelas de gestão do ER

| Tabela | Especificação |
|---|---|
| *replcheck_stat* | informações sobre a checagem da consistência dos dados. |
| *replcheck_stat_node* | informações sobre a checagem da consistência dos dados de um determinado servidor. |
| *syscdr_ddr* | informações sobre o status da captura dos dados ou dos bloqueios de transações. |
| *syscdr_nif* | informações sobre a conexão de dados entre os servidores que fazem parte da replicação. |

Fonte: (IBM Corporation, 2010)

Figura 4 – Exemplo do sistema de replicação do ER.




Fonte: (IBM Corporation, 2010).

## 3.2 MYSQL

A versão mais recente do MySQL, 8.3, disponibiliza ferramentas de replicação entre bancos de dados, com suporte para replicação síncrona e assíncrona. Assim como no ER via banco de dados Informix, no MySQL não é necessário que todas as máquinas estejam operantes na rede de replicação para que uma operação seja efetivada no sistema.

Para a operação do banco de dados, é necessário ter um nodo principal e réplicas configuradas. A versão 8.3 introduz o método de replicação *Global Transaction Identifiers (GTID)*, que permite identificar de forma única uma operação específica e realizar a replicação das informações diretamente para uma réplica. Com o uso do GTID, não é mais necessário depender dos arquivos de *logs*, já que o controle da replicação é realizado diretamente pela ação da operação.

Além disso, a ferramenta de replicação do MySQL oferece a possibilidade de combinar

métodos de sincronização, permitindo mesclar replicação síncrona e assíncrona. O sistema monitora se pelo menos uma réplica reconhece a transação antes de finalizá-la nos outros nodos da rede.

O MySQL também disponibiliza três tipos principais de replicação, que tratam da seleção e processamento dos registros replicados para as réplicas:

- **Statement Based Replication (SBR)**: Neste tipo, todos os SQLs executados no nodo principal são replicados para as réplicas, sem aplicar filtros. Ou seja, qualquer comando executado é replicado.
- **Row Based Replication (RBR)**: Apenas as linhas que sofreram modificações são replicadas, o que diminui o tráfego de dados desnecessários e melhora a eficiência da replicação.
- **Mixed Base Replication (MBR)**: O tipo padrão é o SBR, mas dependendo da operação e da arquitetura, o sistema pode alternar para RBR. O MBR oferece flexibilidade, permitindo ao sistema selecionar o tipo de replicação mais adequado de acordo com a situação.

Essas abordagens tornam a replicação no MySQL altamente flexível, permitindo diferentes estratégias de replicação conforme o caso.

As informações mencionadas nesta seção foram obtidas na documentação oficial do MySQL (ORACLE, 2024a), (ORACLE, 2024b) e (ORACLE, 2024c).

## 3.3 POSTGRESQL

A replicação lógica utiliza estruturas para garantir o envio de dados entre os nodos da rede, empregando os termos "publicadores"e "assinantes"para definir os participantes. Um publicador pode ser entendido como o nodo responsável por enviar as informações, enquanto os assinantes atuam como receptores. A replicação das informações sempre ocorre por meio de um novo publicador, que, a partir de um *snapshot* de alteração, copia os dados para os demais nodos em tempo real, garantindo a ordem de execução das operações realizadas pelo nodo principal, todas identificadas por uma única assinatura. Esse processo é conhecido como replicação transacional.

Assim como o publicador envia dados para os assinantes, a mesma metodologia pode ser aplicada caso um assinante precise atuar como publicador. No entanto, esse processo deve ser tratado com cautela, pois, à medida que mais nodos se tornam responsáveis por realizar a replicação, podem surgir conflitos devido a múltiplas assinaturas.

Em relação à ocorrência de impasses (*deadlocks*) entre os procedimentos, o banco de dados oferece algumas opções de resolução. Uma delas é a intervenção do administrador, que

define qual a melhor solução para o problema. Outra alternativa é permitir que o próprio assinante resolva os conflitos, utilizando regras pré-definidas, sem a necessidade de análise por parte do administrador.

As informações mencionadas nesta seção foram obtidas na documentação oficial do PostgreSQL (POSTGRESQL, 2024b), (POSTGRESQL, 2024a) e (POSTGRESQL, 2024c).

## 3.4 COMPARATIVO SGBDS

Visando o comparativo dos SGBDs e a ampla visibilidade de suas características, segue abaixo quadro contendo as informações relacionadas aos aspectos e os seus respectivos banco de dados:

Quadro 5 – Quadro comparativo das características dos SGBDs

| **Aspecto** | **Informix ER** | **MySQL** | **PostgreSQL** |
|---|---|---|---|
| Versão | 14.10 | 8.3 | 16.4 |
| Tipo de Replicação | Unidirecional e Bidirecional | SBR, RBR e MBR | Lógica e Física |
| Modo de Replicação | Assíncrono e Assíncrono | Síncrono e Assíncrono | Síncrono e Assíncrono |
| Suporte Técnico | Suporte Oficial | Suporte Oficial | Empresas especializadas e comunidade |
| Resolução de Conflitos | Resolução automática ou manual | Resolução automática ou manual | Resolução manual ao atribuição aos assinantes |
| Replicação | Transação | *Statements* | WAL |

Fonte: (IBM Corporation, 2010), (ORACLE, 2024a), (POSTGRESQL, 2024b)

# 4 PROPOSTA DO PROTÓTIPO

Atualmente, o ambiente corporativo em questão é de uma empresa de grande porte que gerencia estruturas de TI em âmbito internacional. A empresa utiliza um ERP que mantém os dados em um SGBD Informix e possui uma equipe de TI interna, dividida entre as áreas de infraestrutura e desenvolvimento. Essa equipe é responsável por desenvolver e manter a operação de diversas unidades que fazem parte do grupo. Todos os servidores hospedam exclusivamente bancos de dados Informix na versão 14.10, e em alguns casos, a informação contida no destino precisa ser utilizada em outro local ou até mesmo centralizada. Com a proposta apresentada neste trabalho, os usuários, que serão os DBAs responsáveis pelas estruturas de replicação, poderão realizar a manutenção e a criação de regras de replicação, tanto na origem quanto no destino, acompanhando todo o processo diretamente por meio de um protótipo.

A utilização de uma ferramenta de replicação nativa de um SGBD agrega valor ao sistema, eliminando a necessidade de tecnologias externas, que poderiam gerar um alto grau de complexidade. Embora a replicação ofereça facilidade de uso, será necessário realizar tarefas de monitoramento e cadastro de novas regras de replicação, o que frequentemente gera um gargalo para a equipe de DBAs. Esse gargalo poderia ser atribuído a um responsável pelo subsistema que necessita da manutenção, permitindo que a equipe de DBAs foque em tarefas mais complexas.

Este trabalho propõe o desenvolvimento de um protótipo de ferramenta para o gerenciamento de replicação utilizando a ferramenta ER do Informix. Essa abordagem garante o uso do *logical log* para assegurar a recuperação em caso de falhas na comunicação da replicação, além de possibilitar a comunicação assíncrona via banco de dados Informix. A estrutura resultante do trabalho proposto caracteriza-se como um sistema distribuído homogêneo. O objetivo do protótipo é fornecer uma solução complementar para auxiliar o monitoramento e a manutenção das regras de replicação pela equipe de DBAs. O desenvolvimento será realizado com base nas melhores práticas do mercado, utilizando tecnologias e linguagens que agreguem valor ao produto e garantam um desenvolvimento seguro e eficiente.

## 4.1 SITUAÇÃO ATUAL

O sistema de replicação do ambiente corporativo passou por diversas melhorias. Durante um longo período, foram utilizados métodos de controle que não eram apropriados para a funcionalidade de replicação. Nesse período, as ferramentas do ER não eram utilizadas; em vez disso, uma aplicação no servidor gerenciava a replicação através de uma fila, armazenada internamente em tabelas do banco de dados. Esse procedimento foi progressivamente descontinuado à medida que os novos métodos de replicação do ER foram introduzidos no sistema.

Atualmente, o sistema possui conexão com mais de 40 servidores físicos distribuídos por

vários continentes e países. Apenas os servidores que hospedam as unidades operacionais do varejo no Brasil ainda não utilizam o novo sistema de replicação ER.

A proposta deste trabalho visa auxiliar a equipe de DBAs por meio de uma ferramenta de monitoramento das estruturas de replicação atuais, facilitando o acompanhamento das informações de maneira centralizada. O protótipo também servirá como uma ferramenta de verificação das regras de replicação, permitindo que outros grupos de trabalho possam conferir e monitorar possíveis divergências nas estruturas.

## 4.2 ARQUITETURA

Para garantir uma melhor separação de responsabilidades, o protótipo será desenvolvido utilizando três camadas distintas: apresentação, regras de negócio e manipulação de dados. Essa organização segue o padrão de desenvolvimento conhecido como Model-View-Controller (MVC), que agrega valor à aplicação ao permitir a separação dos processos, proporcionando maior flexibilidade e auxiliando na agilidade das manutenções que possam surgir no futuro (THAKUR; PANDEY, 2019).

Os DBAs responsáveis terão acesso à aplicação por meio de um portal interno, o mesmo utilizado para acessar outras aplicações web corporativas, como agendamento de exames médicos, consultas ao RH e outras ferramentas. Ao acessar o protótipo, a tela inicial apresentará um panorama geral das estruturas de replicação existentes, oferecendo uma visão abrangente do ambiente. Além da visualização, a aplicação permitirá realizar a manutenção das regras de replicação já configuradas e a criação de novas regras. A arquitetura dessa estrutura é apresentada na Figura 5.

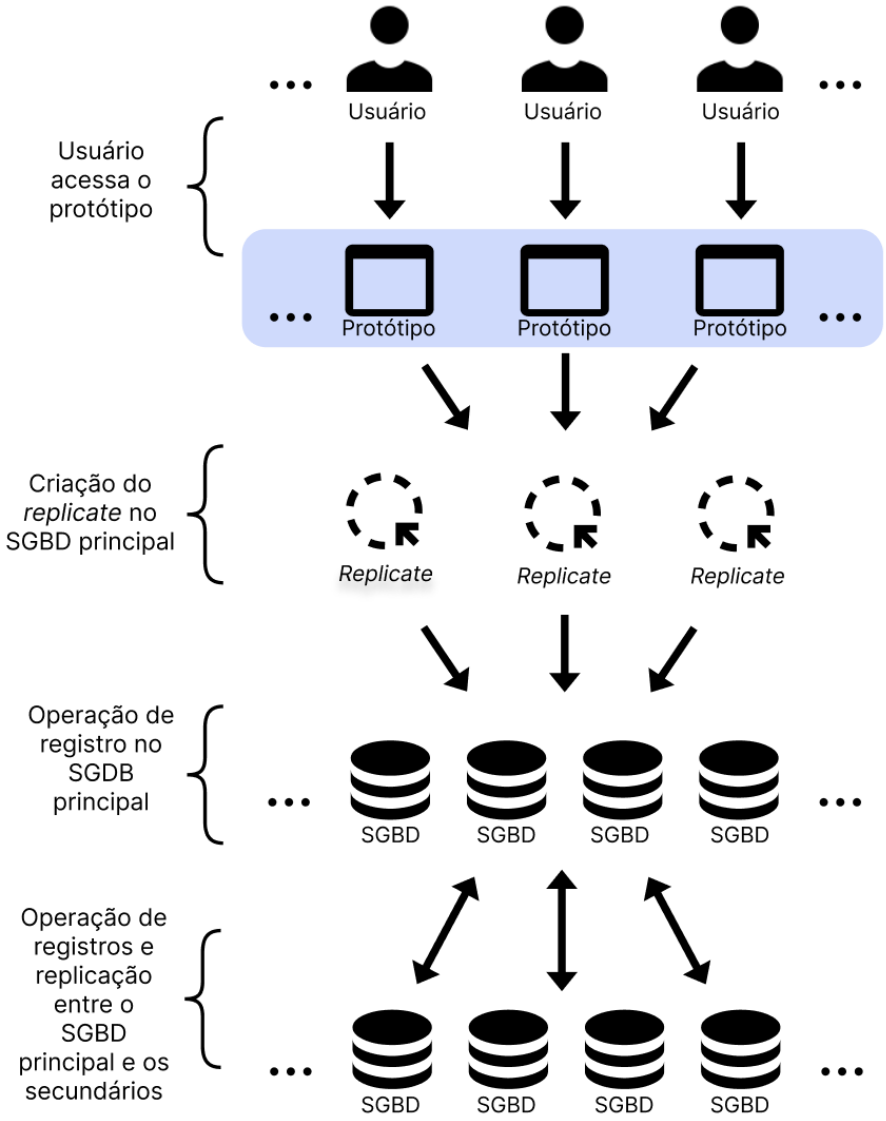
### 4.2.1 Backend

O protótipo terá uma camada de persistência ao banco de dados, a qual será responsável por toda a comunicação direta com a base de dados. Dessa forma, a comunicação ocorrerá via API do tipo *Representation State Transfer (REST)*, onde haverão *endpoints* que farão a manutenção e retorno das regras de replicação e estruturas relacionadas (EHSAN *et al.*, 2024).

> "APIs RESTful, amplamente conhecidas como APIs WEB, consistem em *endpoints*. Cada ponto final é uma funcionalidade concreta implementada de um processo de negócios. Essas APIs geralmente são acessíveis por *Hypertext Transfer Protocol (HTTP)*, incluindo verbos padrões definidos como *GET*, *POST*, *PUT* e *DELETE*. APIs RESTful são invocadas com a ajuda de um endereço, isso é conhecido como *Uniform Resource Identifier (URI)*."(Traduzido via plataforma Google Tradutor) (EHSAN *et al.*, 2024).

O desenvolvimento da API será realizado na linguagem C#, pois a mesma proporciona um ambiente moderno para desenvolvimento, tendo a orientação de objetos e a forte tipagem

Figura 5 – Utilização do protótipo.




Fonte: Elaborado pelo autor

como vantagens na escolha de sua utilização. Outra vantagem que pode ser citada no uso da tecnologia é o desenvolvimento no ecossistema .NET, que oferece um sistema de execução virtual chamado de *Common Language Runtime (CLR)*, uma implementação da própria Microsoft que utiliza um padrão de ferramentas denominado *Common Language Infrastructure (CLI)*, o qual é a base para o desenvolvimento de aplicações e é considerado um padrão internacional (MICROSOFT, 2024).

### 4.2.2 Frontend

Para realizar a interação do usuário com o protótipo, haverá uma interface WEB, permitindo ao usuário visualizar as informações de *status* dos serviços de replicação, as estruturas de replicação que foram criadas e os dados vinculados ao sistema. Para o desenvolvimento do

protótipo, será utilizada a biblioteca *open-source* JavaScript React com o *framework Next.js* (O'REILLY MEDIA INC., 2024).

A biblioteca React foi desenvolvida por colaboradores internos da empresa Meta, com o objetivo principal de solucionar desafios no desenvolvimento de aplicações WEB.

> "React é uma biblioteca para ajudar os desenvolvedores a construir User Interface (UI) como uma árvore de pequenos pedaços chamados componentes. Um componente é uma mistura de HyperText Markup Language (HTML) e JavaScript que captura toda a lógica necessária para exibir uma pequena seção de uma UI maior. Cada um desses componentes pode ser construído em partes sucessivamente complexas de um aplicativo."(O'REILLY MEDIA INC., 2024).

O desenvolvimento com React torna-se rápido e prático devido à funcionalidade de auto renderização do código alterado, que reflete diretamente no *layout* da aplicação. Assim, não é necessário realizar a compilação completa do código-fonte para visualizar as alterações no resultado.

O *framework* Next.js auxilia nos processos de configuração do ambiente React, facilitando o desenvolvimento da aplicação como um todo. Ele permite desenvolver de maneira mais ágil e flexível funcionalidades do *frontend*, como o roteamento de páginas da aplicação e o acesso a *endpoints*. Além disso, o uso de métodos de *async/await* simplifica requisições para componentes do lado do servidor, otimizando o processo de desenvolvimento (VERCEL, 2024).

A escolha da biblioteca React para o *frontend* foi baseada em fatores determinantes. Um deles é seu uso específico para páginas no ambiente WEB, o que está alinhado com a proposta do desenvolvimento da aplicação. Outro ponto importante é o conhecimento prévio da equipe sobre sua utilização em projetos internos do ambiente corporativo, o que reduz o tempo de desenvolvimento e minimiza a curva de aprendizagem.

As informações sobre a biblioteca React foram obtidas diretamente da página oficial do *framework* Next.js (VERCEL, 2024).

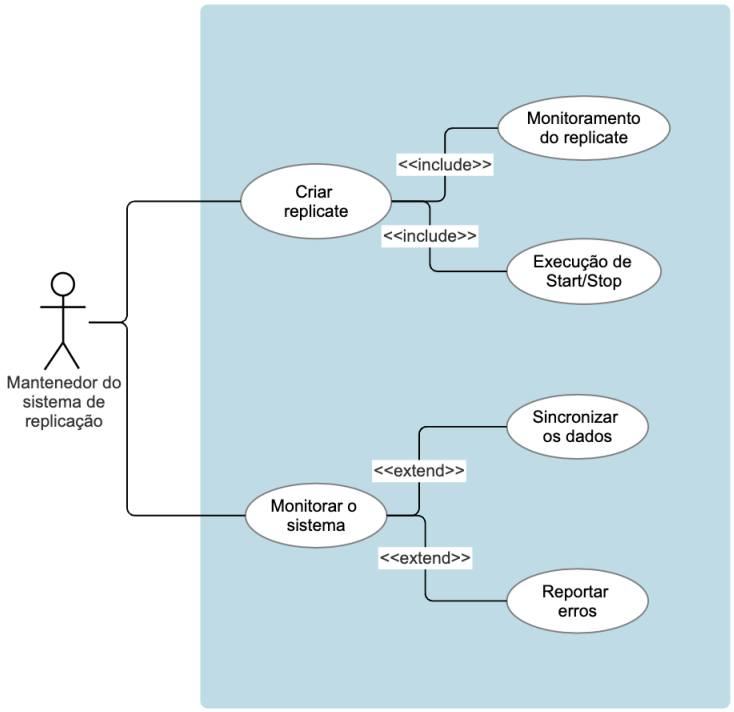
## 4.3 DIAGRAMA DE CASO DE USO

O usuário poderá interagir com o protótipo para criar novas estruturas de replicação ou visualizar as estruturas atualmente existentes. No diagrama da Figura 6, são exemplificados os casos de uso que serão implementados. Os demais requisitos que não estão diretamente relacionados à replicação serão atendidos pelas funcionalidades já existentes no ERP, ao qual o protótipo será integrado.

## 4.4 REQUISITOS FUNCIONAIS

Visando um melhor funcionamento do protótipo, é necessária a implementação de requisitos específicos que permitam a elaboração da estrutura e a gestão da replicação. Esses

Figura 6 – Exemplificação do caso de uso do mantenedor do sistema




Fonte: Elaborado pelo autor.

requisitos são:

- **RF01:** Permitir o login do usuário no sistema.
- **RF02:** Permitir que o usuário se autentique para acessar o sistema.
- **RF03:** Permitir o monitoramento do recurso de replicação em tempo real.
- **RF04:** Implementar resolução de conflitos, caso necessário.
- **RF05:** Desenvolver a aplicação visando o maior grau de segurança.
- **RF06:** O responsável poderá criar ou excluir estruturas de replicação.
- **RF07:** O responsável poderá realizar o *start* ou o *stop* das estruturas de replicação.
- **RF08:** O usuário poderá modificar alguma estrutura de replicação existente.
- **RF09:** O sistema deverá ser acessado a partir de qualquer dispositivo com acesso à internet em ambiente WEB.
- **RF10:** O sistema deverá realizar o controle de uma estrutura de *logs* para monitoramento.
- **RF10:** O sistema deverá realizar a gestão de auditoria da execução dos processos de manutenção das informações.

## 4.5 REQUISITOS NÃO FUNCIONAIS

Requisitos não funcionais são definidos como necessidades que visam a melhora da usabilidade do usuário. Entre eles, podem ser listados os seguintes requisitos:

- **RNF01:** O protótipo deverá ter uso intuitivo.
- **RNF02:** O protótipo deverá ser responsivo para diferentes dispositivos.
- **RNF03:** O protótipo deverá ter uma ótima resposta de retorno ao acessar as opções.
- **RNF04:** Assegurar que o protótipo seja capaz de lidar com uma grande quantidade de dados.
- **RNF05:** Garantir o desenvolvimento de um código de fácil manutenção e entendimento, diminuindo o impacto para o usuário e melhorando na manutenção.

## 4.6 DIAGRAMA DE FLUXO

Na figura Figura 7 é exemplificado um diagrama de fluxo onde mostra as possíveis opções que o usuário poderá ser direcionado na utilização do protótipo.

## 4.7 ESTRUTURA DO PROTÓTIPO

O desenvolvimento do protótipo terá como base a construção uma arquitetura dividida em pequenos serviços, onde cada um será responsável por atuar em uma parte específica da estrutura. Assim, haverá a implementação de um *backend* que conterá todas as regras de manutenção e retorno de informações que fazem parte das estruturas de replicação. O serviço de *frontend* será responsável em disponibilizar os acessos ao *backend* através de uma página gráfica de acesso.

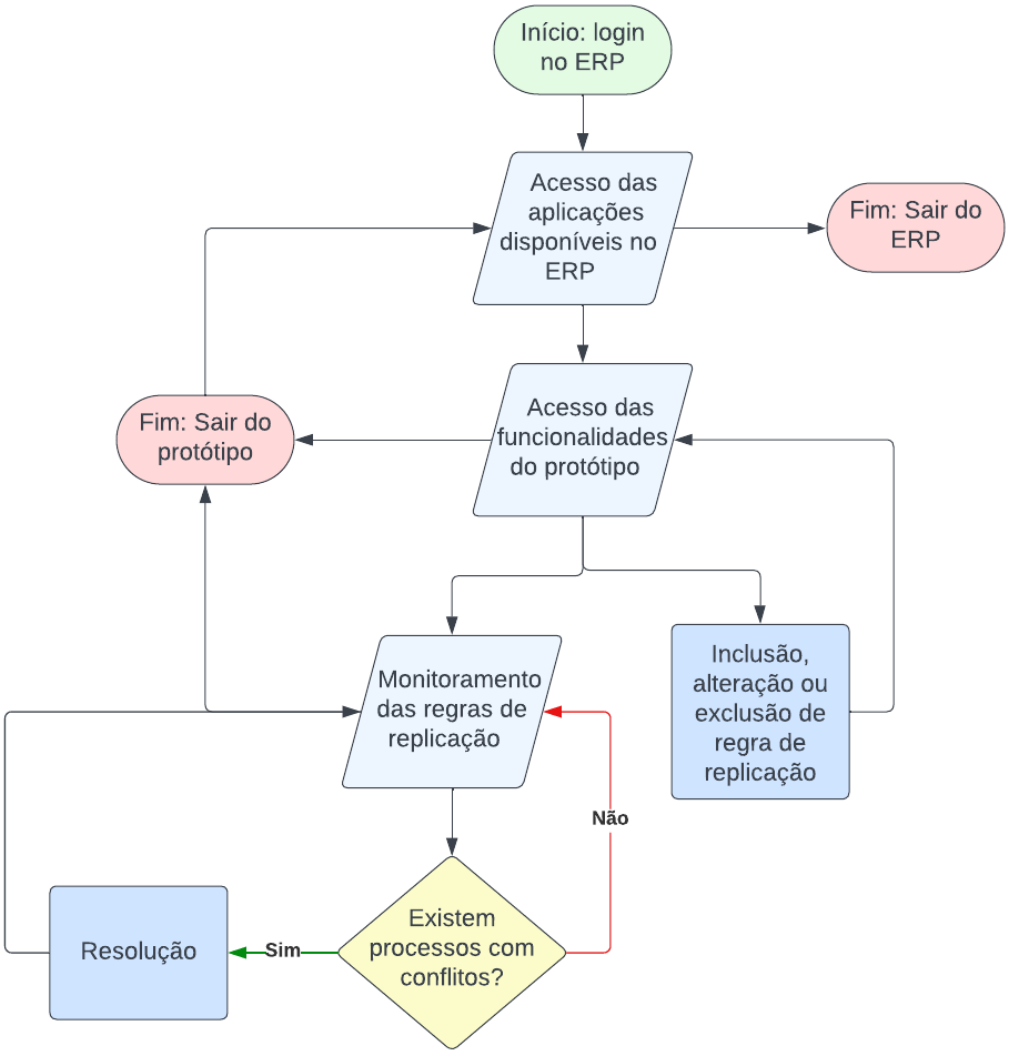
Em conjunto com a utilização dos serviços de manutenção das regras de replicação haverá um serviço que será destinado a realização de auditorias, onde para cada operação executada gravará informações importantes do estado do dado antes e depois da operação, e de quem fez e quando a fez.

## 4.8 TESTES

Após o desenvolvimento final do protótipo serão realizados diversos testes, tanto unitários como nos processos em conjunto, com objetivo de validar se os mesmos estão funcionando da maneira que foi proposto. Dessa forma, inicialmente foram organizados os seguintes testes:

- Testar cada *endpoint* individual do *backend* através da criação de um projeto xUnit;

Figura 7 – Caso de fluxo da utilização do protótipo




Fonte: Elaborado pelo autor.

- Testar a execução dos processos de persistência no banco de dados pelo *backend*;
- Testar as requisições realizadas pelas diferentes páginas e processos do *frontend*;
- Testar a autenticação do usuário no protótipo;
- Testar as regras de negócio para manutenção dos *replicates*;
- Testar a responsividade dos componentes das diferentes páginas do *frontend*;
- Testar a execução do protótipo em diferentes navegadores;
- Testar a utilização do protótipo em diferentes dispositivos móveis;
- Testar a execução dos serviços;

- Testar e validar se os *replicates* cadastrados estão funcionando conforme regra cadastrada, para a replicação funcionar de acordo o esperado;

# 5 DESENVOLVIMENTO

A presente seção tem como objetivo descrever as metodologias e tecnologias utilizados para o desenvolvimento do protótipo de gestão de replicação, exemplificando em detalhes como as estruturas e processos foram desenvolvidos para contemplar a construção da solução.

## 5.1 BACKEND

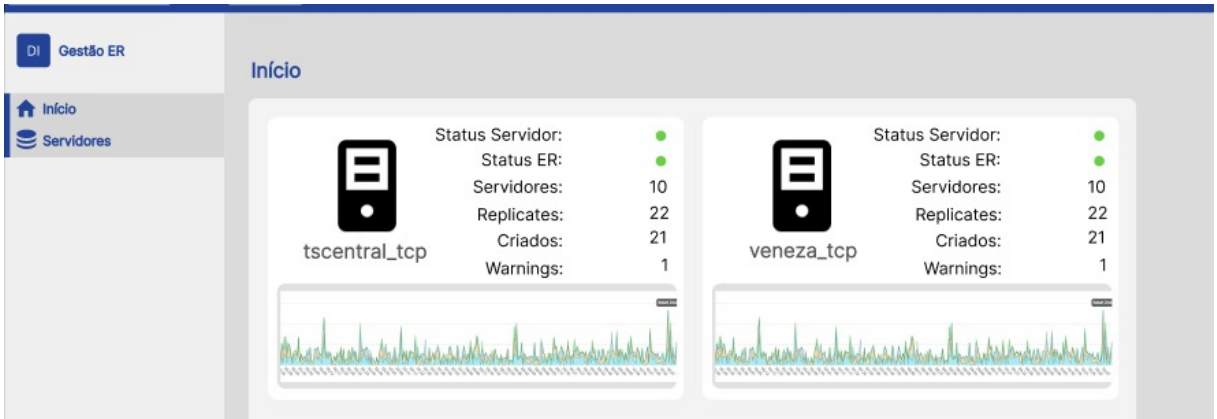
O desenvolvimento do *backend* do protótipo foi realizado criando uma Rest Web API na linguagem C#, onde a mesma tem como principal objetivo ser utilizada como ponto de acesso para manutenção das regras de replicação como também fornecimento dos dados para a aplicação do *frontend*.

### 5.1.1 Layout inicial e estrutura

Para auxiliar e embasar o desenvolvimento da aplicação, foi construído um *layout* inicial com o objetivo de validar os requisitos em conjunto com o usuário e também criar uma primeira versão da aplicação para guiar o desenvolvimento.

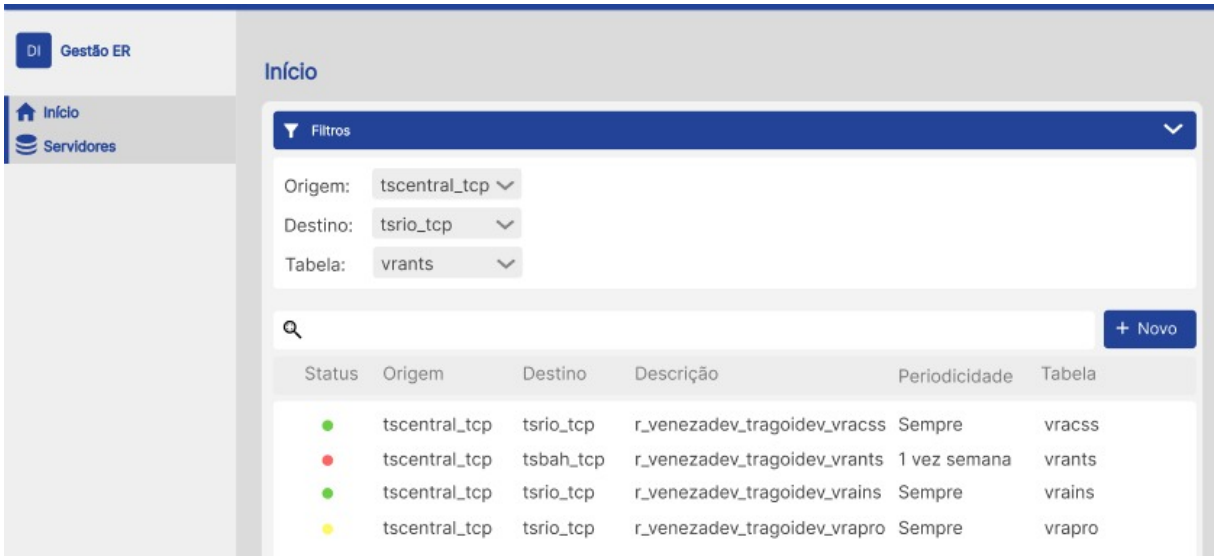
O protótipo de *layout* teve a construção de três modelos simulando as páginas de *layout* da aplicação. A Figura 8 exemplifica o primeiro modelo da página com as informações principais vinculadas a cada participante existente no sistema, como quantidade de regras vinculadas e o status de cada. O segundo modelo via Figura 9 representa os detalhes das regras de replicação com os seus dados, como status da regra, descrição, banco de dados e outros. A terceira e última janela via Figura 10 mostra as informações necessárias para a criação de uma regra de replicação com os *inputs* necessários para a execução.

Figura 8 – *Layout* inicial da tela de início



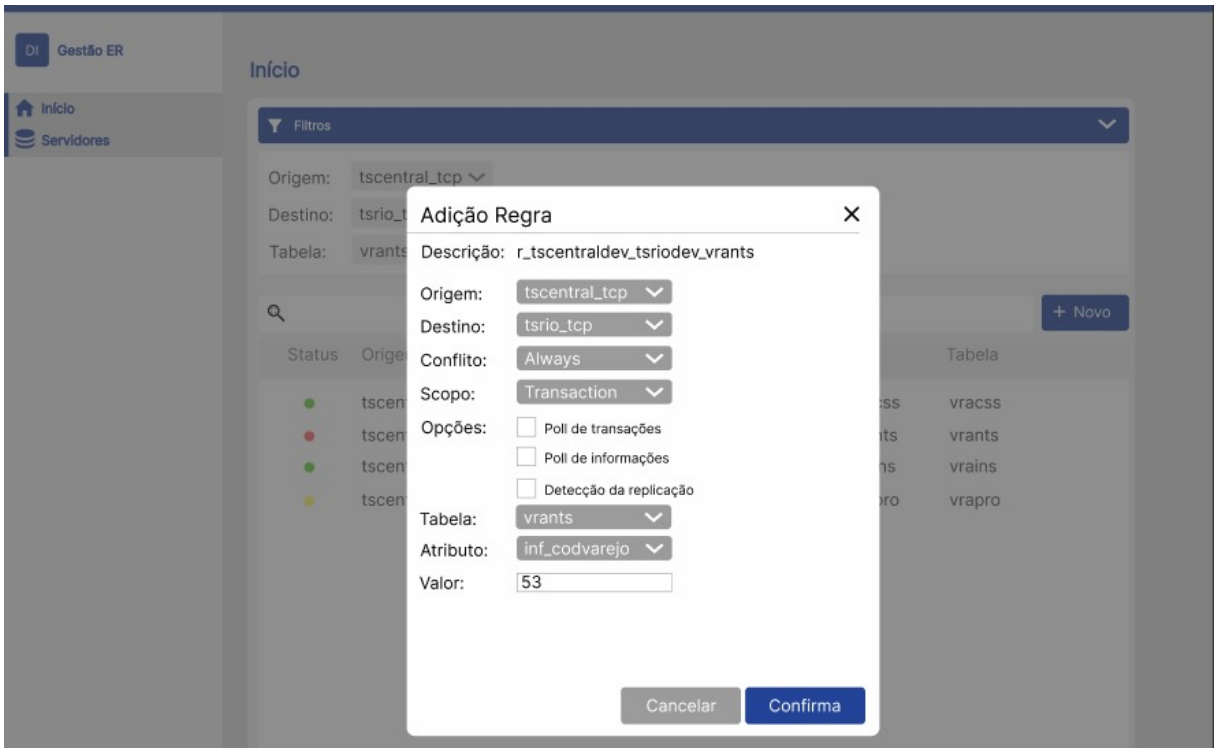

Fonte: Elaborado pelo autor.

Figura 9 – *Layout* inicial da tela das regras de replicação




Fonte: Elaborado pelo autor.

Figura 10 – *Layout* inicial da tela da manutenção das regras de replicação




Fonte: Elaborado pelo autor.

Conforme o andamento do desenvolvimento do trabalho, houve ajustes significativos no *layout* e nos processos, os quais visaram, principalmente, contemplar requisitos funcionais inicialmente não previstos e não mapeados, relacionados aos processos de gestão das regras de replicação de dados.

- **Filtros:** Ajuste dos campos de filtro de pesquisa das regras de replicação para "Status da

regra", "Banco de dados", "Tabela"e "Participante";

- **Microsserviços:** A ideia inicial era utilizar microsserviços, mas mudou-se a abordagem, pois foi identificado que o projeto inicial não comportaria a implementação, tornando-a mais complexa. No lugar de utilizar microsserviços, foi implementada uma estrutura de *failover* para conexão em diferentes bases de dados, caso alguma esteja indisponível;
- **Auditoria:** Criação de processos de auditoria para a execução das operações de manutenção das regras de replicação, com a gravação das informações em um banco de dados separado do protótipo;
- **Informações dos participantes:** Adição dos dados de banco de dados, tabela e filtros SQL segregados por participante. Cada participante pode ter a replicação de informações com diferentes estruturas, desde que as informações replicadas correspondam ao mesmo tipo de estrutura de tabela;
- **Manutenção das regras de replicação:** Somente foram implementadas as alterações de adicionar ou remover participantes de uma determinada regra de replicação, devido à complexidade de implementação;
- **Janela para manutenção da regra de replicação:** Ao invés de utilizar uma janela sobre a tela principal, foi criada uma nova página para realizar as manutenções das regras de replicação, garantindo melhor visualização e manipulação das informações;

Esses requisitos foram identificados após um estudo mais profundo do produto ER e a validação dos mesmos com os usuários-chave, adequando-os, dessa forma, à usabilidade do protótipo.

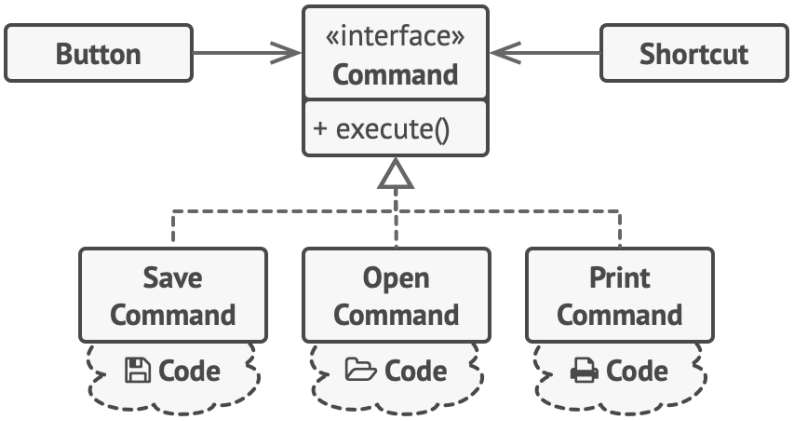
### 5.1.2 Arquitetura

Para a criação da API, foi utilizada uma estrutura em camadas, onde o código é separado conforme o padrão MVC, auxiliando na manutenção futura do código e melhorando a distinção das responsabilidades de cada camada internamente no sistema.

A solução adotada seguiu o padrão de projeto *Command Query Responsibility Segregation (CQRS)*, que propõe a separação entre as operações de leitura e escrita. Essa abordagem permite que as operações de consulta sejam tratadas por uma classe distinta das operações de comando, melhorando a escalabilidade e a clareza na gestão do sistema. Após o recebimento do comando pela classe *handler*, ela é responsável por orquestrar os processos subsequentes, podendo utilizá-los via injeção de dependência. Nos casos de execução de comandos de retorno de informações, são utilizados métodos contidos nas classes de *queries*. Já para operações que alteram o estado de um dado, o processo é validado por uma classe de serviço, para que, então, a classe de *repository* seja utilizada.

A implementação seguiu também o padrão de projeto *command*, no qual as classes executam os comandos recebidos pela classe controladora, utilizando um comando único de envio centralizado em uma interface. Isso evita a implementação de métodos específicos para cada operação, conforme exemplificado na Figura 11.

Figura 11 – Exemplo de estrutura de projeto command



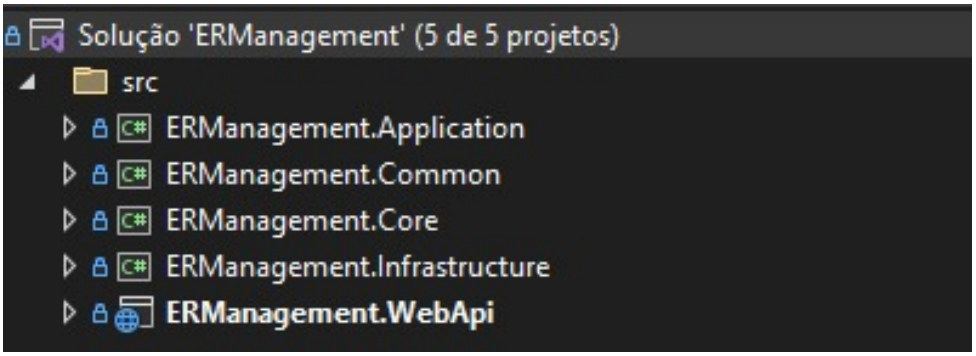

Fonte: (REFACTORING.GURU, 2024).

### 5.1.3 Configuração da aplicação

A aplicação *backend* foi desenvolvida dentro de uma solução composta por projetos com propósitos diferentes, com o objetivo de dividir as responsabilidades da aplicação. Todas as requisições recebidas pela API chegam primeiramente no projeto API Web, que basicamente contém os métodos controladores, para então serem direcionadas aos demais projetos de classes localizados nas camadas mais profundas da aplicação. A separação em diferentes projetos auxilia no desenvolvimento de diversos testes e, como mencionado no início, melhora a divisão das responsabilidades. Na figura Figura 12, pode ser visualizada a estrutura do projeto, conforme as especificações abaixo:

- **WebApi:** Centraliza as classes de *controllers*, *middlewares* e classes de extensão;
- ***Application*:** Gerencia as classes e métodos de validação e seleção de informações;
- ***Common*:** Gerencia as classes e métodos de uso geral entre os projetos;
- **Core:** Gerencia as classes centrais da aplicação, como modelos de domínio, conexão com o banco de dados e outros;
- ***Infrastructure*:** Gerencia as classes de manutenção das informações, executando comandos diretamente no banco de dados;

Figura 12 – Estruturação da solução




Fonte: Elaborado pelo autor.

### 5.1.4 Conexão entre os projetos

Toda requisição que o *backend* recebe possui uma classe *controller* correspondente, com o objetivo de separar e centralizar as requisições por modelo de dados. Para haver a comunicação entre os projetos WebApi e *Application* é utilizado um componente *nuget* que é composto por uma coleção de bibliotecas e códigos que podem ser reutilizados na aplicação. Esse pacote é denominado como IMediator, onde uma classe é assinada como remetente e outra como destinatário conforme mostrada nas imagens Figura 13 e Figura 16. Na classe controladora quando a requisição chega ao *endpoint* conforme imagem Figura 14, é criado um objeto com a especificação de *command* com os dados necessários e então executado um comando de envio para então ser recebido pela classe *handler* localizado no projeto *Application* conforme mostrado na imagem Figura 15.

Figura 13 – Criação do objeto de comando

```
5 referências | Augusto Fernando Klein, há 3 dias | 1 autor, 1 alteração
public class StartReplicateCommand(int userCode, string replicateName) : IRequest<Result<MaintenanceResponseDTO>>
{
    1 referência | Augusto Fernando Klein, há 3 dias | 1 autor, 1 alteração
    public int UserCode { get; set; } = userCode;
    4 referências | Augusto Fernando Klein, há 3 dias | 1 autor, 1 alteração
    public string ReplicateName { get; set; } = replicateName;

    1 referência | Augusto Fernando Klein, há 3 dias | 1 autor, 1 alteração
    public static StartReplicateCommand CreateCommand(int userCode, string replicateName) =>
        new(userCode, replicateName);
}
```

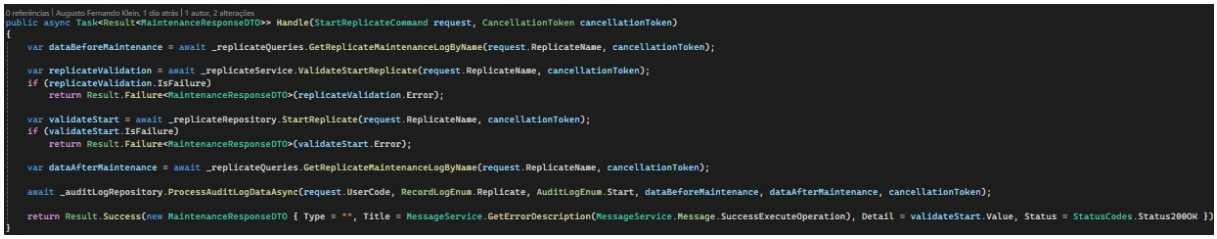
Fonte: Elaborado pelo autor.

O uso desse tipo de padrão no projeto ajuda a evitar a criação de dependências de processos entre as classes, centralizando os procedimentos de execução e envio das mensagens em uma única classe responsável pela gestão dos processos. Outra vantagem de sua utilização é a definição de responsabilidades claras para a classe recebedora do comando *handler*, que se encarrega da execução de um determinado conjunto de dados do sistema.

Figura 14 – Instanciação do objeto de comando e envio


Fonte: Elaborado pelo autor.

Figura 15 – Método de recebimento do comando




Fonte: Elaborado pelo autor.

Figura 16 – Definição do comando na classe *handler*


Fonte: Elaborado pelo autor.

### 5.1.5 *Endpoints*

Os *endpoints* têm como finalidade ser o ponto de entrada entre uma aplicação que precise realizar operações de consulta e manutenção diretamente nos dados contidos no banco de dados. Esse tipo de acesso é realizado por requisições via protocolo HTTP, sendo vantajoso contar com uma aplicação web que se encarregue de requisitar as informações.

Com base no planejamento dos requisitos propostos, o sistema pode ser dividido em três componentes fundamentais que compõem uma regra de replicação:

- **Grupos:** servidores com características similares organizados em grupos únicos, identificados por uma nomenclatura específica para serem utilizados na vinculação às regras de replicação;
- **Participantes:** grupos vinculados a uma regra de replicação, que atuarão ativamente na replicação das informações;

- **Regras:** estruturas criadas para gerenciar a replicação das informações, sendo anexadas informações como frequência, escopo, tratamento de conflitos e os participantes;

Assim, foi desenvolvido um *controller* único para cada estrutura, com o objetivo de melhorar a centralização dos processos que fazem parte da entidade e auxiliar na manutenção futura que possa ser realizada.

Cada *endpoint* possui a identificação da versão a ser utilizada pela aplicação que venha a utilizar seus processos. Dessa forma, podem coexistir diferentes versões do mesmo *endpoint*, mas com estruturas de processos distintas. A utilização de versionamento facilita a evolução da API, manutenções futuras e o atendimento a diferentes tipos de tecnologias e clientes que necessitem de diferentes métodos de acesso. No exemplo da Figura 17 pode-se observar que o versionamento se faz presente acima da declaração do *endpoint* de numeração 1.

Figura 17 – Exemplo de versionamento de *endpoint*

```
[MapToApiVersion("1")]
[HttpPost("create")]
[Roles("manutencao")]
0 referências | Augusto Fernando Klein, há 4 dias | 1 autor, 10 alterações
public async Task<IActionResult> CreateReplicate([FromBody] CreateReplicateInputModel inputModel, CancellationToken cancellationToken)
```

Fonte: Elaborado pelo autor.

Além da possibilidade de versionamento do *endpoint*, também é possível realizar o versionamento de toda a classe *controller*, em situações em que seja necessária uma reformulação total da entidade para desenvolvimento ou aprimoramento. Outro ponto importante a ser mencionado é que, quando toda a classe é versionada, os *endpoints* seguirão a versão descrita, não sendo necessária a definição de versão em cada *endpoint* individualmente.

Figura 18 – Exemplo de versionamento de controller

```
[Authorize]
[ApiController]
[ApiVersion("1")]
[Route("v{version:ApiVersion}/[controller]")]
[Profile("wsp_ermanager")]
0 referências | Augusto Fernando Klein, há 4 dias | 1 autor, 29 alterações
public class ReplicateController(IMediator mediator) : ControllerBase
```

Fonte: Elaborado pelo autor.

### 5.1.6 *Middleware*

Para o desenvolvimento de algumas funcionalidades do protótipo, foi necessário utilizar um *middleware* como base para a criação de diferentes conexões, que variam de acordo com a hospedagem do participante. Os dados necessários são enviados no cabeçalho da requisição e utilizados para definir o acesso. O *middleware* é empregado nas estruturas de definição de um novo participante, envolvendo o conhecimento das estruturas de banco de dados e tabelas

existentes, que podem ser integradas a uma determinada regra de replicação. Dessa forma, ao definir um novo participante, o usuário precisará informar qual *database* integrará a regra e qual tabela correspondente será utilizada. Esse processo envolve consultas frequentes aos participantes, para buscar os dados e apresentá-los ao usuário na interface. Na imagem Figura 19 é possível visualizar os processos necessários para interceptar a requisição e criar uma conexão personalizada com o banco de dados.

Figura 19 – Exemplo de *middleware* para conexão entre diferentes bancos de dados

```
0 referências | Augusto Fernando Klein, há 2 dias | 1 autor, 3 alterações
public async Task Invoke(HttpContext context, IConfiguration configuration, IParticipantDatabaseFactory participantDatabaseFactory)
{
    if (context.Request.Headers.TryGetValue("Participant-Name", out var participantName))
    {
        string participantValue = participantName.ToString().Trim();

        if (!string.IsNullOrEmpty(participantValue) && !string.IsNullOrWhiteSpace(participantValue))
        {
            const string sql = @"SELECT host_name as HostName,
                                        host_port as HostPort
                                   FROM participant_connection
                                  WHERE participant_name = @participantName";

            var erManagementConnectionString = configuration.GetConnectionString("ermanagement");

            if (!string.IsNullOrEmpty(erManagementConnectionString))
            {
                ERManagementDatabaseFactory erManagementDatabase = new(erManagementConnectionString);
                var participantConnectionData = await erManagementDatabase.Connection.QueryFirstAsync<ParticipantConnectionData>(sql, new { participantName });
                erManagementDatabase.Dispose();

                if (participantConnectionData != null)
                {
                    var connectionString = $"Server={participantConnectionData.HostName},{participantConnectionData.HostPort}" + ";" + $"Database={GetDatabaseContext(context)}" + ";" + configuration.GetConnectionString("participant");
                    participantDatabaseFactory.Connect(connectionString);
                }
            }
        }
    }

    await next(context);
}
```

Fonte: Elaborado pelo autor.

### 5.1.7 Retorno das mensagens de execução dos comandos

Na execução de comandos de manutenção das regras de replicação, o sistema do ER devolve um *output* de status e informações sobre os processos executados durante a operação de manutenção. Esse tipo de retorno ao usuário é de extrema importância, pois é com base nele que o DBA pode confirmar se o comando foi executado corretamente ou, em caso de erro, identificar o tipo de erro gerado. Esse retorno foi desenvolvido para fornecer uma mensagem ao usuário, servindo como um *feedback*.

Na Figura 20 é possível verificar o tipo de retorno de uma manutenção bem sucedida de manutenção de uma regra de replicação. Na Figura 21 é possível verificar que na adição de um novo participante são retornadas informações importantes em relação aos processos executados. Já na Figura 22 é exemplificado um comando que ocasionou algum erro na vinculação mostrando todos os passos que foram realizados.

Figura 20 – Mensagem de retorno de sucesso ao executar uma operação de manutenção

```
{
    "type": "",
    "title": "Operação executada com sucesso",
    "detail": "OK",
    "status": 200
}
```

Fonte: Elaborado pelo autor.

Figura 21 – Mensagem de retorno de sucesso ao adicionar um participante


```
{
    "type": "",
    "title": "Operação executada com sucesso",
    "detail": "Verification of tramon@grp_trarec_dev:informix.vracss started \nVerification of tramon@grp_trarec_dev:informix.vracss is successful",
    "status": 200
}
```


Fonte: Elaborado pelo autor.

Figura 22 – Mensagem de retorno de erro ao adicionar um participante


```
{
    "type": "/v1/participant/add",
    "title": "Erro na execução da operação",
    "detail": "Verification of tramon@grp_trarec_dev:informix.vracss started \nVerification of tramon@grp_trarec_dev:informix.vracss failed          \nparticipant tramon@grp_trarec_dev:informix.
        vracss 'select * from informix.vracs where inf_codvarejo in (31,53) and etbcod = 23'\ncommand failed -- table does not exist  (29)",
    "status": 400
}
```


Fonte: Elaborado pelo autor.

### 5.1.8 Multiconexão entre instâncias e bancos de dados

No momento da criação de uma regra de replicação ou na adição de novos participantes a uma regra já existente, é necessária a busca de algumas informações, como os bancos de dados e as estruturas de tabelas correspondentes, para então disponibilizá-las ao usuário, permitindo que essas informações sejam agregadas durante a execução da operação.

Quando o usuário seleciona algum grupo de replicação ou banco de dados no *frontend*, é enviado um comando de busca das informações, adicionando os parâmetros no cabeçalho da requisição. Um *middleware* é responsável por realizar a conexão, selecionando os dados como *host* e porta, conforme a especificação do grupo de replicação e do banco de dados. As informações de conexão foram armazenadas em *secrets* do próprio pod de execução da API, utilizando o sistema de orquestração de *containers*.

### 5.1.9 Estrutura de *failover*

O sistema do ER utiliza uma estrutura de catálogo onde cada grupo de replicação tem o conhecimento do restante da estrutura. Dessa forma, uma determinada operação de criação ou manutenção de regra de replicação pode ser executada mesmo havendo algum grupo indisponível no instante. Esse tipo de operação abre a possibilidade da não centralização das operações, sendo possível utilizar qualquer grupo como base para a execução das operações.

Para poder garantir uma maior resiliência da disponibilidade do protótipo foi realizada a implementação de uma lógica de *failover*, onde basicamente no momento em que é necessário o processamento de uma determinada operação no banco de dados é selecionada a primeira conexão válida entre de um conjunto pré definido, onde a primeira conexão com disponibilidade é selecionada para executar a operação. Na figura Figura 23 é exemplificado o modo de implementação das conexões.

Figura 23 – Implementação de processo de *failover* de conexão

```
1 referência | Augusto Fernando Klein, há 5 dias | 1 autor, 2 alterações
public static string? GetSyscdrDefaultConnectionString(this IConfiguration configuration, string connectionSection)
{
    var connectionsStrings = new Dictionary<string, ConnectionString>();
    configuration.GetSection($"ConnectionStrings:{connectionSection}").Bind(connectionsStrings);

    int totalConnections = connectionsStrings.Values.Count;
    int currentIndex = 0;

    foreach (ConnectionString connectionString in connectionsStrings.Values)
    {
        currentIndex++;

        try
        {
            var builtConnectionString = connectionString.Build();
            using SyscdrDatabaseFactory syscdrDatabaseFactory = new(builtConnectionString);
            syscdrDatabaseFactory.Dispose();

            return builtConnectionString;
        }
        catch (Exception ex)
        {
            if (currentIndex == totalConnections)
            {
                throw new InvalidOperationException("Error defining the connection to perform maintenance", ex);
            }

            continue;
        }
    }

    throw new InvalidOperationException("No valid connection strings provided");
}
```

Fonte: Elaborado pelo autor.

## 5.2 *FRONTEND*

A programação da parte do *frontend* foi desenvolvida utilizando o *framework* React com a biblioteca Next.js, onde a interação do usuário ocorre totalmente em ambiente web. A escolha dessa tecnologia levou em consideração o prévio conhecimento da linguagem em ambiente corporativo, o que reduziu a curva de aprendizagem, além da possibilidade de acesso via qualquer dispositivo com conexão à internet.

### 5.2.1 Arquitetura

A arquitetura da aplicação *frontend* foi baseada na construção de pequenos componentes, onde cada um possui uma responsabilidade única, tanto para renderizar informações na tela quanto para persistir operações no banco de dados.

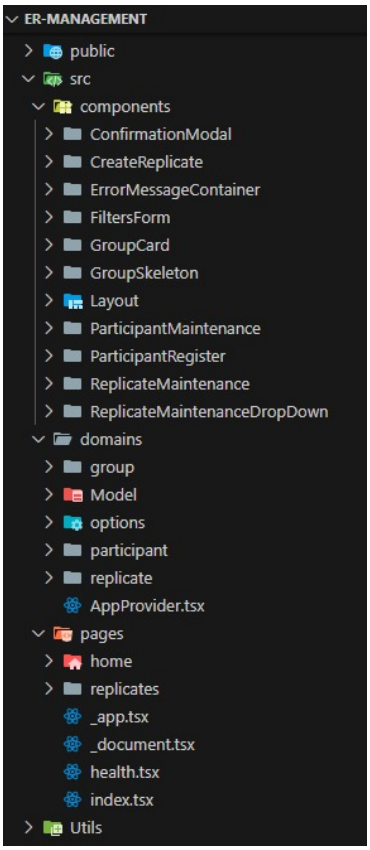
Conforme exemplificado na Figura 24 o projeto é dividido em vários diretórios contendo componentes e processos distintos:

- ***public*:** Contém os arquivos de tradução para os idiomas português, inglês e espanhol;
- **src:** Diretório que contém os componentes do protótipo;

- ***components*:** Componentes responsáveis pela renderização de estruturas na interface, como botões, tabelas, *inputs*, *comboboxes* e outras estruturas gráficas, com extensão *tsx*;
- ***domains*:** Estruturas de domínio, como tipos de dados utilizados nas requisições para a API, estruturas para carregamento de informações, e outras. Também armazena as definições das requisições contendo os endereços dos *endpoints* para acesso às APIs;
- ***pages*:** Arquivos que constituem as páginas do protótipo, responsáveis por instanciar os componentes para formar a estrutura de visualização;
- **utils:** Funções, constantes e estruturas de enumerações genéricas que são utilizadas ao longo da aplicação;

Atualmente, no ambiente corporativo, durante a criação do projeto *frontend*, é desenvolvido um *layout* base contendo os principais componentes, como *toolbar*, páginas e informações de login. Essa criação inicial facilita o andamento do desenvolvimento e contribui para a padronização das aplicações construídas internamente na empresa.

Figura 24 – Arquitetura do projeto *frontend*



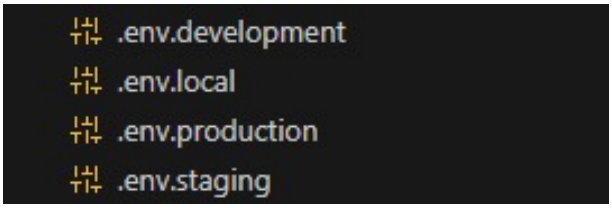

Fonte: Elaborado pelo autor.

### 5.2.2 Variáveis de ambiente

O protótipo *frontend* utiliza arquivos específicos para organizar as variáveis de ambiente empregadas na aplicação. Na maioria dos casos, essas variáveis contêm informações sensíveis que não devem ser expostas por motivos de segurança, como *endpoints* e chaves de acesso. Durante o processo de *deploy* da aplicação, as variáveis de ambiente são armazenadas em *secrets* no próprio pod de execução, sendo utilizadas de acordo com o ambiente específico onde a aplicação será executada. Na Figura 25, é possível identificar que são utilizados arquivos distintos para diferentes ambientes de execução, cada um com declarações específicas de acesso aos dados. No protótipo, são utilizados os seguintes arquivos:

- **.env.local:** Variáveis de ambiente para uso local;
- **.env.development:** Variáveis de ambiente para o ambiente de desenvolvimento;
- **.env.staging:** Variáveis de ambiente utilizadas em ambientes de pré-produção, destinados à execução de testes, simulando um ambiente próximo ao de produção;
- **.env.production:** Variáveis de ambiente para o ambiente de produção;

Figura 25 – Arquivos de ambientes de execução




Fonte: Elaborado pelo autor.

### 5.2.3 Acesso aos *endpoints* do *backend*

Para acessar os *endpoints*, são realizados processos de requisição correspondentes aos métodos HTTP definidos no *backend*. Ao longo do protótipo, há situações em que é necessário apenas consultar informações para popular um componente ou página. Em muitos casos, esse processo de requisição é feito diretamente no componente. Na Figura 26, é ilustrado um exemplo de carregamento de dados para os componentes de filtros das regras de replicação, onde são realizadas várias requisições para buscar informações.

Para os serviços de persistência, foram criados diretórios específicos para centralizar os métodos. Essa metodologia visa melhorar a organização do código, agrupando os processos em diretórios chamados "serviços", onde a execução das operações é realizada internamente nas funções. Na Figura 27, é exemplificado um processo de envio de comando para iniciar uma determinada regra de replicação.

Figura 26 – Processo de carregamento de dados de filtros

```
const { data: groupRegisters } = useFetchOnce<GroupOption[]>(
  `/api-er/v1/group/registers`,
);

const { data: databaseRegisters } = useFetchOnce<DatabaseOption[]>(
  `/api-er/v1/replicate/databases`,
);

const { data: tableRegisters } = useFetchOnce<TableOption[]>(
  `/api-er/v1/replicate/tables`,
);
```

Fonte: Elaborado pelo autor.

Figura 27 – Serviço de alteração do estado de uma regra de replicação

```
export const executeProcessStartReplicate = async (
  form: ReplicateOperation,
): Promise<ApiRequestResponseData> => {
  return await requestErrorWrapper(async () => {
    const response = await api.post(`/api-er/v1/replicate/start`, {
      ...form,
    });
    return response.data;
  });
};
```

Fonte: Elaborado pelo autor.

Com os exemplos de processos de busca de informações e execução de comandos de persistência, é importante destacar que todos os processos foram desenvolvidos para serem executados de forma assíncrona, utilizando o comando *async*. No *frontend*, o código aguarda o retorno do *backend* por meio do comando *await*, com a resposta correspondendo ao comando executado. Essa metodologia foi adotada porque o usuário precisa do retorno das informações para dar continuidade a um determinado processo. Por exemplo, para a criação de uma regra de replicação, é necessário obter dados sobre os grupos disponíveis, bancos de dados e tabelas.

## 5.3 AUTENTICAÇÃO DO USUÁRIO

A autenticação do usuário é um fator crucial para o uso do protótipo, pois garante que somente usuários autorizados tenham acesso. Os procedimentos de autenticação utilizados tanto no *backend* quanto no *frontend* seguem os métodos padrão de autenticação desenvolvidos internamente no ambiente corporativo, padronizando a metodologia de uso em todas as equipes de trabalho.

Atualmente, todas as aplicações internas no sistema corporativo exigem que o usuário possua um perfil com as funcionalidades autorizadas. Caso contrário, o usuário não terá acesso a nenhuma funcionalidade do sistema. Para aumentar ainda mais a segurança, o *token* utilizado para acessar o *backend* do protótipo é gerado exclusivamente por um componente no *frontend*, garantindo que o acesso seja liberado de forma controlada e autorizada.

## 5.4 GERENCIAMENTO DE *LOG*

Com a necessidade de uso do protótipo por múltiplos usuários no ambiente corporativo, tornou-se essencial implementar uma estrutura de gestão de *logs*, permitindo a auditoria das execuções das operações e o acompanhamento das informações antes e depois das modificações.

Para a gestão dos *logs*, foi utilizado um *pod* orquestrado pelo *Kubernetes*, com um banco de dados Informix relacional dedicado à aplicação. Isso possibilita a gestão eficiente das informações de *log*. A gravação dos *logs* ocorre em cada operação que altere o estado de uma regra de replicação, registrando o usuário que realizou a execução, a estrutura alvo da manutenção e o dado antes e depois da execução.

Na Figura 28 é possível visualizar que a gestão do *log* no protótipo é realizada pela estrutura de três tabelas localizadas no banco de dados, que são: `record_operation_type`, `audit_operation_type` e `registrations_audit`. A tabela nomeada via nomenclatura `record_operation_type` possui a informação de qual record se destina o objetivo de alteração das informações, onde atualmente pode ser na rede de replicação quanto nos participantes que compõem a regra. A tabela `audit_operation_type` é correspondente ao tipo da operação de manutenção, *insert*, *remove*, *start*, *stop*, *suspend* e *resume*. A última tabela denominada `registrations_audit` possui os registros de *log* com todas as informações necessárias para identificar uma operação realizada pelo usuário.

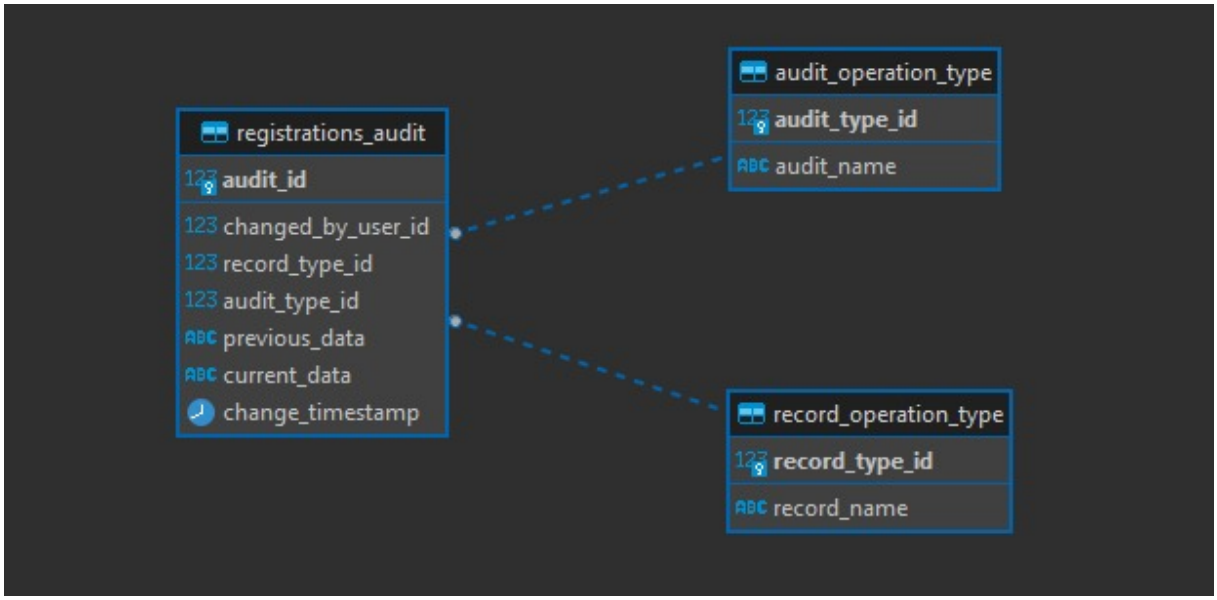
As tabelas `record_operation_type` e `audit_operation_type` possuem chaves primárias dos tipos de dados, que então são utilizadas como chaves estrangeiras pela tabela `registrations_audit`. A arquitetura auxilia na criação de uma dependência entre as tabelas, que caso futuramente houver a necessidade de haver novos tipos de dados, os mesmos poderão ser inseridos e já utilizados.

## 5.5 PERFIS DE ACESSO

Para gerenciar os acessos no protótipo, foram criados perfis com privilégios distintos, cada um correspondente às permissões necessárias para o perfil do usuário. A aplicação pode ser acessada tanto pelos DBAs responsáveis pela manutenção das regras de replicação quanto por usuários chave definidos dentro dos *squads* de grupos de trabalho.

A atribuição dos perfis de acesso será realizada por meio da aplicação interna já existente

Figura 28 – Estrutura de tabelas para gestão do *log* da aplicação

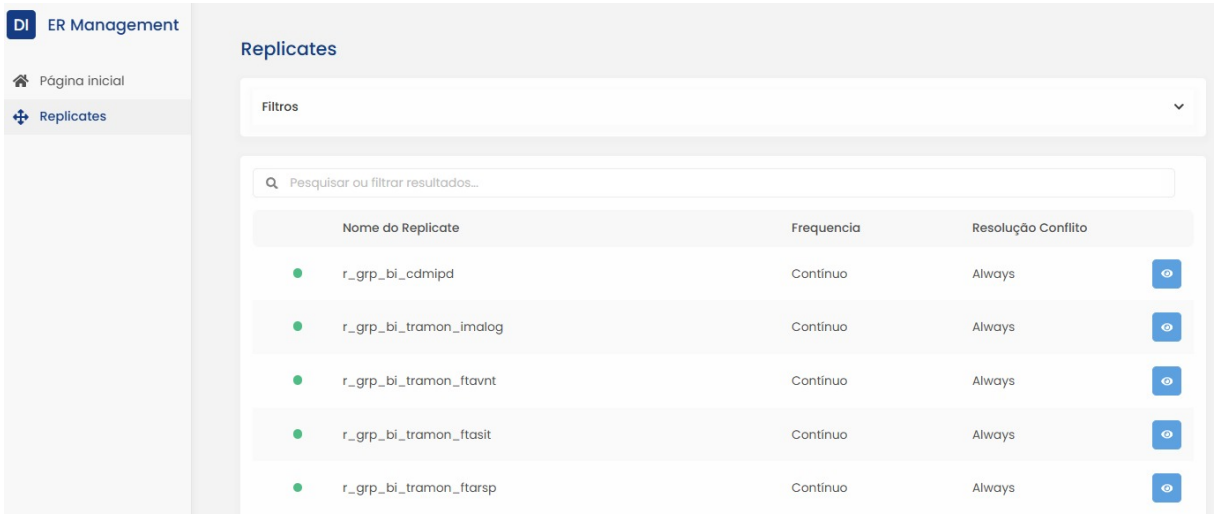



Fonte: Elaborado pelo autor.

na empresa. Nela, é possível definir diferentes tipos de permissões conforme a necessidade, tornando a gestão mais flexível para situações específicas.

Para usuários que apenas realizarão a visualização das informações, será atribuído o perfil de "Consulta", com permissão apenas para consultar e verificar os dados na plataforma. Nas imagens Figura 29 e Figura 30, é possível verificar como o usuário visualiza os dados com esse perfil.

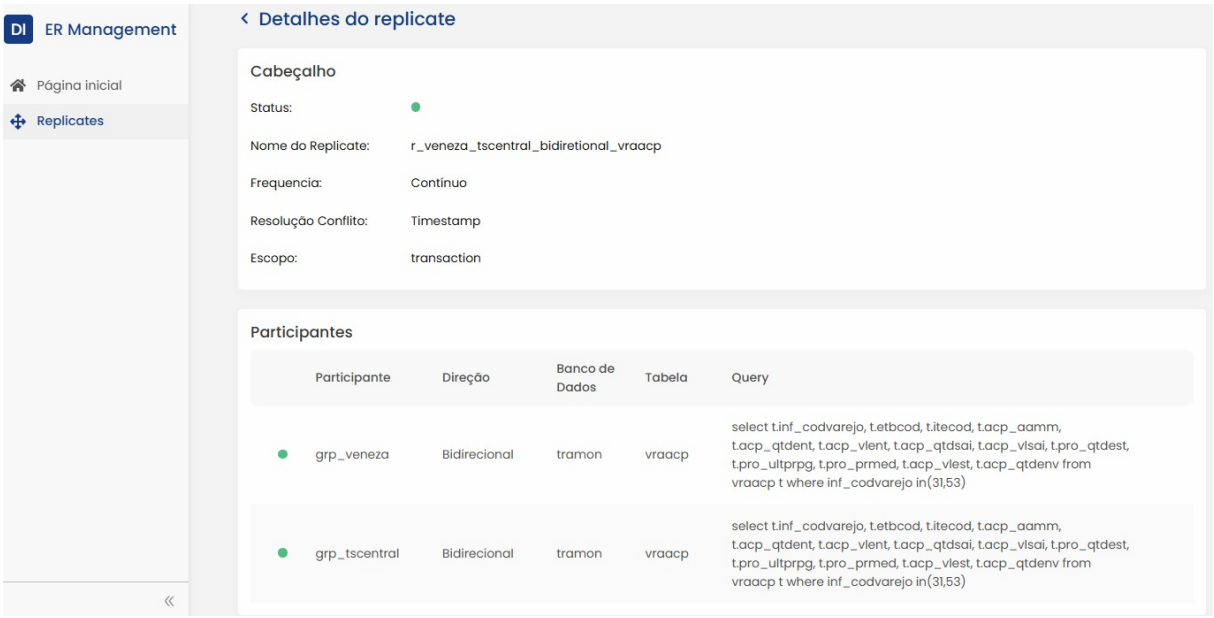
Figura 29 – Visualização das regras de replicação para o perfil de consulta




Fonte: Elaborado pelo autor.

Para os usuários que necessitam realizar manutenções nas regras de replicação, será atribuído o perfil de "Manutenção". Esse perfil permite a criação de novas regras de replicação, bem como a manutenção do estado das regras e dos participantes que as compõem. Ele é

Figura 30 – Visualização dos detalhes da regra de replicação para o perfil de consulta



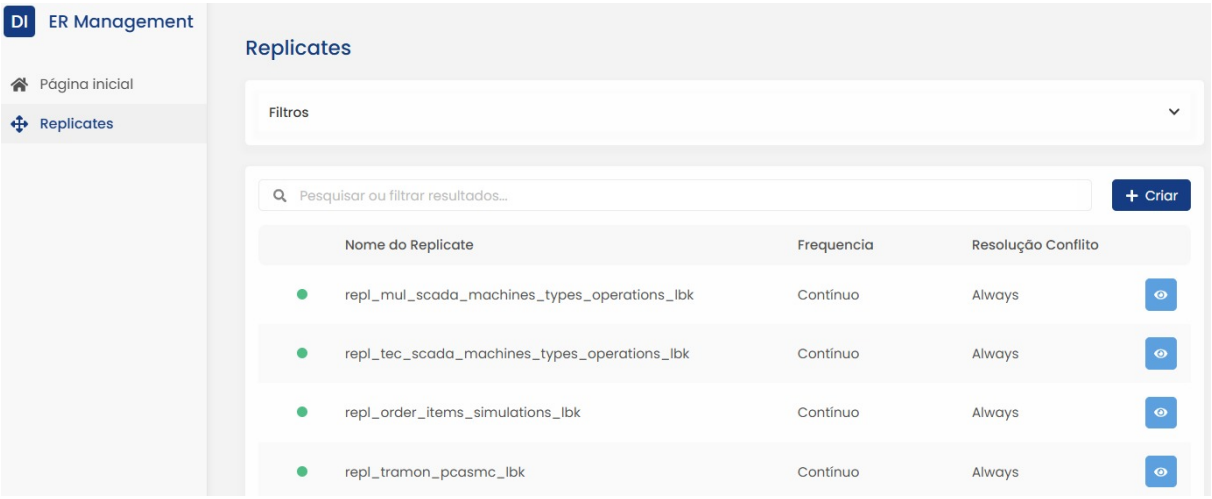

Fonte: Elaborado pelo autor.

destinado, principalmente, aos DBAs responsáveis pela atualização e manutenção de programas, onde em muitos casos será necessária a alteração de uma regra de replicação.

Já para os DBAs que atuam diretamente na parte de replicação, será atribuído o perfil de acesso "Total". Esse perfil é destinado aos principais usuários do protótipo, permitindo que eles realizem todas as manutenções disponíveis no perfil de "Manutenção", além de acessar funcionalidades adicionais que serão desenvolvidas futuramente, exclusivas para esse tipo de usuário.

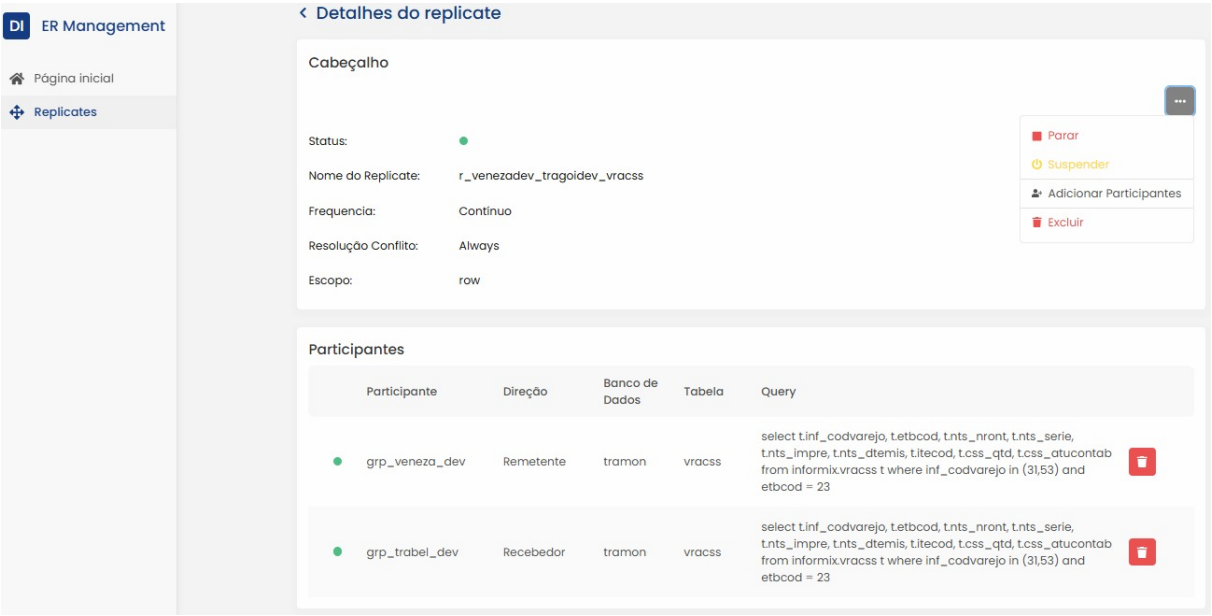
Na Figura 31 e Figura 32, é possível visualizar como a aplicação é renderizada para os usuários com os perfis de acesso "Manutenção"e "Total"vinculados.

Figura 31 – Visualização das regras de replicação para os perfis Manutenção e Total




Fonte: Elaborado pelo autor.

Figura 32 – Visualização dos detalhes da regra de replicação para os perfis Manutenção e Total




Fonte: Elaborado pelo autor.

# 6 ESTUDO DE CASO E TESTES

Para embasar o uso do protótipo, definiu-se um cenário hipotético para simular um ambiente onde exista a necessidade de replicação de determinadas informações para validar os processos e regras necessárias para a criação de uma estrutura desse modelo. E para avaliar o funcionamento geral das estruturas implementadas, fez-se algum testes mais específicos.

## 6.1 ESTUDO DE CASO

Em um ambiente hipotético, é criada a necessidade de haver a replicação de uma determinada tabela que gerencia informações de saída de embalagens no PDV de uma unidade. Seguem abaixo os requisitos para a replicação das informações:

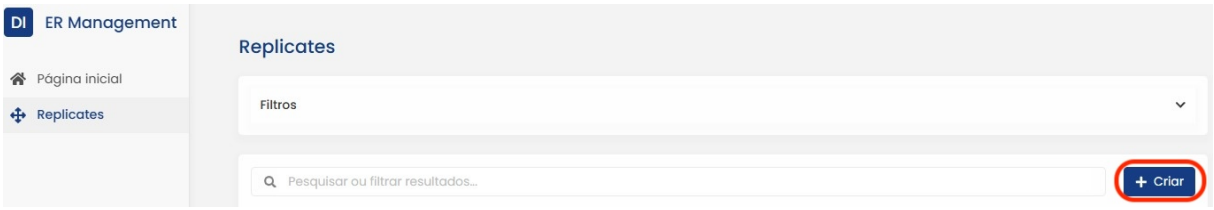
- **Participantes:** Um dos participantes é o ambiente onde ocorre a operação de saída das sacolas, e o outro deve apenas receber as informações;
- **Databases:** Os participantes utilizarão a mesma estrutura de *database*;
- **Tabela:** Estrutura de tabela para o gerenciamento das informações de saída de embalagens do PDV da unidade;
- **Filtros:** No ambiente existem diversas outras unidades operando, portanto, deve haver um filtro para que somente o estabelecimento de número 1 tenha as informações replicadas;
- **Resolução de Conflito:** Caso ocorra qualquer conflito, ele deverá ser tratado adequadamente;
- **Frequência da Replicação:** A replicação deve ocorrer continuamente, em tempo real;
- **Escopo:** A replicação deverá ser realizada por transação, ou seja, os dados devem ser enviados somente quando a transação for finalizada.

O processo de criação da regra de replicação inicia-se ao acessar o botão 'Criar', localizado na página das regras de replicação. Ao clicar no botão, abre-se uma estrutura modal, conforme mostrado na Figura 33, onde campos para inserção dos dados são disponibilizados.

Ao verificar as informações do cabeçalho da regra de replicação, são exigidos campos, conforme mostrada na Figura 34, onde é necessário informar o nome, a resolução de conflito, o escopo e se a regra de replicação será baseada em *loopback*.

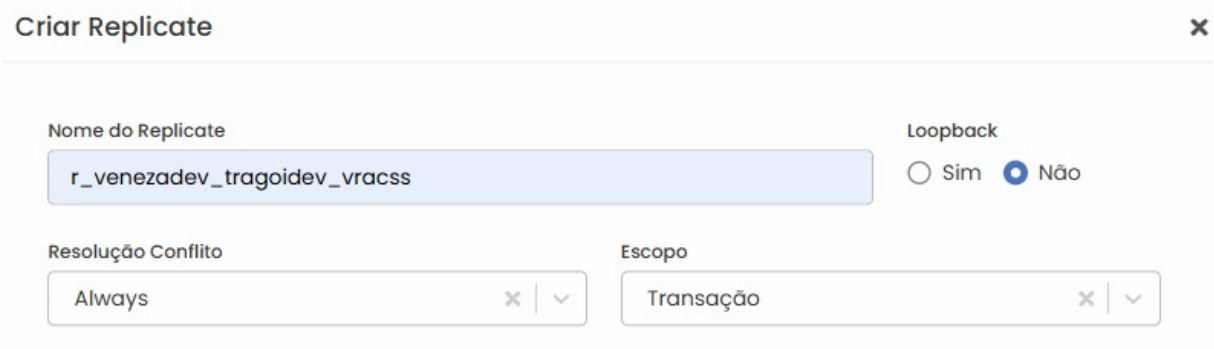
Após inseridas as informações do cabeçalho da replicação, será possível inserir os dados dos participantes. Na Figura 35 e na Figura 36, é possível verificar que, para cada grupo, é

Figura 33 – Botão para criação da regra




Fonte: Elaborado pelo autor.

Figura 34 – Cabeçalho da regra de replicação




Fonte: Elaborado pelo autor.

necessário informar o participante, sua direção, o banco de dados, a estrutura de tabela e o filtro correspondente ao participante.

Figura 35 – Dados do primeiro participante


Fonte: Elaborado pelo autor.

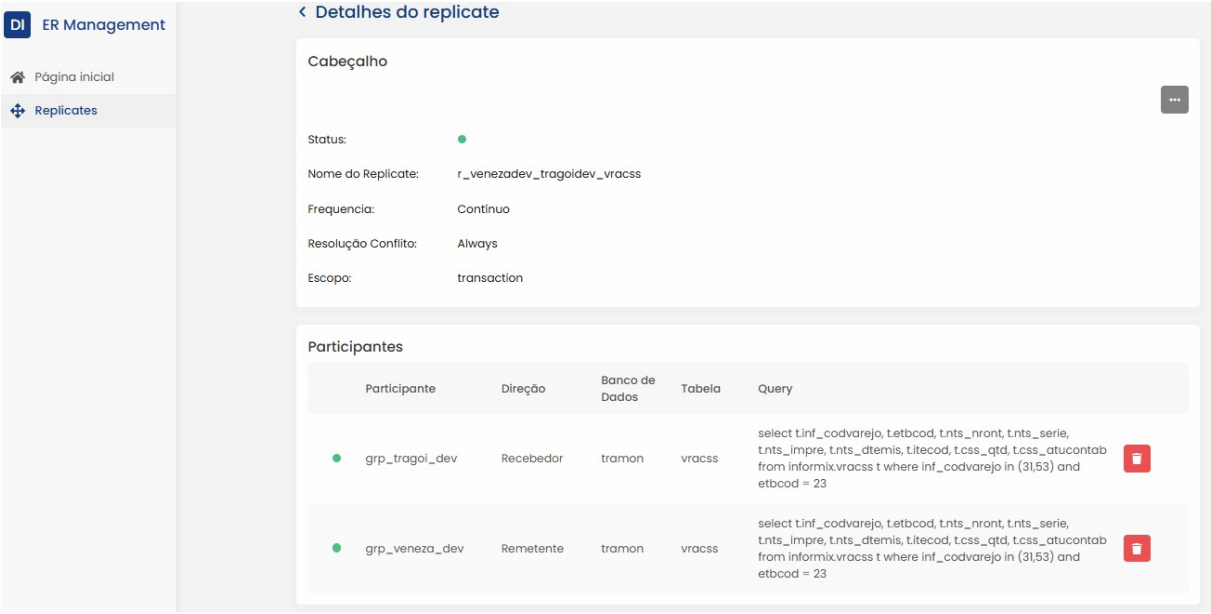
Após a execução da criação da regra de replicação, ela estará válida no sistema. Caso alguma operação seja equivalente aos filtros definidos pelo participante, a replicação será executada. As informações da regra criada podem ser verificadas naFigura 37.

O protótipo também permite a modificação da regra, possibilitando a adição ou remoção de novos participantes. Para adicionar um novo participante, é necessário selecionar as opções

Figura 36 – Dados do segundo participante


Fonte: Elaborado pelo autor.

Figura 37 – Dados da regra de replicação criada




Fonte: Elaborado pelo autor.

da regra e escolher a opção 'Adicionar Participante'. Isso abrirá uma nova janela modal com campos destinados às informações do participante, conforme identificados na regra de criação.

O protótipo também disponibiliza a opção de remoção de um participante vinculado à regra de replicação. Para realizar a operação de remoção, basta clicar no botão localizado ao lado de cada participante presente na regra de replicação.

## 6.2 TESTES

O desenvolvimento dos testes foi segregado conforme o desenvolvimento da aplicação. A execução dos procedimentos de teste foi implementada após a validação dos requisitos com os usuários-chave, após a implementação, garantindo assim a validação da implementação.

> "O resultado desejado do teste é uma garantia que um programa satisfaz suas especificações, mas como observado por Edsger Dijkstra, os testes podem provar a presença de erros, mas não sua ausência. No entanto, testes cuidadosos aumentam muito a confiança de que um programa funciona conforme o esperado."(OLAN, 2001).

As estruturas de teste do protótipo foram concentradas no *backend*, onde as regras de negócios estão presentes, sendo a parte mais sensível da aplicação, necessitando de um enfoque maior. A parte do *frontend* também foi testada de forma diferente, onde não foram construídos processos de validação do código desenvolvido, mas sim, utilizado pelos usuários-chave para realizar a manutenção de regras e consulta de informações.

Para a estrutura do *backend*, a utilização dos testes foi focada nas estruturas que visassem validar as regras de negócio que envolvem a manutenção das regras de replicação do protótipo, simulando a execução das regras da forma mais fiel possível ao ambiente produtivo.

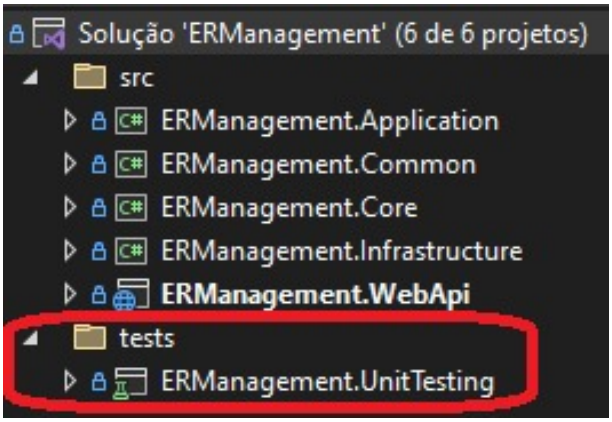
No projeto da API em C# foi realizada a criação de um projeto destinado a realizar somente testes de estruturas pre definidas. Assim, foram selecionadas as classes *handler* e *service* de cada modelo de dados onde são elas que realizam a orquestração e validação das informações.

Para a criação do projeto de testes foi escolhido o modelo xUnit da ferramenta onde a mesma disponibiliza métodos próprios para realizar a implementação, preparação e execução dos testes. Com base na estrutura do trabalho foi selecionado o modelo de testes unitários, onde visa a realização de testes unitários em diversas partes distintas das classes. Na Figura 38 é possível verificar o projeto de testes criado.

Para a implementação dos testes, foi utilizado o padrão Arrange, Act, Assert (AAA), que visa realizar testes separadamente com base na estrutura da classe. A escolha desse modelo de teste também se baseia no conhecimento prévio já adquirido em ambiente corporativo, onde o mesmo é utilizado em outros projetos desenvolvidos.

> "O padrão AAA refere-se ao layout de três seções de escrever um caso de teste de unidade: organizar, agir e afirmar [1]. O padrão AAA proporciona um fluxo natural e intuitivo para criar um caso de teste de unidade. Em *Arrange*, o necessário ambiente, como criação de objetos e configuração simulada, está preparado. A função *act* que está sendo testada é executado. Em *assert*, o resultado real do ato é verificado contra a expectativa. Uma falha no teste é levantada para atenção quando o resultado real não corresponde à expectativa."(WEI *et al.*, 2024).

Figura 38 – Estrutura com o projeto de teste




Fonte: Elaborado pelo autor.

O desenvolvimento dos testes visa à criação de dois métodos para cada estrutura: um que verifica o sucesso da operação e outro que simula um caso hipotético de erro, como falha de regra, conexão com o banco de dados ou injeção de dependências na própria classe. Esse tipo de teste ajuda a validar se a regra funciona corretamente em ambos os casos, de sucesso e erro, garantindo uma visão mais abrangente de possíveis situações que possam ocorrer.

Na Figura 39, é possível verificar que, na parte superior da classe, é realizada a injeção das dependências necessárias para prover a execução e disponibilizar os métodos para uso das propriedades. Após isso, as estruturas são preparadas com base no método *arrange*, para, então, serem executadas utilizando o padrão *act* e validadas com o padrão *assert*.

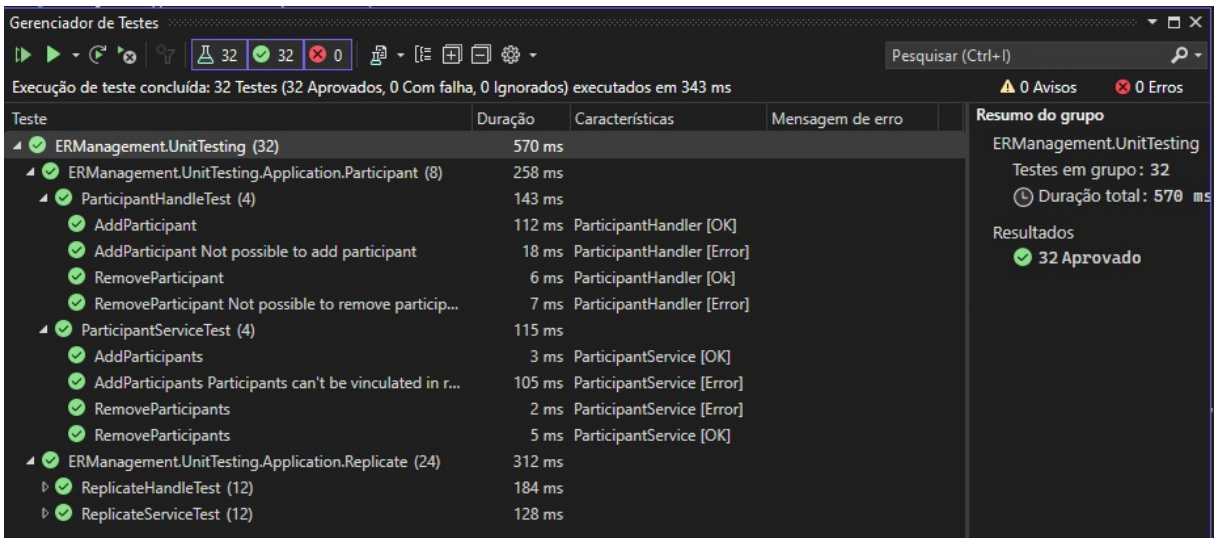
Figura 39 – Estrutura da classe de validação

```
private readonly Mock<IParticipantQueries> _participantQueries = new();
private readonly Mock<IReplicateQueries> _replicateQueries = new();

[Trait("ParticipantService", "OK")]
[Fact(DisplayName = "AddParticipants")]
● | 0 referências | Augusto Fernando Klein, há 3 dias | 1 autor, 1 alteração
public async void CanAddParticipants_ShouldBeOK()
{
    var participants = new List<ParticipantData>{
        new() { ParticipantName = "Part1", ParticipantDirection = "P", ParticipantDatabase = "testeDB", ParticipantTable = "table_test", ParticipantStatement = "INSERT" },
        new() { ParticipantName = "Part2", ParticipantDirection = "P", ParticipantDatabase = "testeDB", ParticipantTable = "table_test", ParticipantStatement = "INSERT" },
    };

    // Arrange
    _replicateQueries.Setup(p => p.ReplicateExist(It.IsAny<string>(), It.IsAny<CancellationToken>())).ReturnsAsync(true);
    _participantQueries.Setup(p => p.ValidateParticipantsExists(participants, It.IsAny<CancellationToken>())).ReturnsAsync(Result.Success);
    _participantQueries.Setup(p => p.ValidateParticipantsNotAssociatedWithReplicate(It.IsAny<string>(), participants, It.IsAny<CancellationToken>())).ReturnsAsync(Result.Success());

    var service = new ParticipantService(_participantQueries.Object, _replicateQueries.Object);

    // Act
    var result = await service.ValidateAddParticipantsReplicate("ReplicateTest", participants, It.IsAny<CancellationToken>());

    // Assert
    Assert.True(result.IsSuccess);
}
```

Fonte: Elaborado pelo autor.

Com as estruturas preparadas, é realizada a execução dos testes, e os resultados são exibidos em uma nova janela da ferramenta de desenvolvimento. Na Figura 40, é possível verificar o resultado da execução dos testes, onde são fornecidas estatísticas de cada execução, como sucesso ou erro, tempo de execução, quantidade de testes e outras informações relevantes para o ambiente.

Figura 40 – Resultado da execução dos processos de teste




Fonte: Elaborado pelo autor.

# 7 RESULTADOS E CONCLUSÕES

O desenvolvimento deste trabalho permitiu observar resultados significativos relacionados aos processos de criação do protótipo de replicação. Durante o processo, alguns requisitos inicialmente levantados sofreram modificações, enquanto outros foram aprofundados, fruto de uma melhor compreensão do ecossistema e da validação com os usuários-chave.

No início, a hipótese sugeria que o protótipo ajudaria a minimizar gargalos nas demandas de manutenção das regras de replicação, permitindo que os responsáveis de cada grupo de trabalho tivessem autonomia para realizar modificações. No entanto, ao longo do projeto, constatou-se que permitir o acesso de diversos usuários a uma ferramenta crítica poderia ser inadequado. Assim, a responsabilidade pela gestão das regras de replicação foi atribuída aos DBAs, enquanto os usuários dos grupos de trabalho receberam apenas permissões de visualização. Essa mudança no escopo visou aumentar a segurança do ambiente corporativo, ao mesmo tempo em que oferecia a visualização necessária para os grupos de trabalho impactados pelas estruturas de replicação.

A criação de um protótipo de replicação agregou valor principalmente para os usuários dos grupos de trabalho, que antes não podiam acessar essas estruturas devido à restrição de acesso, disponível apenas para usuários com maior nível de permissão, como os DBAs. Com o protótipo, tornou-se possível acessar essas informações, permitindo monitorar as operações e agir proativamente antes que outros usuários identificassem eventuais problemas.

A centralização de informações proporcionada pelo protótipo diminuiu a necessidade de acessar múltiplos terminais para verificar as regras e o esquema de tabelas, integrando todas essas informações em uma única plataforma web. Além disso, a usabilidade da ferramenta foi projetada para permitir acesso em qualquer dispositivo com conexão à internet, facilitando a visualização e a gestão das regras de replicação de forma eficiente.

Durante o desenvolvimento, o protótipo enfrentou desafios inesperados, como a complexidade das conexões remotas e uma maior complexidade na criação das regras de replicação. Outro desafio foi o desenvolvimento do *frontend* e *backend*, que se revelou mais complexo do que o previsto por exigir estruturas complementares para a construção. Inicialmente, a arquitetura de microsserviços foi considerada para aumentar a resiliência da API a falhas de conexão. No entanto, percebeu-se que essa abordagem traria complexidade desnecessária ao projeto, optando-se por um mecanismo de *failover* para garantir a operação contínua.

Com o protótipo desenvolvido, os DBAs responsáveis puderam testá-lo e verificar suas funcionalidades, validando as regras e estruturas de replicação criadas. A centralização das informações melhorou a visibilidade para usuários e DBAs, trazendo agilidade e praticidade à gestão das regras. Além disso, a ferramenta pode evoluir para incluir outras funcionalidades, como a sincronização de tabelas e a criação de grupos participantes, consolidando-se como um

software completo para o tratamento e visualização dos dados de replicação.

Atualmente, foi liberada a primeira versão para que os usuários, por grupo de trabalho, possam consultar as estruturas de replicação correspondentes a determinados processos. As operações de manutenção das regras de replicação ainda estão em período de homologação com os DBAs responsáveis, e, quando todos os processos forem homologados, uma nova versão será liberada para o ambiente de produção.

Por fim, este trabalho destaca a importância da gestão de replicação no ambiente corporativo, abordando uma necessidade observada durante os processos de replicação. Espera-se que este estudo contribua para futuras melhorias no ambiente distribuído considerado nesse trabalho e também em outras organizações com características ou problemas similares, proporcionando benefícios tangíveis para organizações que lidam com grandes volumes de dados e processos críticos.

# REFERÊNCIAS

EHSAN, A. *et al.* Restful api testing methodologies: Rationale, challenges, and solution directions. **Applied Sciences**, v. 13, n. 10, 2024. Disponível em: <https://www.mdpi.com/2076-3417/12/9/4369>. Acesso em: 19 may 2024.

EZECHIEL, K. K.; AGARWAL, R.; KAUSHIK, B. Synchronous and asynchronous replication. **Journal of Computer Science and Engineering**, v. 8, n. 2, p. 45–51, 2020. Disponível em: <https://www.researchgate.net/profile/Katembo-Ezechiel/publication/330485205_Synchronous_and_Asynchronous_Replication/links/5c874a94458515b59e4549a1/Synchronous-and-Asynchronous-Replication.pdf>. Acesso em: 20 abr. 2024.

IBM. **Transaction management**. 2022. Disponível em: <https://www.ibm.com/docs/en/informix-servers/14.10?topic=management-transaction>. Acesso em: 12 maio 2024.

____. **What is the logical log?** 2023. Disponível em: <https://www.ibm.com/docs/en/informix-servers/14.10?topic=log-what-is-logical>. Acesso em: 14 abr 2024.

IBM Corporation. **IBM Informix Enterprise Replication Guide**. [S.l.], 2010. Version 11.70. SC27-3539-00. © Copyright IBM Corporation 1996, 2010. US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp. Disponível em: <https://www.oninit.com/manual/informix/117/documentation/ids_erp_bookmap.pdf>. Acesso em: 21 mar. 2024.

LI, X. *et al.* Context aware middleware architectures: Survey and challenges. **Research Center on Software Technologies and Multimedia Systems for Sustainability (CITSEM)**, Campus Sur UPM, Ctra. Valencia, Km 7, 28031 Madrid, Spain, 2015. Received: 30 April 2015 / Accepted: 13 August 2015 / Published: 20 August 2015. Disponível em: <https://www.mdpi.com/1424-8220/15/8/20570>.

LOTFY, A. E. *et al.* A middle layer solution to support acid properties for NoSQL databases. **Future Generation Computer Systems**, v. 38, 2014. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1319157815001044>. Acesso em: 27 abr. 2024.

MICROSOFT. **Um tour por C**. 2024. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/>. Acesso em: 14 abr 2024.

MOIZ, S. A. *et al.* Database replication: A survey of open source and commercial tools. **International Journal of Computer Applications**, v. 13, n. 6, January 2011. ISSN 0975-8887. Disponível em: <https://www.ijcaonline.com/volume13/number6/pxc3872469.pdf>. Acesso em: 16 abr 2024.

OLAN, M. Unit testing: Test early, test often. In: RICHARD STOCKTON COLLEGE, COMPUTER SCIENCE AND INFORMATION SYSTEMS. **Proceedings of the 16th Annual Consortium for Computing Sciences in Colleges, Northeastern Conference**. Pomona, NJ, 2001. p. 253–260. *Presented at the conference. Disponível em: <https://www.researchgate.net/profile/Michael-Olan/publication/255673967_Unit_testing_Test_early_test_often/links/5581783608aea3d7096ff00c/Unit-testing-Test-early-test-often.pdf>.

ORACLE. **MySQL Replication**. 2024. Disponível em: <https://dev.mysql.com/doc/refman/8.3/en/replication.html>. Acesso em: 14 abr 2024.

____. **MySQL Replication FTID Method**. 2024. Disponível em: <https://dev.mysql.com/doc/refman/8.3/en/replication-gtids.html>. Acesso em: 14 abr 2024.

____. **MySQL Replication Type Statements**. 2024. Disponível em: <https://dev.mysql.com/doc/refman/8.3/en/replication-formats.html>. Acesso em: 14 abr 2024.

O'REILLY MEDIA INC. **What React Is and Why It Matters**. 2024. Disponível em: <https://www.oreilly.com/library/view/what-react-is/9781491996744/ch01.html>. Acesso em: 11 may 2024.

POSTGRESQL. **PostgreSQL Logical Replication**. 2024. Disponível em: <https://www.postgresql.org/docs/current/logical-replication.html>. Acesso em: 16 abr 2024.

____. **PostgreSQL Replication**. 2024. Disponível em: <https://www.postgresql.org/docs/current/runtime-config-replication.html>. Acesso em: 14 abr 2024.

____. **What is the logical log?** 2024. Disponível em: <https://www.postgresql.org/docs/current/logical-replication-conflicts.html>. Acesso em: 16 abr 2024.

REFACTORING.GURU. **Command Design Pattern**. 2024. Disponível em: <https://refactoring.guru/design-patterns/command>. Acesso em: 12 out 2024.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Sistema de Banco de Dados**. 7th. ed. [s.n.], 2020. Revisão Técnica: Daniel Sado Menasché, Ph.D., University of Massachusetts Amherst. Professor Associado no Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro (UFRJ). Disponível em: <https://integrada.minhabiblioteca.com.br/reader/books/9788595157552/epubcfi/6/10[%3Bvnd.vst.idref%3Dcopyright]!/4>. Acesso em: 30 mar. 2024.

THAKUR, R. N.; PANDEY, U. S. The role of model-view controller in object oriented software development. **Nepal Journal of Multidisciplinary Research (NJMR)**, v. 2, n. 2, p. 1, June 2019. ISSN 2645-8470. Corresponding Author: Ram Naresh Thakur, PhD Scholar, Mewar University, Chittorgarh, Rajasthan, India; U S Pandey, Professor, University of Delhi, India. Disponível em: <https://www.researchgate.net/profile/R-N-Thakur/publication/337128796_The_Role_of_Model-View_Controller_in_Object_Oriented_Software_Development/links/64f98d5b8ea93c20d224279f/The-Role-of-Model-View-Controller-in-Object-Oriented-Software-Development.pdf>.

VERCEL. **Next.js**. 2024. Disponível em: <https://nextjs.org/docs>. Acesso em: 14 abr 2024.

VERMA, P. A brief study of middleware technologies: Programming applications and management systems. **Research Gate**, 2022. A* Author's affiliation and further details not provided. Disponível em: <https://www.researchgate.net/publication/360506335_A_Brief_Study_of_Middleware_Technologies_Programming_Applications_and_Management_Systems>.

WEI, C. *et al.* How do developers structure unit test cases? an empirical study from the "aaa"perspective. **IEEE Transactions on Software Engineering**, IEEE, v. 50, n. 5, p. 2301–2316, 2024. Disponível em: <https://arxiv.org/pdf/2407.08138>.

WEINBERG, P.; GROFF, J.; OPPEL, A. **SQL: The Complete Reference**. 3rd. ed. McGraw-Hill, 2010. Disponível em: <https://ci-ceit.edu.ck/wp-content/uploads/2021/01/sql-the-complete-reference-third-edition-sep-2009.pdf>. Acesso em: 01 maio 2024.

WOODHULL, A. S. T. e A. S. **Sistemas Operacionais, Projeto e Implantação**. 3rd. ed. [s.n.], 2008. Versão impressa desta obra. Disponível em: <https://integrada.minhabiblioteca.com.br/reader/books/9788577802852/pageid/85>. Acesso em: 27 abr. 2024.