UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Tecnologia

Leandro Augusto Fabri Pereira

Reformulação de arquitetura de comunicação para uma aplicação Web

Limeira
2022

Leandro Augusto Fabri Pereira

Reformulação de arquitetura de comunicação para uma aplicação Web

Monografia apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Guilherme Palermo Coelho

Este exemplar corresponde à versão final da Monografia defendida por Leandro Augusto Fabri Pereira e orientada pelo Prof. Dr. Guilherme Palermo Coelho.

Limeira
2022

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Faculdade de Tecnologia
Luiz Felipe Galeffi - CRB 8/10385

P414r

Pereira, Leandro Augusto Fabri, 1996-
Reformulação de arquitetura de comunicação para uma aplicação web / Leandro Augusto Fabri Pereira. – Limeira, SP : [s.n.], 2022.

Orientador: Guilherme Palermo Coelho.
Trabalho de Conclusão de Curso (graduação) – Universidade Estadual de Campinas, Faculdade de Tecnologia.

1. Arquitetura da informação. 2. Banco de dados. I. Coelho, Guilherme Palermo,1980-. II. Universidade Estadual de Campinas. Faculdade de Tecnologia. III. Título.

Informações adicionais, complementares

Palavras-chave em inglês:
Information architecture
Databases
Titulação: Bacharel
Banca examinadora:
Guilherme Palermo Coelho [Orientador]
André Leon Sampaio Gradvohl
Plinio Roberto Souza Vilela
Data de entrega do trabalho definitivo: 23-11-2022

FOLHA DE APROVAÇÃO

Abaixo se apresentam os membros da comissão julgadora da sessão pública de defesa de dissertação para o Título de Bacharel em Sistemas de Informação, a que se submeteu o aluno Leandro Augusto Fabri Pereira, em 23 de novembro de 2022 na Faculdade de Tecnologia – FT/UNICAMP, em Limeira/SP.

Prof. Dr. Guilherme Palermo Coelho
FT/UNICAMP- Presidente da Comissão Julgadora

Prof. Dr. André Leon Sampaio Gradvohl
FT/UNICAMP

Prof. Dr. Plínio Roberto Souza Vilela
FT/UNICAMP

Ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Trabalhos de Conclusão de Curso e na Secretaria de Graduação da FT.

Agradecimentos

Sou grato a todos os envolvidos na criação deste trabalho, em especial a minha família e amigos que sempre estiveram ao meu lado em momentos difíceis, e também ao meu orientador Guilherme Palermo que sempre esteve disponível para me ajudar e oferecer ideias para o meu trabalho, sendo sempre muito solícito e prestativo.

Resumo

Este projeto teve como objetivo analisar a arquitetura de comunicação do *Conecta*, que é um aplicativo Web desenvolvido para ser uma extensão de sistemas *Enterprise Resource Planning* (ERP) desktop compatíveis para administração de condomínios, imobiliárias, recursos humanos, dentre outros. O *Conecta* permite que seus usuários consigam gerar relatórios gerenciais acerca de informações contidas no ERP, e, para este fim, foi desenvolvido um processo de solicitação, no qual um usuário do sistema solicita uma informação, e essa mesma é processada pelo ERP no ambiente em que está instalado (um servidor rodando Windows). O sistema atual possui problemas intrínsecos ao seu processo de solicitação, pois todo o tráfego de dados passa por um banco de dados MySQL e o mesmo não é desenhado para esse fim, causando um problema de lotação no armazenamento do servidor em que está operando. Tal problema não parece estar relacionado a falhas no próprio SGBD (Sistema Gerenciador de Banco de Dados), mas sim à abordagem para concretização de solicitações. Neste trabalho foi proposta uma correção para este problema através da implementação de *message brokers* ao invés de SGBDs. *Message brokers* são programas desenhados para a troca de informações entre aplicações e serviços, sendo proposto então a adequação do sistema Conecta a essa nova arquitetura, substituindo a anterior. Como resultado, foram desenvolvidas aplicações que usam o *message broker NATS* para este processo de solicitação-resposta, além de *Application Programming Interfaces (APIs)* que servem tanto para o novo processo de solicitação, quanto para comparativos entre o método antigo e o novo, em termos de performance e tráfego de informações. Um estudo comparativo deixou evidente a melhora da velocidade da aplicação *Conecta* como um todo, em cerca de 50%, e eliminando o problema causado pela sobrecarga de dados do método anterior de solicitações.

Palavras-chave: arquiteturas de comunicação; *message brokers*; sistemas gerenciadores de banco de dados.

Abstract

This project aimed to analyze the communications architecture of *Conecta*, which is a Web application developed to be an extension of compatible Enterprise Resource Planning (ERP) systems, to manage apartment complexes, real estates, and human resources among others. It allows its users to generate management reports and other informations in and around the ERP system, and, to that end, it comprises a request process that is processed by the ERP in the environment that it is installed on (a Windows machine). The current system has intrinsic problems within its request process, because all of the data traffic has to go through a MySQL server which is not designed for this use, causing a data storage overflow within the server that it's running in. This problem is not related to any MySQL bugs, being a design flaw in the fulfillment of the request process. In this document, it is proposed a solution to this problem by implementing *message brokers* instead of a DBMS (Database Management System), which are programs designed to manage information exchange between application and services in mind. As a result, it was developed applications that implement the NATS messaging system to replace the old ones implemented in *Conecta*, and also an Application Programming Interface (API) to serve the new request process and to compare between the old and new methods. Comparative experiments were made and, in terms of performance and information exchange traffic, it became clear the overall speed increase of the new proposal as a whole, 50% on average, and the elimination of the problem caused by data overhead in the previous request method.

Keywords: Communications architecture, message brokers, database management systems.

Lista de Figuras

2.1 Diagrama simplificado de funcionamento do *Conecta*. 12
2.2 Página principal (*frontend*) do Conecta para um Usuário Síndico. 14
2.3 Diagrama da arquitetura utilizada atualmente no *Conecta*. 14
2.4 Diagrama de arquitetura do ERP. 16

3.1 Arquitetura do MySQL, adaptado de (CHIN; ANDREW ROSZKO, 2002). . . . 20
3.2 Estrutura da tabela após simulação. 21
3.3 Estrutura da tabela após exclusão do campo resposta. 22
3.4 Tamanho da tabela após otimização. 22
3.5 Tamanho da tabela em disco após otimização. 22

4.1 Arquitetura geral de um message broker. 25
4.2 Arquitetura de mensageria do NATS. 26
4.3 Arquitetura da aplicação servidor. 28
4.4 Arquitetura da API Rest. 30
4.5 Tela de login. 32
4.6 Tela de solicitações, com a consulta de dados de um condomínio fictício. . . . 33
4.7 Tela de testes. 35

Sumário

1 Introdução **10**

2 O sistema Conecta **12**
2.1 Visão geral do sistema . . . 13
2.2 Processo solicitação-resposta . . . 13
2.3 Processamento de solicitações . . . 15

3 Problema atual **18**
3.1 Problema ou funcionalidade? . . . 18
3.1.1 Arquitetura do MySQL . . . 19
3.1.2 Estrutura do InnoDB . . . 20
3.1.3 Otimização de tabela . . . 20
3.2 Exemplo na prática . . . 21
3.3 Outros problemas não relacionados . . . 23

4 Solução proposta **24**
4.1 *Message Brokers* . . . 24
4.2 *Backbone* escolhido . . . 26
4.3 Novo modelo solicitação-resposta . . . 27
4.3.1 Microsserviços . . . 27
4.3.2 Aplicação servidor . . . 28
4.3.3 API Rest . . . 29
4.4 *Frontend* da aplicação . . . 31
4.4.1 Página de registro . . . 31
4.4.2 Página de solicitações . . . 32
4.4.3 Página de testes . . . 34

5 Conclusões **36**

Referências bibliográficas **37**

A NestJS **38**

B ReactJS **39**

Capítulo 1

Introdução

Este trabalho teve como objetivo prover uma análise de uma aplicação Web, o *Conecta* [1], bem como propor uma correção para um problema de comunicação observado em sua implementação. Para contextualizar, a aplicação *Conecta* é uma extensão de outros sistemas *Enterprise Resource Planning (ERP)*, sendo voltado a usuários que queiram acessar dados, via Web, pertinentes aos ERPs de cada cliente, a partir de informações contidas no sistema da administradora, que também é um sistema de ERP utilizado para o seus trabalhos e processos diários. O *Conecta* integra dois sistemas:

1. Condomínios (CD);
2. Imobiliária (LC).

Cada um de seus sistemas têm suas particularidades, mas ambos geram relatórios gerenciais que podem ser úteis para terceiros, como síndicos ou condôminos, no caso do sistema de condomínios. Dentre os processos abrangidos por cada um, os mais utilizados pelo *Conecta* incluem:

- Prestação de contas;
- Cadastramento de informações;
- Geração e monitoramento de cobranças;
- Geração de relatórios e resumos em geral;
- Consulta dos dados.

[1] Um produto comercial da D&G Sistemas.

O problema do *Conecta* está em sua arquitetura de comunicação, que mesmo em seu processo ideal, gera arquivos com tamanhos excessivos que, com o tempo, tornam-se inviáveis para a continuação de seu funcionamento.

Este problema pode ser resolvido pelo uso de outra arquitetura de comunicação, utilizando-se de *message brokers* para lidar com o ir e vir de dados dentro do sistema, solução esta adotada aqui.

Neste trabalho foi construída uma aplicação de *frontend* em *ReactJS* que funciona de maneira análoga às apresentadas neste documento. Mas o *frontend* atual do sistema *Conecta* (desenvolvido em PHP) ainda não foi modificado, por estar em ambiente produção.

Esta monografia está organizada da seguinte forma: No Capítulo 2 é apresentado o sistema *Conecta* em detalhes. No Capítulo 3 discute-se o problema que ocorre atualmente no sistema, enquanto que no Capítulo 4 é apresentada a solução proposta. Por fim, no Capítulo 5 são apresentadas as conclusões deste trabalho.

Capítulo 2

O sistema Conecta

No geral, pode-se conceitualizar o sistema *Conecta* com dois atores: O Usuário e a Administradora, cabendo à administradora alimentar o ERP com informações, executar processos e utilizá-lo para gerenciar recursos de seus clientes. Já o usuário busca ter acesso a serviços prestados pela administradora, tais como tirar segundas-vias de boleto e verificar a saúde financeira de seu imóvel ou condomínio, dentre outras. Isso é feito pela Web por meio de uma aplicação atualmente escrita em *PHP*, através de um processo de solicitação-resposta, como ilustrado no diagrama da Figura 2.1.

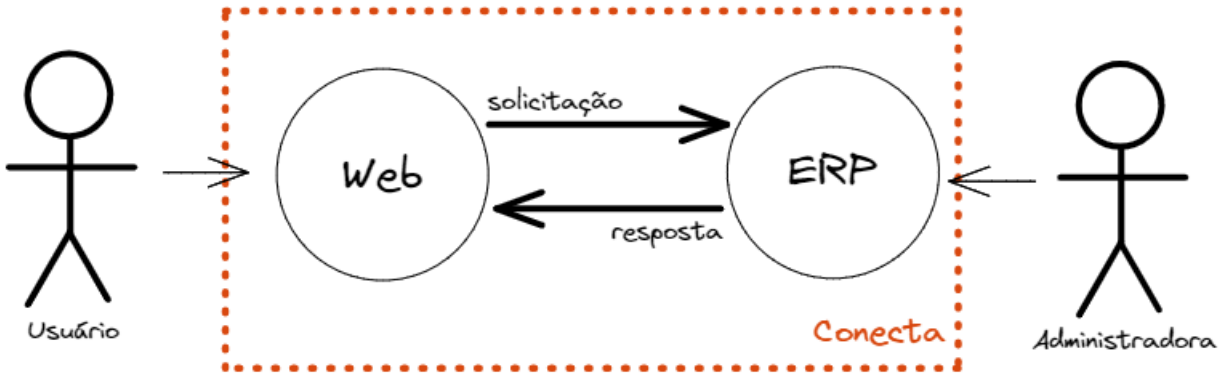

Figura 2.1: Diagrama simplificado de funcionamento do *Conecta*.

Como observado na Figura 2.1, os dois atores principais atuam no contexto que tem como fim demonstrar a interligação de ambos, tendo de um lado o usuário do sistema *Conecta*, requisitando informações e executando operações em seu condomínio, e de outro lado a Administradora, que utiliza-se dos sistemas ERP compativeis para a gerência de seus condomínios, alimentando-o com informações que não necessariamente são interessantes

para si, mas sim para os usuários. A aplicação *Conecta* então visa automatizar este processo através de trocas de mensagens de solicitação e resposta.

2.1 Visão geral do sistema

Uma vez autenticado no *Conecta*, o Usuário é colocado frente a diversas opções, dependendo de seu nível de acesso e processo envolvido (CD ou LC):

- Administradora (CD, LC);
- Condômino (CD);
- Síndico (CD);
- Porteiro (CD);
- Inquilino (LC);
- Proprietário (LC).

Cada um dos níveis de acesso permite a execução de operações distintas. Como exemplo, um usuário fictício de nível '*Síndico*' tem acesso à tela dada na Figura 2.2, que ilustra a página principal e menu de opções do sistema *Conecta*

Na sua maioria, as solicitações de usuários são relatórios gerenciais ou boletos em PDF, tendo ainda outras opções implementadas no sistema que não são pertinentes a este trabalho devido à não utilização do processo de solicitação-resposta tratado aqui.

2.2 Processo solicitação-resposta

Quando alguma informação é solicitada pelo Usuário através de uma opção do menu, uma nova solicitação é gerada para processamento no servidor juntamente com o ERP da Administradora (através de seu *runtime*), como ilustrado no diagrama da Figura 2.3.

O diagrama da Figura 2.3 detalha um pouco melhor o processo todo, em que cada solicitação gera um novo registro dentro de uma tabela de banco de dados MySQL. Por outro lado, haverá uma aplicação no servidor do cliente (Administradora) verificando se novos registros (solicitações) foram adicionados àquela tabela. Caso afirmativo, tal solicitação é

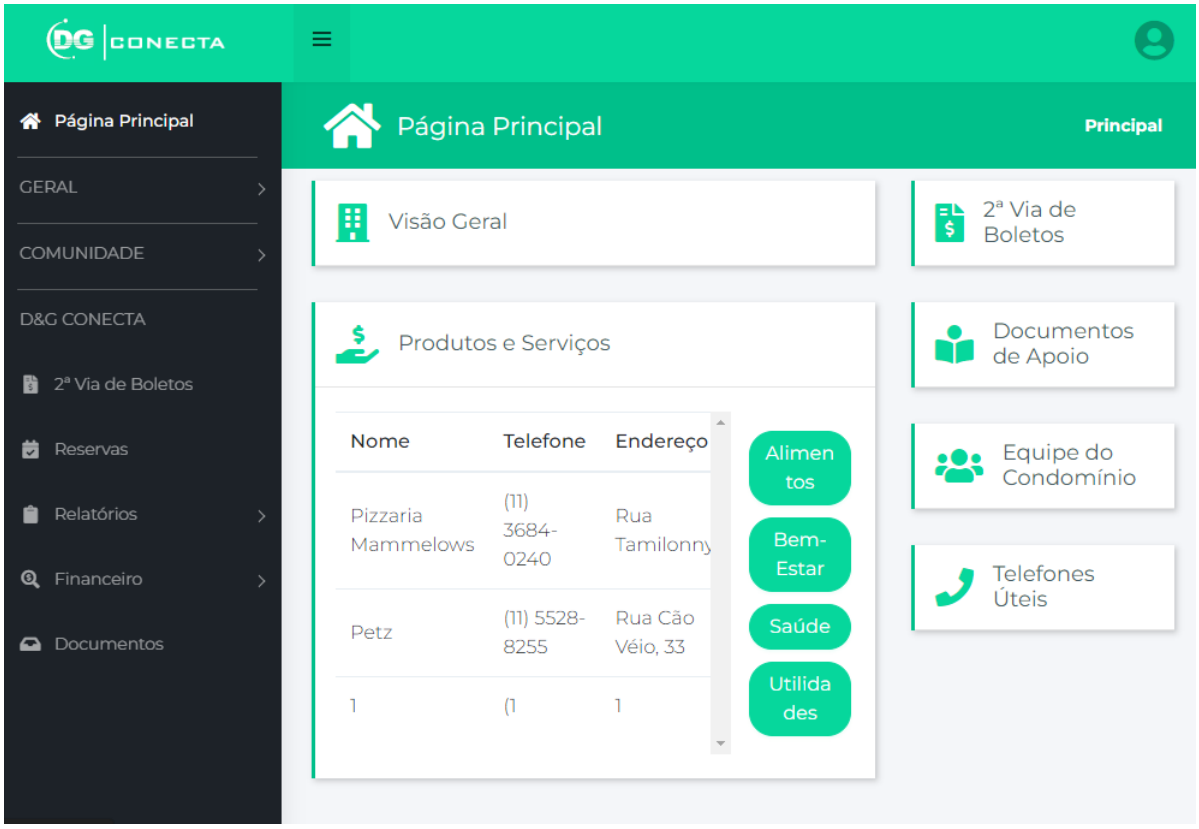

Figura 2.2: Página principal (*frontend*) do Conecta para um Usuário Síndico.

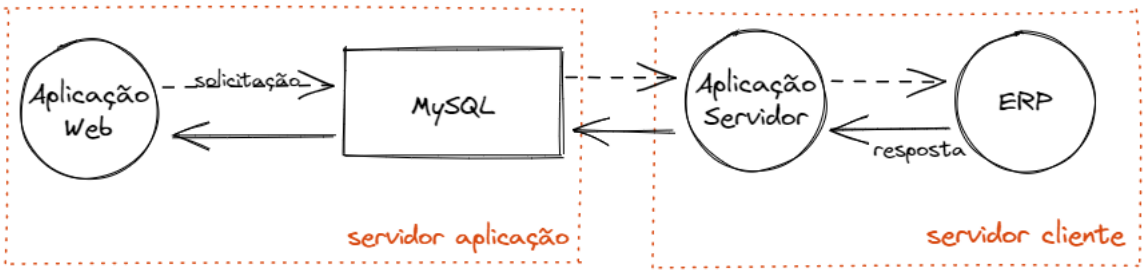

Figura 2.3: Diagrama da arquitetura utilizada atualmente no *Conecta*.

processada pelo ERP e então devolvida a resposta ao mesmo registro, que por sua vez é exibido na aplicação Web no formato especificado.

Pode-se enumerar a sequência de passos necessária para um usuário do *Conecta* concluir seu objetivo no sistema:

1. O usuário se autentica;
2. É escolhida uma opção do menu;
3. Uma solicitação é gerada no banco de dados (SGBD) pelo aplicativo web;
4. A aplicação servidor, que verifica o SGBD periodicamente, recebe a nova solicitação e grava em disco os parâmetros para processamento pelo ERP;
5. Um executável, o *runtime* do Cobol, é instanciado para carregar o ERP e processar a solicitação;
6. A aplicação servidor, que agora está então verificando periodicamente pelo arquivo 'DGTRARES.DEG', aguarda a resposta do ERP;
7. A aplicação servidor carrega o campo 'DG05COM_RESPOSTA' ou 'DOC' do mesmo registro no banco de dados com o conteúdo do arquivo 'DGTRARES.DEG';
8. A aplicação web, que está periodicamente verificando se há algo no campo 'DG05COM_RESPOSTA' ou DOC', gera a visualização para o usuário quando se encontra a resposta;
9. O usuário faz outra solicitação (volta ao passo 2); ou
10. O usuário sai do sistema

2.3 Processamento de solicitações

Detalhando um pouco melhor o processamento de solicitações, assim que uma nova solicitação é identificada, um subsistema do ERP (aplicativo servidor) lê os parâmetros dessa transação e grava-os em um arquivo denominado *DGTRAREQ.DEG* para processamento, no diretório pré-especificado por arquivos de configuração.

O ERP em si é quem faz o processamento da solicitação e grava a resposta. Processo melhor ilustrado no diagrama da Figura 2.4. O ERP é um conjunto de subprogramas escritos em Cobol, precisando executar o seu processo de *runtime* dentro do ambiente *Windows* para o carregamento do programa *DGXC00A*, que por sua vez irá efetuar o processamento dessa transação de fato, e então gravará o arquivo *DGTRARES.TXT* no diretório de saída. Tal arquivo será levado como resposta pelo aplicativo servidor de volta para a aplicação web através do registro de solicitação (no SGBD) que fora gerado pelo mesmo.

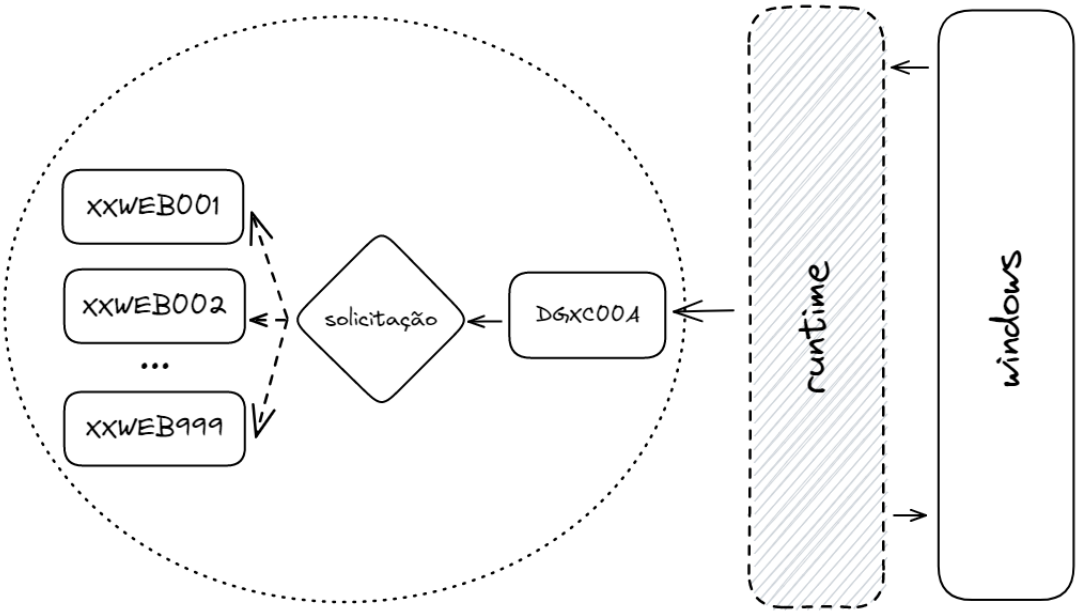

Figura 2.4: Diagrama de arquitetura do ERP.

Após o processamento, a resposta é então devolvida para geração da visualização para o usuário pelo aplicativo Web, podendo ser:

- Geração de um PDF com a resposta;
- Geração de gráficos;
- Segunda-via de boletos (outro relatório PDF);
- Reserva de ambientes (um calendário).

O processo de solicitação é linear, pois processa arquivos um por um, ou seja, onde é permitida a leitura de apenas um arquivo de transação por vez para processamento, sendo o atual gargalo para a velocidade da aplicação como um todo.

Num processo ideal, não haveria nenhum problema na utilização de registros MySQL para transacionar uma solicitação, mas devido a mecanismos internos do SGBD que serão melhor

entendidos no próximo capítulo, acabam sendo criados arquivos com tamanhos indesejáveis, que tornam-se um problema em ambientes com armazenamento limitado.

Atualmente, no servidor em que está hospedada tanto a aplicação *Conecta* quanto o servidor MySQL, o armazenamento total é de 20GB, e caso não seja feita nenhuma manutenção acerca deste problema discutido até então, o mesmo enche-se em cerca de dois meses de operação (aproximadamente dez mil solicitações), sendo necessária a reconstrução da tabela para reassegurar este espaço que fora ocupado pelo motor InnoDB.

Capítulo 3

Problema atual

O problema atual do aplicativo *Conecta* surge ao se utilizar do mesmo registro para devolver a resposta da solicitação, já que esta resposta pode ser densa em termos de tamanho de arquivo. Pode-se atribuir então o problema à escolha e abordagem da arquitetura de comunicação. Mesmo com a exclusão dos dados do campo de resposta, o arquivo de dados gerado internamente pelo MySQL com o motor de armazenamento InnoDB, utilizado pelo *Conecta*, não diminui em tamanho. Com o tempo, isso vai gerando uma sobrecarga neste arquivo que não pára de crescer. Mais adiante, na Seção 3.2, será entendido melhor o porquê disso.

3.1 Problema ou funcionalidade?

Essa escolha de arquitetura utilizada pelo *Conecta* gera principalmente um *problema de sobrecarga de armazenamento*, que não é exatamente uma falha no banco de dados em si, mas sim na escolha desta tecnologia - Sistema Gerenciador de Banco de Dados (SGBD) - para a transmissão das informações. Esta afirmação é evidenciada nos comentários de submissões de problemas do MySQL (ELLSWORTH, 2003), em que constam diversas mensagens acerca do mesmo problema. Tais mensagens foram respondidas pelos próprios desenvolvedores, que indicam que tal comportamento é uma funcionalidade documentada.

Olhando um pouco mais a fundo, o problema gerado em decorrência desta escolha de arquitetura pode ser entendido quando melhor analisado o mecanismo de funcionamento do SGBD em questão, o MySQL, com o motor de armazenamento InnoDB.

Conceitualmente, pode-se afirmar que um sistema gerenciador de banco de dados deste tipo é projetado para armazenar uma coleção de registros de forma organizada, permitindo a subsequente recuperação destes mesmos registros conforme necessário e de maneira interativa ou programática.

Analisando o caso de uso de um sistema gerenciador de banco de dados, que pode ser definido como *"um sistema de software que permite usuários definir, criar, manter e controlar acesso ao banco de dados"* (CONNOLLY THOMAS M.; BEGG, 2014), entende-se que são criados arquivos em disco (em que o servidor do SGBD está instalado) para o armazenamento de tais registros e seus respectivos índices. Internamente estes arquivos são utilizados para o armazenamento e recuperação de tais registros e índices, sendo controlados pelos *motores* dos bancos de dados.

3.1.1 Arquitetura do MySQL

Segundo Chin; Andrew Roszko (2002), de maneira geral o MySQL possui uma arquitetura definida em três camadas, como ilustrado na Figura 3.1

- Camada de aplicação, em que há interação com o usuário através de *queries*.
- Camada lógica, em que se aplicam regras internas de cada motor de armazenamento escolhido, como MyISAM, InnoDB, etc.
- Camada física, onde se armazena fisicamente os dados, podendo ser em arquivos, em memória, ou algum outro meio dependendo do motor de armazenamento.

Precisamos olhar diretamente para o funcionamento do gerenciador de armazenamento, evidenciado na Figura 3.1, que está contido na camada física para compreender o que acontece. Pode-se afirmar que o problema observado no *Conecta* é devido ao *Log Management* do InnoDB, que grava cada novo registro SQL em um arquivo pré-alocado dentro do arquivo de tabela (*tablespace*). Além disso, estes arquivos não são reanalisados automaticamente durante o seu tempo de vida, apenas através do comando SQL *OPTIMIZE TABLE*, tendo diferentes implementações dependendo da escolha do motor de armazenamento.

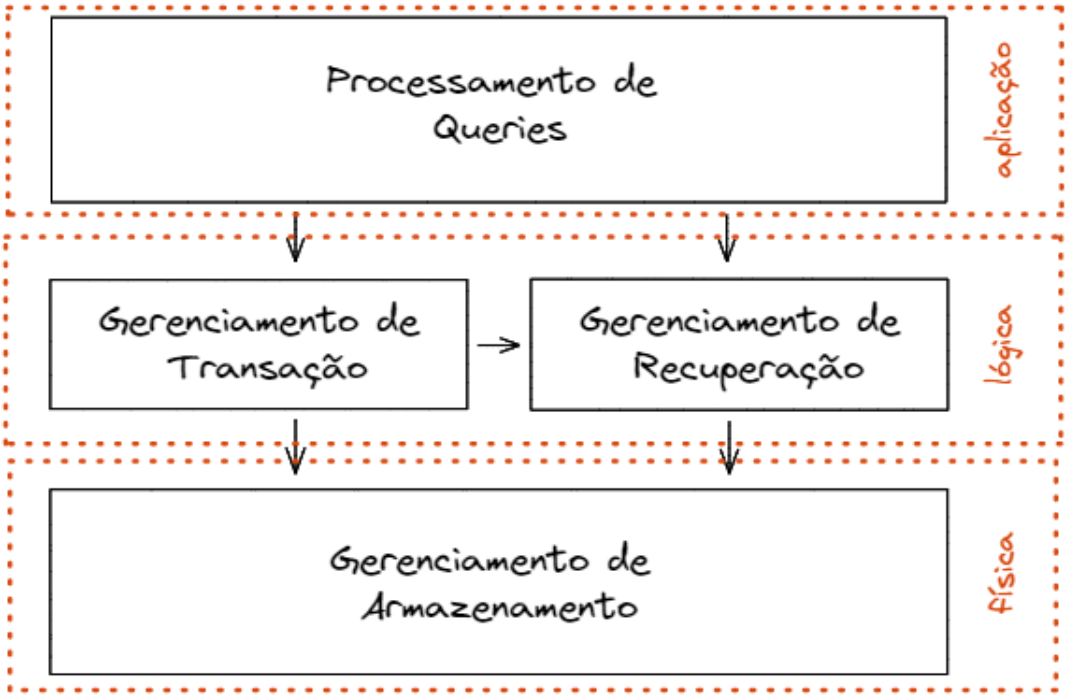

Figura 3.1: Arquitetura do MySQL, adaptado de (CHIN; ANDREW ROSZKO, 2002).

3.1.2 Estrutura do InnoDB

No caso do InnoDB, as tabelas e índices são armazenados em arquivos .idb, seu *tablespace*, contendo os índices e conteúdo da tabela, e o problema parece ocorrer devido à escolha dos desenvolvedores de implementar um sistema de prealocação deste arquivo, que parece aumentar à cada registro inserido, mas não diminuir quando algum dado do registro é removido (CHIN, 2002).

3.1.3 Otimização de tabela

Ao executar o *OPTIMIZE TABLE* na tabela de comunicações *DG05COM* do *Conecta*, devido aos seus mecanismos internos, o InnoDB atrela este comando a um *ALTER TABLE*, que irá reconstruir a tabela do zero, o que pode levar muito tempo devido à quantidade de registros e limitações do InnoDB em otimizar tabelas (MYSQL.COM, 2022). Além disso, muitas vezes tal otimização falha devido à limitação de 20GB de armazenamento do servidor em que a aplicação MySQL está sendo executada.

3.2 Exemplo na prática

Em termos operacionais, não há nada de errado na metodologia do motor de armazenamento InnoDB, mas fica claro que as características mencionadas até aqui não foram levadas em consideração na elaboração do sistema. Na prática, o Conecta evidencia este problema, portanto foi feito um teste em que se trafegou cerca de 512MB de solicitações utilizando-se de uma tabela denominada *DG05COM*, inicialmente vazia.

Tal teste foi uma simulação de alguns meses de uso da aplicação, gerando 11.051 registros de solicitação no processo, e demonstra bem o que acontece. A Figura 3.2 demonstra o seu tamanho após esta simulação.

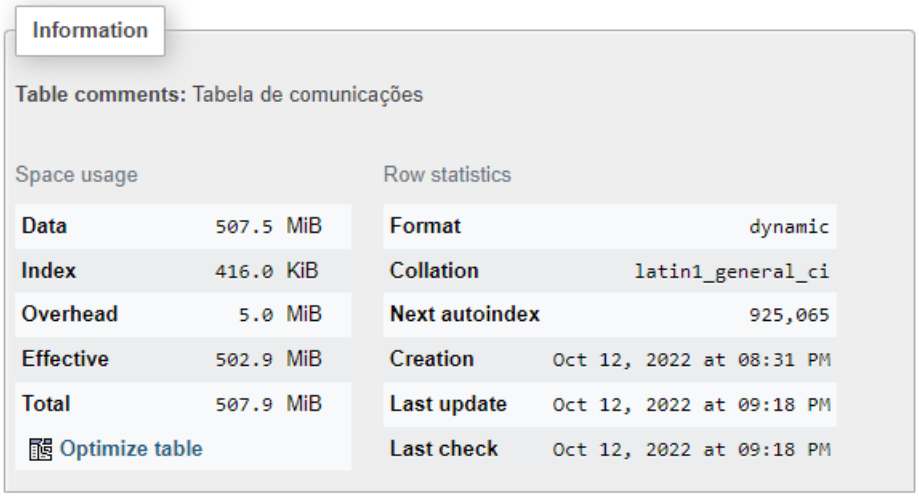

Figura 3.2: Estrutura da tabela após simulação.

Como evidenciado em *Space usage*, o total dos arquivos gerados pelo InnoDB ocupam cerca de 500MB, devido ao campo de resposta de cada solicitação ainda estar alimentado com o resultado do processamento. Portanto, fora executado um comando para limpar o campo (deixá-lo nulo) de todos estes registros, sendo o resultado obtido dado na Figura 3.3.

Conforme observado, mesmo excluindo o conteúdo do campo resposta, o tamanho de *Data* continua o mesmo, tendo sido gerado até mesmo mais *overhead* (cerca de 71MB), não diminuindo de forma alguma o espaço utilizado pelos arquivos de seu *tablespace*.

Uma possível solução para este problema está na otimização da tabela após a remoção do conteúdo da resposta. Mas nem sempre o comando é bem-sucedido, retornando mensagens como *"A tabela está cheia"*. Isso é por conta da ausência de mecanismos de otimização no InnoDB, o que faz com que toda a tabela seja recriada ao utilizar-se este comando, estando sujeito a limitações de memória e de outros recursos da máquina servidor.

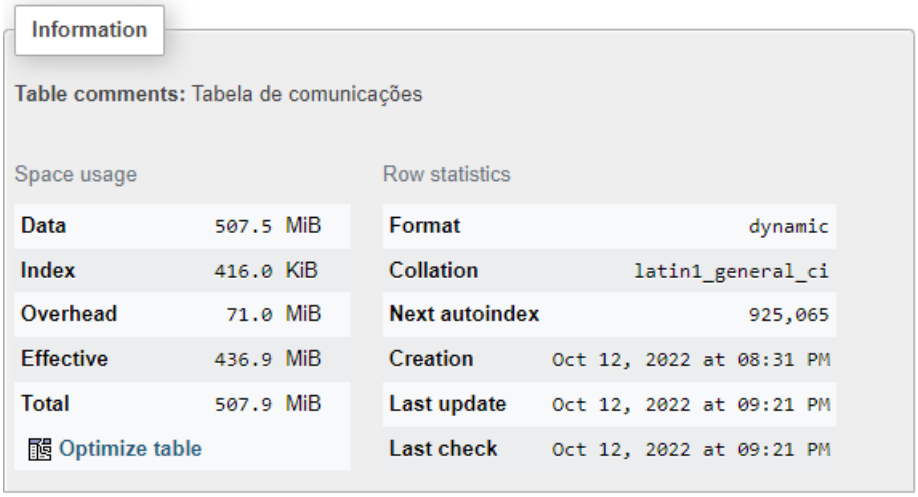

Figura 3.3: Estrutura da tabela após exclusão do campo resposta.

Supondo que não haja esses problemas no processo de otimização, e o comando seja executado corretamente, ainda assim os resultados podem ser inconsistentes, como observado na Figura 3.4, em que o *phpMyAdmin* acusa o tamanho total de seu *tablespace* em cerca de 1.9MB

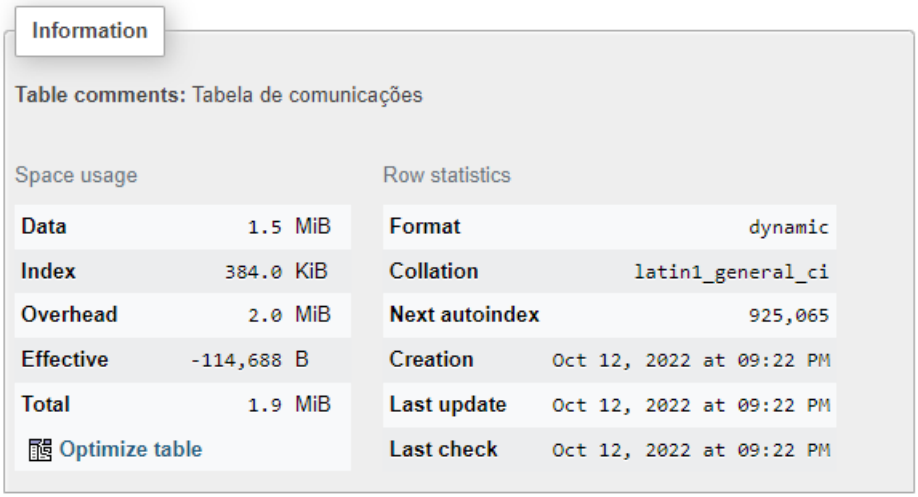

Figura 3.4: Tamanho da tabela após otimização.

No entanto, se verificarmos o tamanho efetivo em disco dos arquivos, através do comando *ls -lah*, com seu resultado evidenciado na Figura 3.5, obtemos aproximadamente 7MB, o que já é uma melhora significativa.

```
-rw-rw---- 1 mysql mysql 2.7K Oct 12 21:22 DG05COM.frm
-rw-rw---- 1 mysql mysql 7.0M Oct 12 21:22 DG05COM.ibd
```

Figura 3.5: Tamanho da tabela em disco após otimização.

Fica claro que ao utilizar um banco de dados desse tipo, que não possui em seus casos de uso a utilização para transmissão de dados, como está acontecendo atualmente no *Conecta*, há o armazenamento de informações consideradas ‘inúteis‘ para o contexto da aplicação, gerando um uso desnecessário de espaço em disco que só é recuperado após a reconstrução da tabela como um todo, tarefa essa que é custosa em termos de recursos computacionais necessários, além de ser um processo periódico a mais para a manutenção do sistema como um todo.

3.3 Outros problemas não relacionados

Também vale a pena ressaltar que devido a esta metodologia de solicitação-resposta, o aplicativo servidor precisa ficar fazendo *queries* frequentes e periódicas para verificar se há uma nova solicitação no banco de dados, levando a um desperdício de processamento e tráfego de dados tanto no servidor do banco de dados quanto no do cliente.

Por outro lado, utilizando-se o modelo proposto neste trabalho, o aplicativo servidor só é acionado quando há uma nova mensagem destinada à ele, evitando tais desperdícios. A solução proposta aqui será discutida no próximo capítulo.

Capítulo 4

Solução proposta

Uma solução para o problema discutido no capítulo anterior está em mudar conceitualmente o processo de requisição-resposta do sistema *Conecta*, substituindo o banco de dados relacional (SGBD) por outra tecnologia que lide com o conceito de *mensageria* (BRITO, 2019) - troca de mensagens gerenciadas. Dessa forma, evita-se a geração de arquivos desnecessários, dando preferência à utilização apenas de memória volátil (RAM), e sem que haja o armazenamento de informações acerca da solicitação.

4.1 *Message Brokers*

Uma forma de mensageria se dá através de *message brokers*, que são servidores de mensagens que garantem o enfileiramento das mensagens recebidas em um *buffer* até que ocorra o seu processamento. São *softwares* que permitem que outras aplicações ou serviços comuniquem entre si para troca de informações. Isso é atingido através do estabelecimento de um protocolo para essa comunicação, o que facilita a sua integração entre aplicações desenvolvidas por diferentes linguagens de programação e plataformas.

Os *message brokers* estão contidos dentro da camada de *middleware* das aplicações, ou seja, não é de sua responsabilidade ter nenhuma lógica de negócios, mas sim viabilizar a transferência entre ‘Produtores‘ e ‘Consumidores‘ através de uma fila interna (Figura 4.1), permitindo a padronização do fluxo de mensagens entre os componentes de uma aplicação, para que se possa focar melhor na lógica de negócios ao invés da infraestrutura de comunicação em si (IBM, 2020).

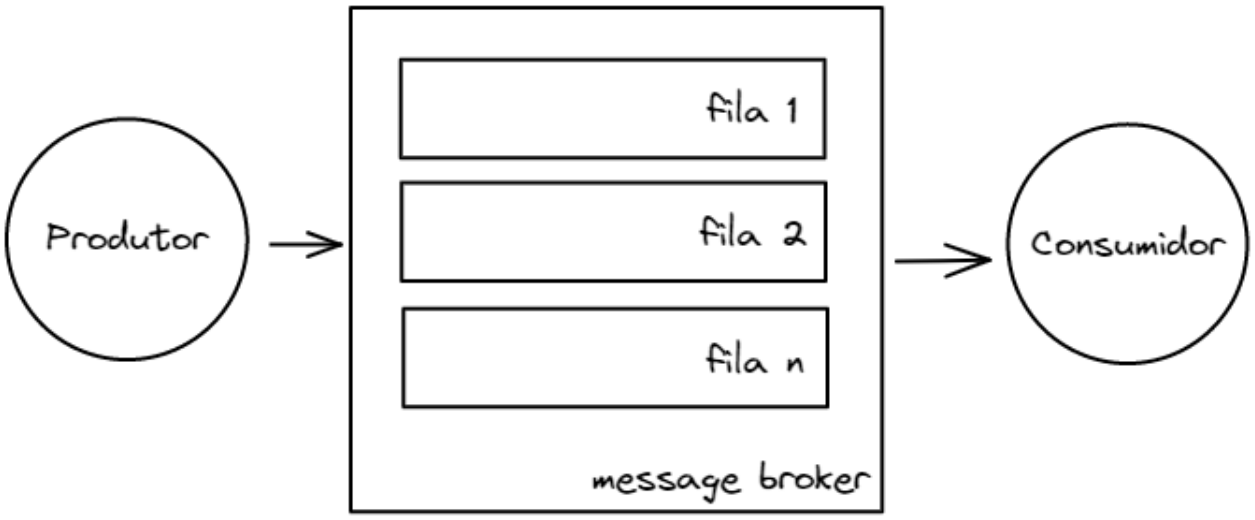

Figura 4.1: Arquitetura geral de um message broker.

O objetivo dos *message brokers* é validar e encaminhar as mensagens para o destinatário apropriado, permitindo um desacoplamento maior entre as partes que formam o todo.

Isso é possível através de uma subestrutura chamada fila de processamento (ou *message queue*), que armazena e ordena as mensagens para que os seus consumidores consigam acessá-las e processá-las, geralmente aguardando uma confirmação de recebimento por parte de quem está processando a mensagem.

Todo *message broker* precisa atender aos seguintes itens em seu ciclo de vida:

- Transmitir mensagem para um ou mais destinatários.
- Agregar mensagens, desacoplando uma mensagem para diversos recipientes, para então retornar para um único remetente.
- Interagir com um repositório externo para servir como um servidor de cache, por exemplo.
- Prover um modelo de distribuição baseado em mensagens usando o padrão '*publish-subscribe*', onde se tem os consumidores, que estão inscritos em uma determinada fila, aguardando a publicação de alguma mensagem pelo produtor.

Este conceito pode ser todo atribuído ao contexto de mensageria, sendo o responsável pela transferência de informações. Os *message brokers* também são chamados de espinha dorsal, ou '*backbone*' da aplicação, sendo essencial para sistemas distribuídos.

4.2 *Backbone* escolhido

Para essa nova estrutura de comunicação da aplicação, foi escolhido o NATS (NATS.IO, 2022) como *message broker*, que servirá de *backbone* de toda a mensageria do sistema Conecta.

NATS é uma infraestrutura que permite a troca de informações entre aplicações e serviços de software, segmentada em forma de *mensagens*, servindo a camada de *middleware*.

O NATS é uma tecnologia conectiva, que funciona como motor de mensageria entre sistemas distribuídos. Uma tecnologia conectiva é responsável por endereçar, descobrir e trocar mensagens impulsionando diferentes metodologias de mensageria, como o modelo '*publish/subscribe*', perguntando e respondendo solicitações para microsserviços.

Fundamentalmente, o papel do NATS é publicar e escutar mensagens de uma fila (endereçadas por um *sujeito*), onde por um lado se tem a aplicação produtora, que vai responder uma solicitação, e por outro as aplicações consumidoras. Ambas as aplicações - produtora e consumidora - dependem de um destinatário, que é o sujeito. De maneira geral, um sujeito é apenas uma *string* de caracteres que compõem um nome, permitindo que duas partes se encontrem. Esse conceito é melhor ilustrado na Figura 4.2:

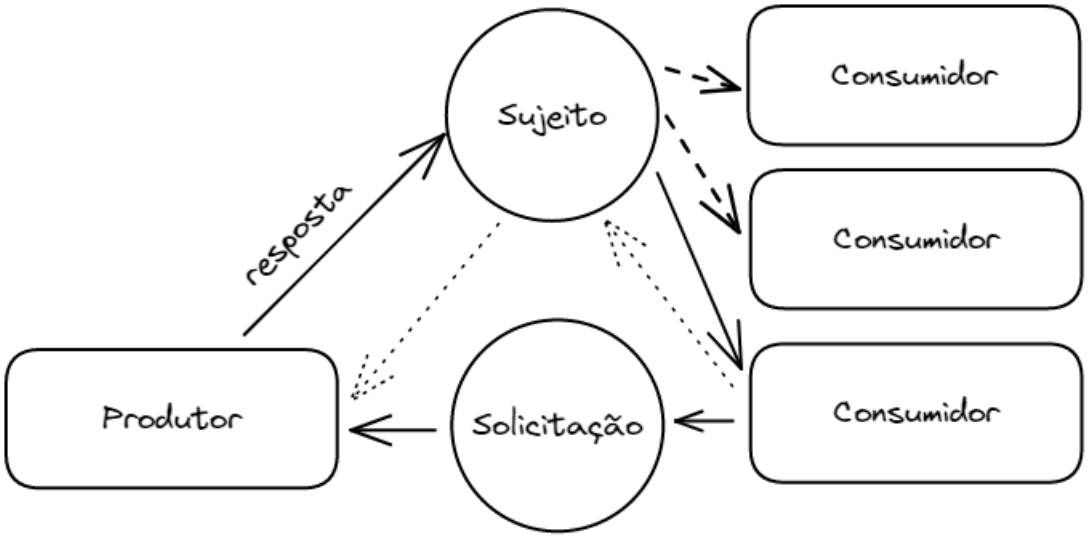

Figura 4.2: Arquitetura de mensageria do NATS.

No caso da Figura 4.2 e fazendo um paralelo com o sistema Conecta, o produtor seria a aplicação de servidor, que estará sendo executada no mesmo ambiente Windows que o ERP, e os consumidores seriam uma API Rest que os *frontends* consomem para execução dos seus serviços. Tais *frontends* referem-se tanto ao que é utilizado atualmente no *Conecta* quanto ao feito para testes, discutido mais adiante.

O NATS gerencia a descoberta baseado em sujeitos, e não portas ou nomes de computador (*hostnames*). Ele tem como padrão comunicação 1:1, que já atenderia a todos os requisitos da nova aplicação, mas também tem suporte para comunicações M:N.

É versátil e pode ser executado em diversos ambientes diferentes, como conteineres Docker, dentro de máquinas virtuais ou nativamente. Pode ser considerado inerentemente seguro, pois não necessita de configurações de *firewall* e provendo mecanismos internos para autenticação

Na solução proposta, o servidor NATS estará hospedado em um servidor de acesso público, e compõe a camada de *middleware* da aplicação toda.

4.3 Novo modelo solicitação-resposta

Se observado em projetos similares, a metodologia solicitação-resposta é comumente utilizada em sistemas distribuídos, onde uma solicitação é enviada para uma aplicação produtora (a aplicação servidor no contexto deste trabalho), por uma aplicação consumidora (a API Rest) e o produtor deve responder dentro de um determinado intervalo limite ou enviar a resposta de maneira assíncrona, utilizando-se do modelo provido pelo NATS.

Dentro da mensageria do NATS, isso é implementado nativamente através de comandos '*publish*' e '*subscribe*', provendo uma forma das duas aplicações se encontrarem através de sujeitos, *strings* que servem de ponto comum entre o produtor e o consumidor. Nessa nova aplicação, o endereçamento é dado pelo sujeito *solicitação.NRCLI*, e então é possibilitada a descoberta das duas aplicações que compõem o novo sistema Conecta, uma delas sendo a produtora e a outra consumidora, tendo uma atribuição 1:1 entre ambas. A atribuição é única devido ao caráter sequencial do número do cliente (NRCLI), que vai ter sempre apenas um servidor respondendo às solicitações.

4.3.1 Microsserviços

A nova aplicação *Conecta* foi estruturada com microsserviços em mente que, em contraste às aplicações monolíticas, servem a um propósito específico e granular dentro do contexto do sistema (NESTJS, 2022), permitindo melhor distribuição lógica na arquitetura geral.

Microsserviços são uma variação do estilo orientado a serviços, que define uma aplicação como coleções de serviços, que estão fracamente acoplados e se comunicam através de protocolos (na camada de *middleware*). Microsserviços têm como objetivo terem o menor

nível de responsabilidade para um determinado fim, funcionando independentemente de outras aplicações ou serviços que estejam implantados.

São duas principais aplicações que funcionam nesta nova arquitetura:

1. Aplicação servidor, responsável por processar uma solicitação junto ao ERP e responder de acordo, atuando como produtor de sujeitos *solicitação.NRCLI* onde *NRCLI* é o número sequencial atribuído a um cliente.

2. API Rest, responsável por solicitar uma resposta ao ERP e respondê-la em formato JSON, sendo o consumidor de mensagens solicitação, dependendo do cliente autenticado.

3. O *frontend* da aplicação que irá consumir a API Rest. Neste trabalho foi feita uma nova versão desse *frontend*, que futuramente dará origem à nova implementação do *Conecta* baseada no estudo desenvolvido aqui.

O *framework* escolhido para a construção destas aplicações foi o *NestJS* (NESTJS, 2022), que implementa diversas tecnologias diferentes de microsserviços, dentre outras funcionalidades mais. Mais detalhes sobre o *NestJS* podem ser encontrados no Apêndice A deste documento.

4.3.2 Aplicação servidor

A aplicação servidor (consumidora) é responsável pelo processamento de uma solicitação junto ao ERP, através do *middleware* (NATS), como descrito no diagrama da Figura 4.3.

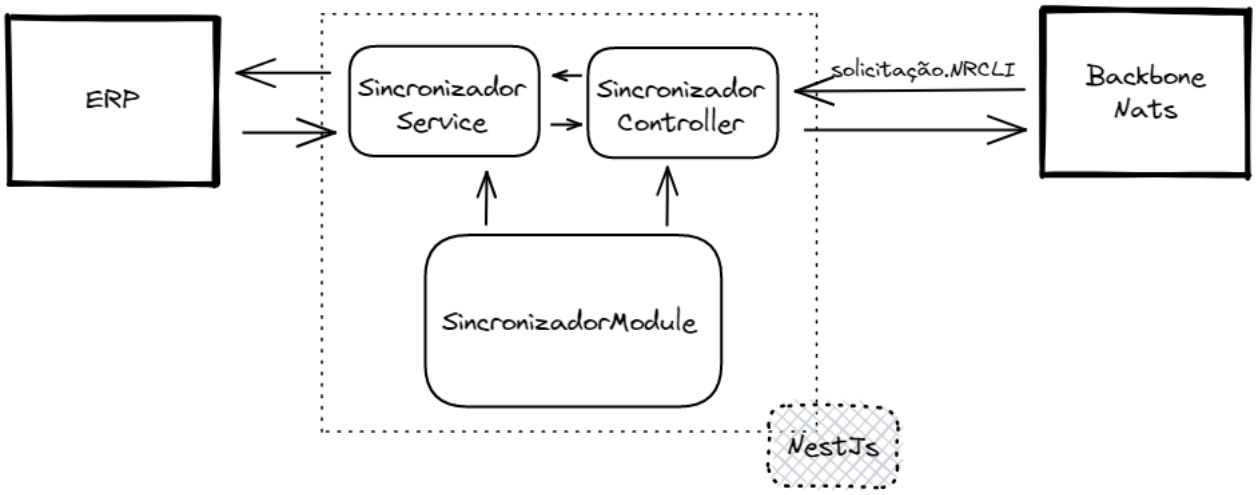

Figura 4.3: Arquitetura da aplicação servidor.

Como evidenciado na Figura 4.3, esta aplicação NestJS possui apenas um módulo: o *Sincronizador*, que provê um controlador e um serviço. O controlador vai aguardar por mensagens com o sujeito determinado. Assim que receber uma solicitação, irá transferir o controle para o serviço da aplicação, que, por sua vez, irá gerar um processo de *runtime* (executável Windows do ERP descrito no capítulo anterior) e aguardar o fechamento (conclusão do processamento) do mesmo. O serviço da aplicação retornará o conteúdo do arquivo de resposta para o controlador, que o devolverá, através do *framework*, para a aplicação solicitante.

Em termos de mensageria, a aplicação consumidora vai sempre escutar pelo sujeito *solicitação.NRCLI*, sendo identificada pelo número sequencial atribuído a cada cliente que possui o ERP instalado.

Ao se inicializar, a aplicação já terá lido e carregado os arquivos de configuração pertinentes para sua execução. Ao receber uma mensagem com um determinado sujeito, irá extrair os parâmetros desta solicitação, gravá-los em disco num diretório pré-determinado e executar o *runtime* ERP para processamento. Após o encerramento deste processo, é lido o arquivo de resposta e seu conteúdo será devolvido à aplicação consumidora que estará aguardando a sua resposta.

4.3.3 API Rest

Para implementar o papel de produtor, foi escrita uma aplicação em NodeJS, utilizando-se também do *framework* NestJS para gerenciar toda a mensageria. Tal aplicação deve ser executada em um servidor público (não necessariamente o mesmo em que estará sendo executado o NATS) e fornecerá uma *API Restful* para consumo, permitindo o uso tanto do método novo de solicitações quanto do antigo. O produtor proposto possui a estrutura ilustrada na Figura 4.4.

Essa aplicação é composta de três módulos:

1. O *Solicitação*, que por sua vez é composto pelos módulos *Nova* e *Antiga*. Este módulo serve um controlador e necessita do seguintes serviço:

    - *SolicitacaoService* provê a camada de serviços da aplicação de solicitação, com métodos diversos que incluem busca de blocos, busca de unidades de um bloco, dentre outras.

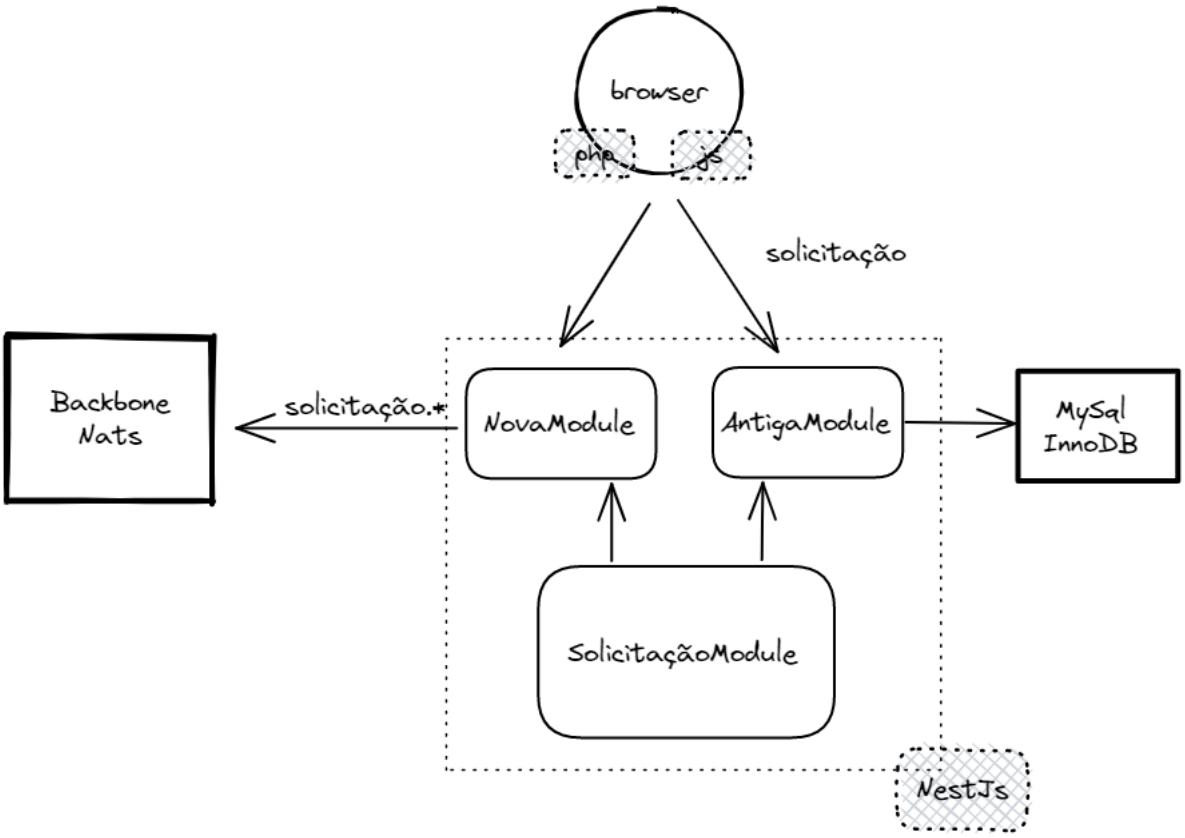

Figura 4.4: Arquitetura da API Rest.

O controlador oferece os seguintes métodos GET:

- /registrar (responsável pela autenticação)
- /vutil/:arquivo/:chave? (lista de algum arquivo de dados, a partir de uma chave)
- /blocos (lista de blocos do condomínio do usuário)
- /unidades/:bloco (lista as unidades de um determinado bloco)

Todos os métodos acima, são protegidos por autenticação, em exceção o de registro, que provê autenticação básica. Todos os outros são protegidos por chaves JSON Web Token (JWT), que identificam quem está fazendo aquela solicitação.

2. O módulo *Nova*, serve como aplicação consumidora da aplicação servidor na versão nova, e irá escutar por novas solicitações através de um controlador que servirá uma *Application Programming Interface* (API) Restful, expondo o método GET de solicitações através do caminho *Hypertext Transfer Protocol* (HTTP) */:tpReq/:params*, onde o primeiro parâmetro é o tipo de solicitação, e o segundo são os parâmetros da solicitação em questão.

3. O módulo *Antiga*, funciona de maneira análoga ao *Nova*, mas ao invés da solicitação ser cumprida pelo *backbone* de mensageria do NATS, é utilizado o SGBD MySQL, assim como era originalmente.

As solicitações são feitas a partir de um browser, seja a aplicação web do *Conecta* ou o *frontend* escrito para os testes deste projeto.

4.4 *Frontend* da aplicação

O *frontend* visto no Capítulo 2 foi mostrado para contextualizar o leitor acerca do sistema *Conecta*. Devido à complexidade desta aplicação e o fato dela já estar sendo usada em ambientes de produção, optou-se pela criação de um novo *frontend* para testes, que servirá ao mesmo fim, da autenticação ao processo de solicitação-resposta.

O *frontend* para testes do sistema de comunicação proposto foi escrito em Javascript, utilizando o *React* para a construção das telas. Maiores detalhes acerca do *React* e as ferramentas utilizadas no *frontend* podem ser encontradas no Apêndice B.

Essa aplicação consiste de três páginas web, que serão melhor descritas nas próximas seções.

4.4.1 Página de registro

A página de registro simula um login no sistema *Conecta*, tendo como entrada o identificador único (ID) do usuário e a sua senha (Figura 4.5).

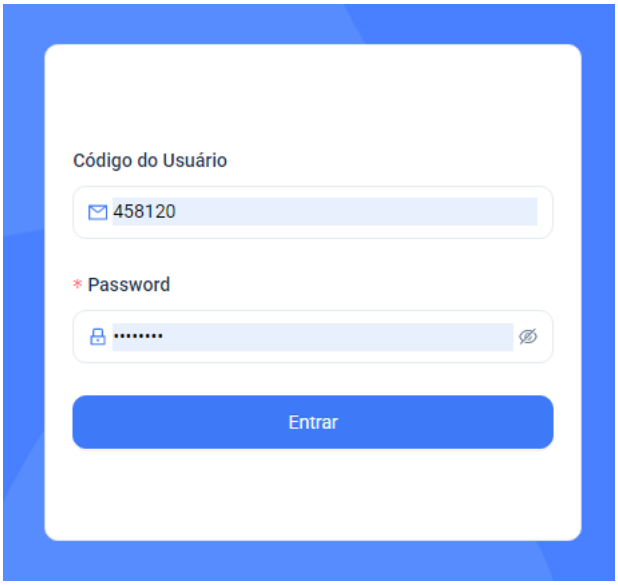

Figura 4.5: Tela de login.

Para os testes descritos aqui, foi utilizado um condômino de um edifício qualquer da *administradora 990*. Ao clicar no botão entrar, o usuário é levado para a página de solicitações.

4.4.2 Página de solicitações

Nesta página se transaciona e demonstra o ciclo de vida de uma solicitação no sistema *Conecta*, tendo filtros para diferentes tipos de solicitação e um campo para exibição de sua resposta em uma seção logo em baixo (Figura 4.6).

As solicitações possíveis deste *frontend* de testes são:

- *Dados do Prédio*, onde o sistema retorna as informações do prédio;
- *Dados de Condôminos*, onde se seleciona um intervalo de unidades para visualizar melhores informações acerca daqueles moradores;
- *Balancete*, onde é gerado um balanço geral financeiro do prédio;
- *Recebimentos*, onde são retornados todos os recebimentos do prédio dentro de um período especificado;
- *Resumo do Balancete* onde se gera um resumo das entradas e saídas financeiras do prédio, dentro de um período especificado;

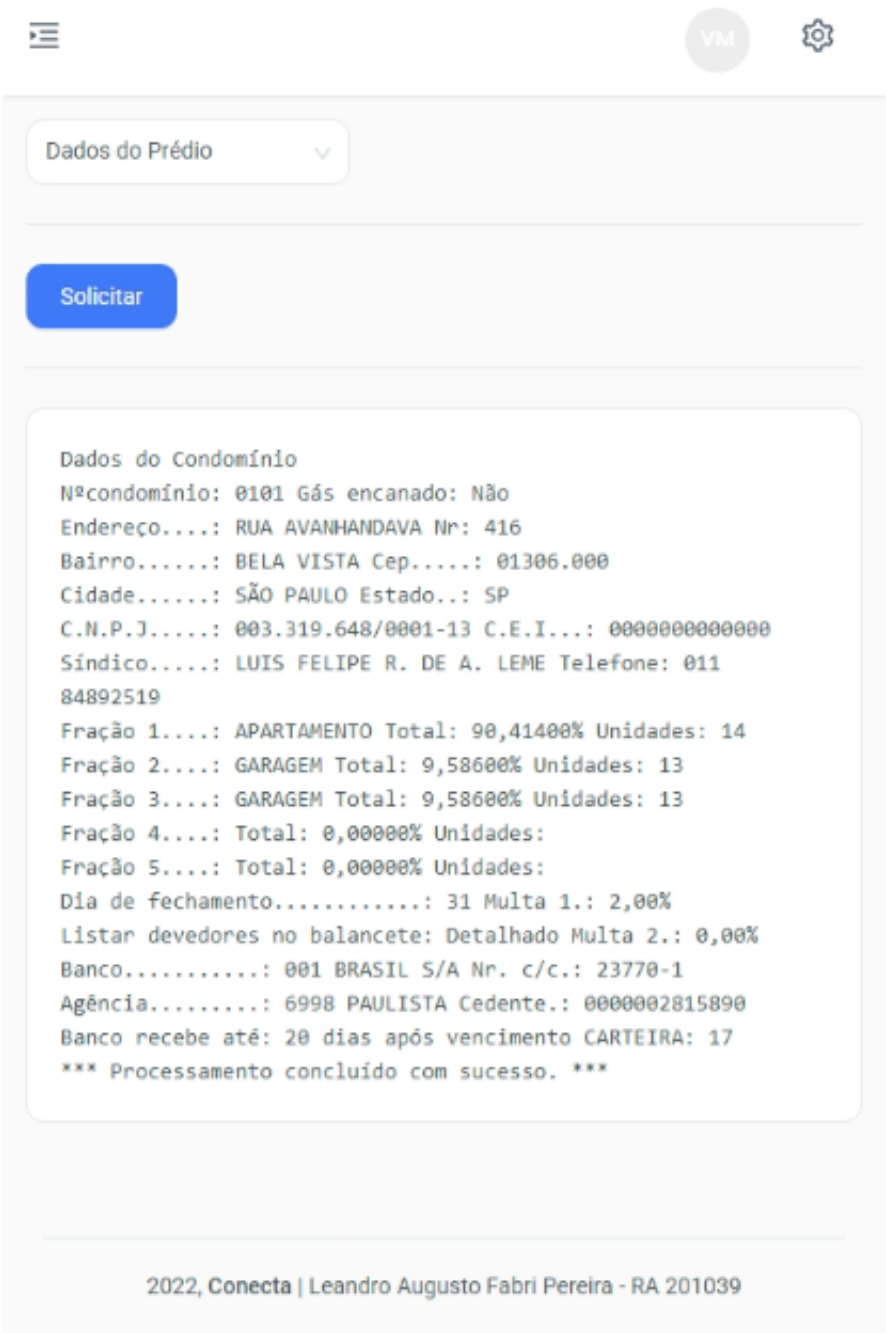

Figura 4.6: Tela de solicitações, com a consulta de dados de um condomínio fictício.

- *Contas a Pagar*, onde são retornadas todas as contas que estão em aberto de um prédio, dentro de um intervalo;
- *Movimento Diário*, que retorna toda a movimentação financeira do prédio dentro de um intervalo;
- *Documentos*, onde é retornado um documento de uma lista, para exibição.

4.4.3 Página de testes

A página de testes (Figura 4.7) permite a execução, em sequência, de um número variado de solicitações, para que seja possível calcular tanto a média em tempo das solicitações quanto a quantidade de dados trafegados através das solicitações, sendo útil para permitir a comparação da versão nova com a antiga.

Após a implementação do sistema, foram feitos experimentos comparativos entre as versões nova e antiga do sistema. Para isso, foram feitas 100 solicitações em sequência, tendo 4,5MB de dados trafegados e um tempo médio de resposta de 122 milissegundos na nova versão da aplicação. Já com o método antigo (também disponibilizado pela API Rest), a quantidade de dados trafegados é a mesma, mas o tempo médio sobe para 300 milissegundos, indicando que o novo método, além de corrigir o problema de armazenamento apontado neste documento, também é mais rápido em termos de processamento da solicitação.

Em outros testes com mil e dez mil solicitações, a aplicação se apresentou estável, respondendo em uma média de 130 milissegundos por solicitação, com um total de cerca de 500MB de dados trafegados. Já o método antigo trafegou a mesma quantidade de dados, mas com uma média de tempo de 270 milissegundos.

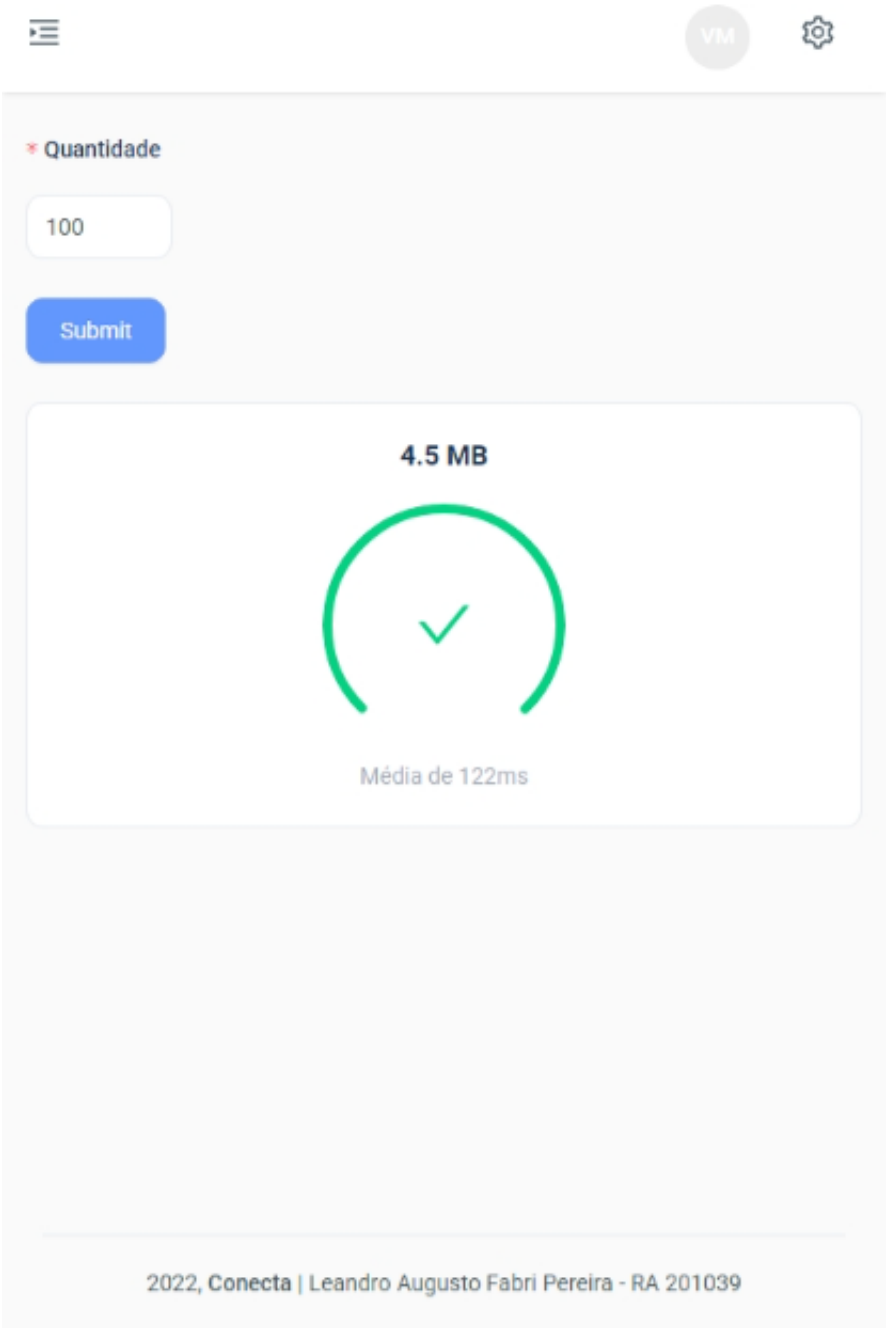

Figura 4.7: Tela de testes.

Capítulo 5

Conclusões

Pela análise feita nos capítulos 2 e 3, onde se introduziu e contextualizou o leitor acerca do sistema *Conecta* e se discutiu qual é o problema enfrentado, pode-se afirmar que realmente existe este problema de sobrecarga de dados na arquitetura de comunicações do sistema atual, provocado pelo uso de um SGBD para trocar informações entre aplicações cliente e servidor. Diante disso, a solução que foi proposta aqui e discutida no Capítulo 4, se baseou na implementação de um *message broker*, o NATS, e o desenvolvimento de novas aplicações cliente e servidor.

Para este trabalho, foram desenvolvidas três aplicações distintas:

- A aplicação servidor, que se comunica com o ERP e processa as transações;
- A *API Rest* que trata de novas solicitações, validando-as e encaminhando-as para a aplicação servidor, através do NATS;
- Um *frontend* para simulações e consumo da *API Rest*.

A API Rest e a aplicação servidor utilizam-se do conceito de mensageria, através do padrão de produtores (aplicação servidor) e consumidores (API Rest). Estes, por sua vez, dependem do NATS para enviar e receber mensagens, que se tornou uma camada de *middleware* da aplicação.

Pode-se concluir então que o problema arquitetural está conceitualmente corrigido, faltando apenas a implementação da API Rest no *frontend* atual que está em produção, este sendo um trabalho futuro.

Referências bibliográficas

BRITO, T. (Ed.). **O que é mensageria?** 2019. Disponível em: <https://medium.com/@devbrito91/mensageria-1330c6032049>. Acesso em: 19 out. 2022.

CHIN, A. **InnoDB Concrete Architecture**. Waterloo, California, fev. 2002.

CHIN; ANDREW ROSZKO, R. B. A. C. lvin. **MySQL Conceptual Architecture**. Waterloo, California, jan. 2002.

CONNOLLY THOMAS M.; BEGG, C. E. **Database Systems – A Practical Approach to Design Implementation and Management (6th ed.).** 6. ed. Desconhecido: Pearson, 2014. P. 1440. ISBN 978-1292061184.

ELLSWORTH, S. (Ed.). **InnoDB ibdata1 never shrinks after data is removed**. 2003. Disponível em: <https://bugs.mysql.com/bug.php?id=1341>. Acesso em: 2 out. 2022.

IBM (Ed.). **O que são message brokers?** 2020. Disponível em: <https://www.ibm.com/br-pt/cloud/learn/message-brokers>. Acesso em: 19 out. 2022.

META (Ed.). **ReactJS**. 2022. Disponível em: <https://reactjs.org/>. Acesso em: 31 out. 2022.

MYSQL.COM (Ed.). **InnoDB Limits**. 2022. Disponível em: <https://dev.mysql.com/doc/refman/8.0/en/innodb-limits.html>. Acesso em: 13 out. 2022.

NATS.IO (Ed.). **Request-Reply**. 2022. Disponível em: <https://docs.nats.io/nats-concepts/core-nats/reqreply>. Acesso em: 12 out. 2022.

NESTJS (Ed.). **Overview - Microservices**. 2022. Disponível em: <https://docs.nestjs.com/microservices/basics>. Acesso em: 12 out. 2022.

NESTJS (Ed.). **NestJs Documentation**. 2022. Disponível em: <https://docs.nestjs.com/>. Acesso em: 13 out. 2022.

Apêndice A

NestJS

O NestJS é um *framework* para construção de aplicações de *backend* em NodeJS, utilizando-se de JavaScript (JS) progressivo (sempre o mais recente). É construído com e suporta nativamente o TypeScript (TS), mas ainda permite que seus utilizadores programem em JS, combinando elementos de Programação Orientada a Objetos (POO), Programação Funcional (PF) e Programação Funcional Reativa (PFR).

Por dentro de seu funcionamento, o Nest faz uso de um servidor *HyperText Transfer Protocol* (HTTP) robusto: o 'Express'. Mas pode também usar o 'Fastify' se assim for configurado. Fundamentalmente, provê uma abstração de ambos os *frameworks* mencionados acima, mas expõe suas *Application Programming Interfaces* (APIs) diretamente para seus desenvolvedores, garantindo a liberdade para usar outros módulos que estão disponíveis para o programador.

Devido à popularidade do NodeJS, e JavaScript ter se tornado uma 'lingua franca', fez-se presente diversos *frameworks* como o Angular, React e o VueJS, mas todos estes são aplicações de *frontend*. Para *backend* faltava algum *framework* que pudesse cuidar de diversos casos de uso e resolver um problema de arquitetura, surge então o NestJS (NESTJS, 2022), provendo uma arquitetura-padrão que permite o desenvolvimento de aplicações testáveis, escaláveis e fracamente acopladas, para fácil manutenção.

Apêndice B

ReactJS

O *frontend* que foi construído para demonstrar o funcionamento do sistema *Conecta* utiliza-se do *framework* ReactJS (META, 2022), que serve para construção de interfaces de usuário. É uma tecnologia de código aberto e gratuíta mantida pela *Meta* (antigo *Facebook*), além de diversos mantenedores individuais.

ReactJS é declarativo, facilitando a criação de interfaces de usuário que são interativas, tornando o código mais fácil e previsível para ser desenvolvido. Também é baseado em componentes, ou seja, porções de código que gerenciam o seu próprio estado e utiliza-os para compor o *frontend* como um todo.

O ReactJS é construído a partir de componentes, que servem para renderizar algum elemento no DOM (*Document Object Model*), podendo ser reutilizados em outras partes da aplicação.