ADEMAR SEIDE JUNIOR

API COLETOR: COLETA AUTOMATIZADA DE INFORMAÇÕES PARA APOIAR A TOMADA DE DECISÃO ORGANIZACIONAL

LAVRAS – MG

2023

ADEMAR SEIDE JUNIOR

API COLETOR: COLETA AUTOMATIZADA DE INFORMAÇÕES PARA APOIAR A TOMADA DE DECISÃO ORGANIZACIONAL

Relatório técnico de graduação apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Sistemas de Informação, para obtenção do título de Bacharel.

Paulo Afonso Parreira Júnior
Orientador

LAVRAS - MG

2023

ADEMAR SEIDE JUNIOR

API COLETOR: COLETA AUTOMATIZADA DE INFORMAÇÕES PARA APOIAR A TOMADA DE DECISÃO ORGANIZACIONAL

API COLETOR: AUTOMATIC COLLECTION OF INFORMATION TO SUPPORT ORGANIZATIONAL DECISION-MAKING

Relatório técnico de graduação apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Sistemas de Informação, para obtenção do título de Bacharel.

APROVADA em 03 de julho de 2023.

Dr. Paulo Afonso Parreira Júnior UFLA

Dra. Renata Teles Moreira UFLA

Dr. Bruno de Abreu Silva UFLA

Prof. Dr. Paulo Afonso Parreira Júnior

Orientador

LAVRAS - MG

2023

Aos meus pais que sempre fizeram o máximo para me proporcionar uma boa educação e boas condições de vida e também àqueles que tive o prazer de chamar de amigo, com os quais pude aprender a ser um profissional e uma pessoa melhor.

AGRADECIMENTOS

Agradeço a Deus, por ter me dado condições de viver com saúde sendo possível correr atrás dos meus sonhos e objetivos.

Aos meus pais Ademar e Juçara, que através de muito esforço sempre me ensinaram sobre honestidade, integridade, esforço e gratidão, esses que nunca deixaram que me faltasse nada que fosse necessário e me ensinaram ao longo desses anos a correr atrás dos meus objetivos.

Aos amigos adquiridos ao longo da vivência na UFLA, com os quais pude aprender e me divertir durante breves mas memoráveis momentos, em especial ao Gustavo, Vinícius, Rafael e Yuri que por diversas vezes me ajudaram durante minha estadia em Lavras.

Aos meus companheiros de viagem da van que compartilharam boas histórias e risadas ao longo dessa jornada.

Ao meu orientador Paulo pelas aulas ministradas, e também durante o processo de realização do trabalho, pela disponibilidade, proatividade e acompanhamento.

À empresa Datacamp e toda a sua equipe que me introduziram no mercado de trabalho além de me suprir com companheirismo e conhecimento.

Aos professores que tive na minha antiga escola Escola Estadual Professor José Monteiro e ao curso pré vestibular Tutores que me proporcionou a oportunidade de conquistar uma bolsa e me preparar melhor para o Enem.

RESUMO

Ao longo da evolução do principal produto de software comercializado pela empresa *Datacamp*, notou-se a necessidade de ter maior controle sobre como os clientes o usam, tanto para aprimorá-lo, quanto para precificá-lo corretamente. Dessa forma, surgiu a demanda por uma ferramenta que conseguisse coletar os dados necessários para fundamentar as discussões sobre o software e seu futuro. A coleta desse tipo de informação é impessoal e muda conforme o tempo, o que destaca duas características importantes no processo de obtenção das informações: nenhuma informação sensível ou privada deve ser comprometida e essa coleta deve ser periódica. Com base nessas informações, surgiu o “API Coletor”, um sistema de informação gerencial que contempla os requisitos acima mencionados e serve como fonte de dados para apoiar o processo de tomada de decisões da empresa *Datacamp*. Como principais resultados alcançados com o uso do “API Coletor”, destacam-se a melhoria dos recursos mais utilizados pelos clientes, a remoção de funcionalidades obsoletas que demandavam manutenção e um conhecimento maior dos clientes da empresa.

Palavras-chave: Sistema de Informação, Interface de Programação de Aplicações, Sistema de Apoio à Decisão.

ABSTRACT

Along with the main software product evolution marketed by the *Datacamp* company, it was noted that there was a need for having more control over how the customers use it, both to improve it and to price it well. In this way, we should have a tool that collects the necessary data to base these debates about the software and its future. The gathering of this kind of information is totally impersonal and changes with time, which highlights two important attributes in the data collection process: no sensitive or private information should be compromised, and this collection must be periodic. Based on that, the "API Coletor" tool emerged, a management information system that fills the above requisites and serves as a data source to support company decisions. As its main results, highlight the improvement of the most used resources by the clients, the removal of useless functionalities that demanded maintenance, and a better knowledge of the company's customers.

Keywords: Information System, Application Programming Interface*, Decision Support Systems*.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

LGPD	Lei Geral de Proteção de Dados
API	Application Programming Interface
SQL	Structured Query Language
ORM	Object Relational Mapper
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
I/O	Input and Output
JSON	Javascript Object Notation
TDS	Tabular Data Stream
REST	Representational State Transfer
CRUD	Create, Read, Update, Delete

SUMÁRIO

1 INTRODUÇÃO

Diante de um país com tantas particularidades, no que diz respeito ao processo de administração de uma empresa que segue a legislação vigente, gerar e manter as informações necessárias para as mais diversas apurações, fiscalizações e auditorias se torna um processo desafiador para 89% das empresas, como aponta o levantamento realizado pela Experian (MERCADOECONSUMO, 2019). Nesse contexto, os sistemas de automação comerciais atuam para facilitar o fluxo de dados da empresa, a fim de organizá-los, classificá-los e disponibilizá-los de forma mais simplificada para consulta. No contexto empresarial, em especial no nicho varejista, os dados coletados vão constituindo a base para se manter a regularidade da empresa com a Secretaria da Fazenda (SEFAZ), como diz Da Silva (2018):

> "A TI [Tecnologia da Informação] facilita a realização das atividades das empresas, no caso varejistas. [Elas] fornecem suporte para organizar e controlar o fluxo das entradas e saídas dos produtos, além de gerar um histórico contendo os dados organizacionais, proporcionando maior agilidade nos processos".

A empresa *Datacamp Automação Comercial*[1] se insere neste mercado, oferecendo, desde 1990, soluções para o pequeno/médio empresário, que precisa gerir seu negócio ao passo em que tem que estar em dia com as obrigações fiscais. Para isso, a solução desenvolvida pela *Datacamp* foi um software de gestão, denominado "Sistema Integrado", que auxilia nos controles básicos de uma empresa, como estoque, financeiro, clientes, fornecedores e também na comunicação com o fisco e na geração dos artefatos necessários para apurações fiscais e contábeis.

À medida que o número de clientes da *Datacamp* aumentava, diversos nichos de mercado foram sendo alcançados, tais como padarias, mercados, postos de combustível, lojas de materiais de construção, dentre outros. Assim, a fim de manter a empresa competitiva no mercado e de valorizar o serviço prestado por ela, percebeu-se a necessidade de se ter um maior conhecimento a respeito do uso do

[1] Por motivo de simplicidade, daqui adiante, a empresa *Datacamp Automação Comercial* será mencionada apenas como *Datacamp*.

“Sistema Integrado” por parte dos clientes. Com isso, seria possível prover um preço personalizado para os clientes, conforme o uso do sistema, prevenir fraudes e usos indevidos e direcionar os esforços das equipes de produto e desenvolvimento para as funcionalidades mais utilizadas, gerando maior satisfação do cliente.

O controle sobre as funcionalidades e extensões do software contratadas pelo cliente era feito apenas no momento de assinatura do contrato, em documentos físicos que, posteriormente, eram armazenados em um arquivo no setor financeiro da empresa. Com isso, a busca pelo tipo de informação desejada pela empresa, conforme explicado anteriormente, demandava muito tempo. Além disso, não era possível saber, apenas por meio desses documentos, quais eram as funcionalidades do sistema mais utilizadas pelos clientes.

Tendo em vista os problemas relatados anteriormente, optou-se pelo desenvolvimento de um software capaz de coletar, automaticamente, informações sobre o uso do sistema, o qual foi chamado de *API Coletor*. Este software coleta dados de uso do sistema periodicamente e os envia para um banco de dados na nuvem, a fim de que possam ser consultados e apresentados aos gestores da empresa. As informações consideradas relevantes para os objetivos de negócio da empresa *Datacamp* foram: (i) o “volume de dados” que os clientes armazenavam no sistema; (ii) a frequência de acesso aos *menus* do sistema; (iii) as funções liberadas para cada cliente; entre outras. É importante salientar que a escolha do tipo de informação a ser coletado, a forma de coleta e de armazenamento levaram em consideração a LGPD (Lei Geral de Proteção de Dados), a fim de que não houvesse violação da mesma.

Este trabalho apresenta detalhes sobre o desenvolvimento e as principais funcionalidades do *API Coletor* e está organizado da seguinte forma: no Capítulo 2, são apresentadas as tecnologias e ferramentas utilizadas para o desenvolvimento da solução; o Capítulo 3, por sua vez, trata da concepção do software em termos mais técnicos; o Capítulo 4 discute sobre os resultados obtidos com a implantação da solução em um ambiente real; por fim, o Capítulo 5 apresenta as considerações finais deste trabalho.

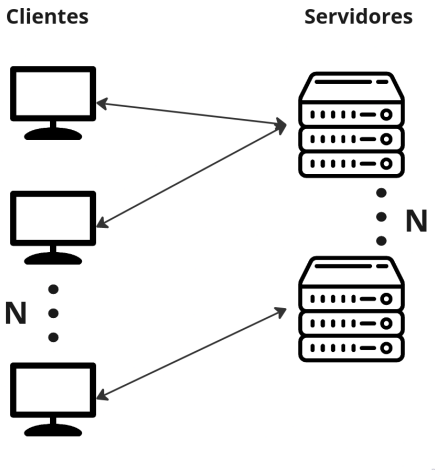
2 REFERENCIAL TEÓRICO

Neste capítulo são apresentadas as tecnologias utilizadas para o desenvolvimento da solução final, assim como um embasamento teórico sobre elas, de forma a contextualizar o leitor sobre suas características e possibilidade de aplicação.

2.1 NodeJS

Aplicações distribuídas exigem definições sobre seus componentes e sobre os relacionamentos inerentes a eles, o que dá origem ao termo **arquitetura**. Dentre os vários tipos de arquitetura existentes, uma bastante conhecida é denominada “cliente-servidor”. Cliente-servidor, como define Bungart (2017), é a relação em que vários dispositivos chamados de “clientes” requisitam os serviços de um outro dispositivo denominado “servidor”, como mostra a Figura 1. Esse tipo de arquitetura permite uma escalabilidade horizontal[2] facilitada, reduzindo os custos de comunicação entre os envolvidos na troca de informações e também o tempo de resposta (De Oliveira, 2003).

Figura 1 - Arquitetura Cliente-Servidor



Fonte: Elaborada pelo Autor

[2] Escalabilidade horizontal é a técnica de replicar a aplicação ao invés de aumentar os recursos de uma única máquina, distribuindo a carga entre as aplicações.

As tecnologias utilizadas no lado servidor, também conhecido como *backend*, devem apresentar características condizentes com a arquitetura em que irão operar. No caso da arquitetura cliente-servidor, dentre as características principais, estão: capacidade de sustentar fluxos de troca de mensagens, gerenciamento e orquestração de requisições e facilidade para replicação.

Tais características são contempladas pela tecnologia “NodeJS”, definida como “um *runtime* assíncrono JavaScript, orientada a eventos, para construir aplicações de rede escaláveis” (NODE, 2023, tradução nossa). NodeJS é uma plataforma que opera de forma assíncrona, isto é, os processos que estão em execução não irão bloquear a entrada e saída de dados (TEIXEIRA, 2012), permitindo um fluxo constante de requisições, mesmo com outras operações em andamento.

NodeJS possui um repositório de código aberto, o *npm*, que conta com mais 1,3 milhões de pacotes disponíveis sobre as mais diversas licenças (GITHUB, 2020). Assim, diversas bibliotecas podem ser usadas para abstrair e acelerar o desenvolvimento de uma aplicação distribuída, dentre elas *ExpressJS*[3] e *PrismaJS*[4], as quais serão retratadas nas próximas seções.

2.2 ExpressJS

ExpressJS, como o próprio projeto se descreve (EXPRESS, 2023), “é um *framework* para desenvolvimento de aplicações web mínimo e flexível, que fornece um conjunto robusto de recursos”. Em suma, com ExpressJS, consegue-se abstrair e simplificar trechos de código NodeJS, sem limitar as funcionalidades nativas da plataforma. Como ilustração, em ambos os trechos de código das Figuras 2 e 3, é instanciado um servidor HTTP que contém uma única rota com retorno de sucesso, a rota “/”. Contudo, no código da Figura 2 foi utilizado apenas NodeJS, enquanto que no da Figura 3 foi utilizado NodeJS juntamente com ExpressJS.

[3] https://expressjs.com/pt-br/
[4] https://www.prisma.io/

Figura 2 - Servidor HTTP implementado com NodeJS puro

```
1  const http = require('node:http');
2
3  const server = http.createServer((req,res) => {
4    if((req.url === '/') && (req.method === 'GET')) {
5     res.writeHead(200);
6     res.end('GET SUCESSO');
7   } else {
8     res.writeHead(404);
9     res.end('NOT FOUND');
10   }
11 });
12
13 server.listen(3000, 'localhost', () => {
14   console.log('Servidor rodando com HTTP');
15 });
```

Fonte: Elaborada pelo autor

Na linha 1 do código da Figura 2, é importada a biblioteca *node:http*, nativa do NodeJS, e instanciada uma constante *server*, que será responsável por armazenar o objeto que representa o servidor. Após criado, o servidor irá interpretar toda requisição feita a ele na porta *3000*, conforme definido na linha 13 do código. Para o método *createServer* (linha 3), é passada uma função que recebe os parâmetros *req* e *res*, abreviaturas para *request* e *response*, respectivamente. Dentro do corpo desta função, está o comportamento desejado para tratar a requisição recebida e gerar uma resposta.

Na Figura 3, ocorre o mesmo processo, porém em vez de instanciar um objeto oriundo de uma classe nativa do NodeJS, é instanciado um objeto “express” (linha 2). Pode-se notar que o código se torna menos verboso, ao retirar a necessidade do uso de operações condicionais, que tendem a se tornar maiores e mais complexas, à medida que o software cresce e oferece mais funções.

Figura 3 - Servidor HTTP implementado com ExpressJS

```
1  const express = require('express');
2  const app = express();
3
4  app.get('/', (req,res) => {
5    res.status(200).send('GET SUCESSO');
6  });
7
8  app.use((req,res) => {
9    res.status(404).send('NOT FOUND');
10 })
11
12 app.listen(3000, () => {
13   console.log('Servidor rodando com ExpressJS');
14 });
```

Fonte: Elaborada pelo autor

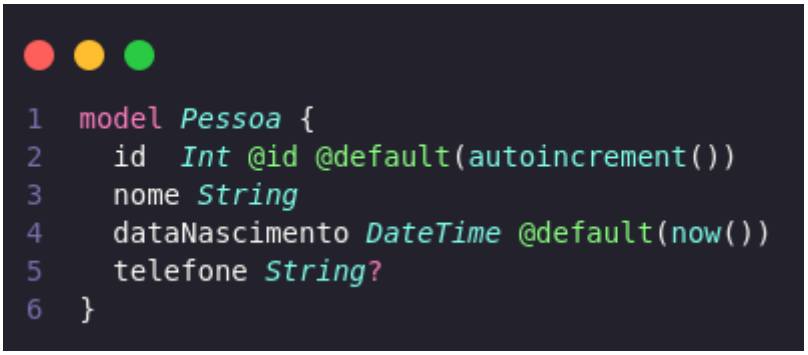
2.3 PrismaJS

PrismaJS é um ORM (*Object Relational Mapping*), isto é, uma biblioteca que abstrai boa parte do código da linguagem SQL necessário para manipulação de dados em uma aplicação, a fim de que o desenvolvedor consiga focar na lógica de negócio do sistema. Em contrapartida, ele também oferece limitações, quanto ao controle sobre o banco de dados e sobre necessidades mais complexas, como criação de *triggers* (gatilhos), por exemplo.

PrismaJS trabalha com o conceito de *migrations*, uma forma de versionar o banco de dados. Essas *migrations* podem ser criadas a partir de esquemas escritos em JavaScript, que são traduzidos para SQL. Por exemplo, a representação de uma tabela de nome *Pessoa*, com os atributos *nome*, *data de nascimento* e *telefone* pode ser visualizada na Figura 4.

A palavra-chave *model* (linha 1) é utilizada para representar uma tabela. Também é possível atribuir características aos atributos, por exemplo, para definir identificadores auto-incrementados, como é o caso do atributo *id* (linha 2).

Figura 4 - Tabela *Pessoa* representada utilizando PrismaJS

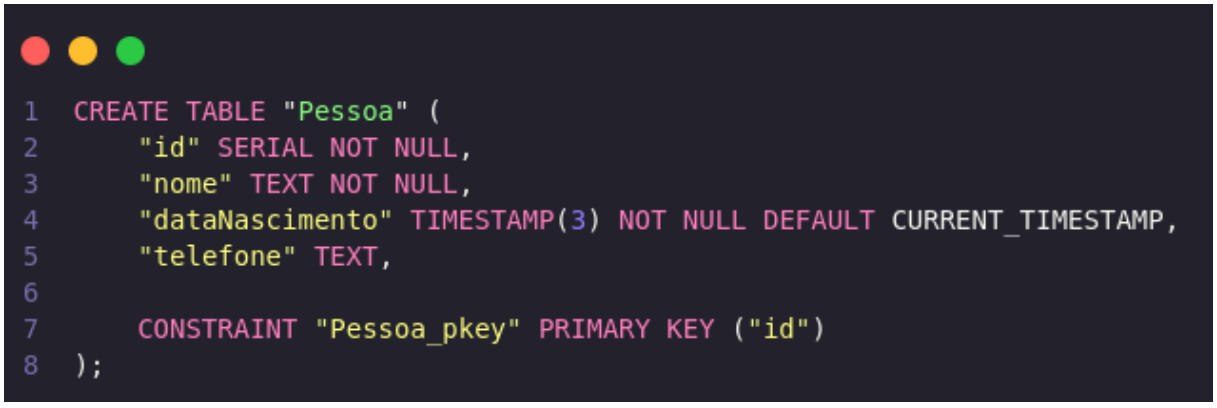


```
model Pessoa {
  id  Int @id @default(autoincrement())
  nome String
  dataNascimento DateTime @default(now())
  telefone String?
}
```

Fonte: Elaborada pelo autor

Na Figura 5, é possível observar a transformação do modelo descrito na Figura 4 para um comando SQL, com suas tipagens próprias, *constraints* (restrições), dentre outras coisas. Essa transformação é feita através de um *provider*, uma implementação responsável por lidar com os arquivos ".prisma", nesse caso provinda do próprio PrismaJS (mais detalhes sobre *provider* são apresentados na Seção 3.6).

Figura 5 - Tabela Pessoa comando SQL



```
CREATE TABLE "Pessoa" (
    "id" SERIAL NOT NULL,
    "nome" TEXT NOT NULL,
    "dataNascimento" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
    "telefone" TEXT,

    CONSTRAINT "Pessoa_pkey" PRIMARY KEY ("id")
);
```

Fonte: Elaborada pelo autor

2.4 Rust

Rust é uma linguagem de programação compilada, que foi concebida dentro de uma das empresas mais conhecidas no ramo da tecnologia, a *Mozilla*. Rust surgiu oficialmente em 2006 e possui como proposta permitir a construção de

aplicações de alta performance, com controles mais próximos de hardware, tais como C e C++, mas mantendo a simplicidade de sintaxe e de recursos de linguagens de alto nível, tais como Go e Python (GOLDEN, 2022).

Na Figura 6, tem-se um código-fonte escrito em C++. Esse código declara duas variáveis, uma de tipo *string* e outra do tipo *inteiro*. Posteriormente, ele faz a exibição desses valores no terminal (linhas 7 e 8). A título de comparação, na Figura 7, há o mesmo algoritmo, porém escrito na linguagem Rust. É notável uma certa semelhança entre eles, mas também exemplifica a diferença de sintaxe, uma vez que Rust tenta se aproximar das linguagens de mais alto nível, como Javascript e Python.

Figura 6 - Código básico C++

```cpp
#include <iostream>

int main() {
  std::string variavel_string = "UFLA";
  int variavel_numerica = 2023;

  std::cout << "Variável em String: " << variavel_string << std::endl;
  std::cout << "Variável em String: " << variavel_numerica << std::endl;

  return 0;
}
```

Fonte: Elaborada pelo autor

Figura 7 - Código básico Rust

```rust
fn main() {
  let variavel_string: &str = "UFLA";
  let variavel_numerica: i32 = 2023;

  println!("Variável string: {}", variavel_string);
  println!("Variável númerica: {}", variavel_numerica);
}
```

Fonte: Elaborada pelo autor

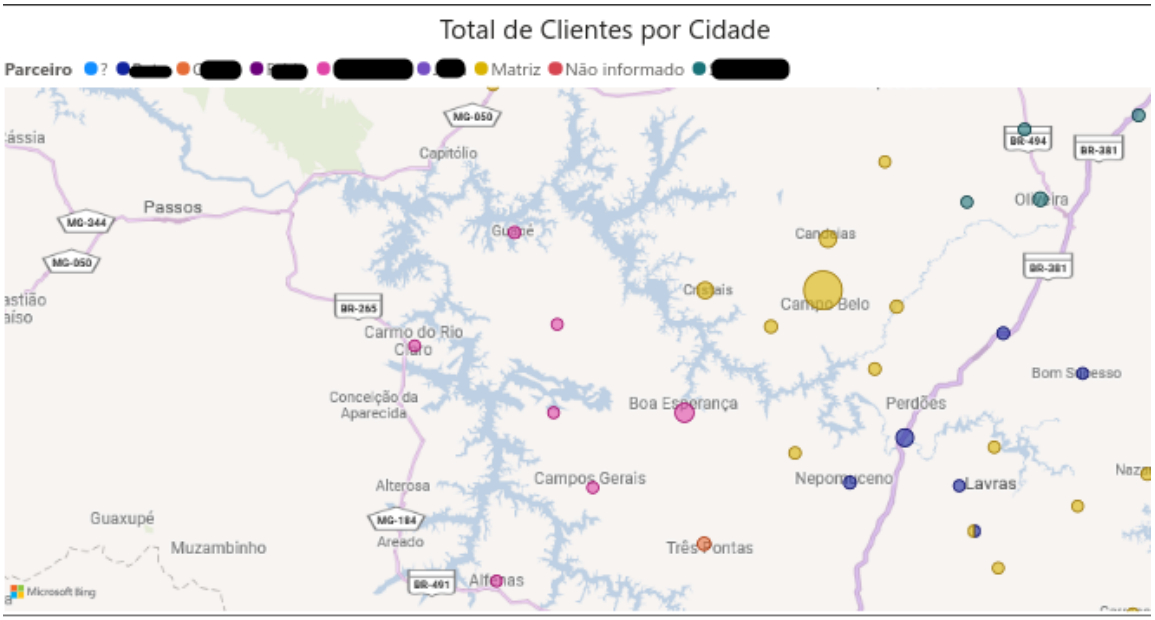
Um dos principais atrativos da linguagem Rust é o seu compilador, responsável por otimizações e principalmente por conta de como ele gerencia a memória. O compilador de Rust ajusta o binário gerado para que não seja necessária uma verificação de tempos em tempos das regiões de memória não mais utilizadas, como faz o *garbage collector* da plataforma Java, por exemplo (KLABNIK; NICHOLS, 2023), obtendo assim melhor performance em tempo de execução.

Outro atrativo de Rust é o seu gerenciador de pacotes, chamado *Cargo*. Ele é responsável por controlar as dependências utilizadas em um programa e também por tornar possível exportar pacotes e distribuí-los (RUST, 2023). Esse recurso, já embutido como ferramenta oficial da linguagem, apresenta a mesma vantagem do *npm* para o NodeJS: uma comunidade *open-source* oferecendo diversas bibliotecas e implementações, contribuindo para o ecossistema da linguagem e a fortalecendo.

2.5 Microsoft Power BI

Power BI é uma coleção de serviços de software e conexões, capaz de transformar dados em informações coerentes (MICROSOFT, 2023). Em outras palavras, o *Power BI* pode ser definido como uma ferramenta que gera relatórios, apresentações gráficas e outras formas de apresentação de dados, a fim de apoiar a tomada de decisão. Por exemplo, com *Power BI*, é possível saber a concentração de clientes da empresa *Datacamp* por área, como mostra a Figura 8. Para isso, esta ferramenta permite utilizar múltiplas fontes de dados externas, tais como banco de dados relacionais (tais como PostgreSQL, MariaDB) ou não-relacionados (tais como MongoDB).

Figura 8 - Exemplo de visualização Power BI por localização



Fonte: Adaptado pelo painel de monitoramento de propriedade pela empresa *Datacamp Automação Comercial*

Power BI também apresenta portabilidade para vários dispositivos, permitindo criar, compartilhar e consumir *insights* empresariais da maneira mais eficiente (MICROSOFT, 2023). Por fim, é importante destacar que o *Power BI* segue a estrutura de negócios *Freemium*, na qual o software pode operar de maneira gratuita, porém com certas restrições.

3 API COLETOR

Em busca de construir uma base de dados para catalogar, classificar e entender as necessidades dos clientes da empresa *Datacamp*, faz-se necessário saber quais dados são relevantes e elaborar uma forma de coletá-los que minimize o custo da operação. Isso poderia ser feito, por exemplo, requisitando as informações diretamente aos clientes, por meio de questionários, ou por meio de um sistema de coleta de dados automatizado. Dentre essas opções, a última se destaca pela garantia de obtenção de respostas e pela agilidade durante a operação.

Assim, foi desenvolvido o *API Coletor,* uma solução automatizada para a coleta de dados, seguindo preceitos de segurança, privacidade e desempenho. Ele tem o propósito de coletar informações que sejam relevantes para controle da empresa prestadora de serviço, isto é, a *Datacamp*, de forma a permitir o acompanhamento do processo de atualização dos clientes, auxiliar em decisões sobre o desenvolvimento de novas funcionalidades, bem como a manutenção ou remoção das já existentes. Este capítulo tem o objetivo de apresentar os requisitos e as funcionalidades mais relevantes da solução proposta.

3.1 Requisitos funcionais

A fim de definir a funcionalidade do API Coletor, a equipe de gestão e produto, a qual requisitou o software, se reuniu e listou quais análises eram desejadas e quais dados seriam necessários para elas. Após a definição das análise e dos dados necessários, os requisitos funcionais do sistema foram definidos e elencados por prioridade (baixa, média e alta), conforme pode ser visto na Tabela 1.

Tabela 1 - Requisitos funcionais - API Coletor

#	Descrição	Prioridade
1	O sistema (*API Coletor*) deve coletar os dados básicos relativos à empresa cliente e de domínio público, tais como razão social, nome fantasia, CNPJ, município sede e telefone de contato.	Alta

2	O sistema deve coletar as informações de forma periódica, em intervalos de 30 dias, sendo que as informações enviadas deverão ser relativas sempre ao mês completo anterior à data de coleta.	Alta
3	O sistema deve coletar as informações relativas ao número de acessos/cliques por *menu* do sistema.	Alta
4	O sistema deve coletar os dados relativos ao software instalado, tais como versão e última data de atualização.	Média
5	O sistema deve coletar dados sobre o número de registros cadastrados para “Produtos”, “Serviços” e “Documentos Auxiliares de Vendas”.	Baixa
6	O sistema deve coletar dados sobre o número de documentos fiscais emitidos no período relativo à coleta.	Baixa

Fonte: Adaptada do documento original de requisitos, de propriedade da empresa *Datacamp Automação Comercial*

3.2 Requisitos não funcionais

Acompanhando os requisitos funcionais do sistema, na Tabela 2, são apresentados os requisitos não funcionais. Os requisitos não funcionais foram fundamentados, em parte, pelas questões técnicas e funcionais para utilização, contudo houve também a preocupação em se estabelecer parâmetros de segurança e comportamento.

Tabela 2 - Requisitos não funcionais - API Coletor

#	Descrição	Prioridade
1	O sistema deve ser transparente ao usuário do sistema, de forma a não “aparecer” ou influenciar no seu uso.	Alta
2	O sistema deve coletar os dados de forma a respeitar a Lei Geral de Proteção de Dados (LGPD), com relação ao sigilo de informações, não coletando qualquer informação de uso que não seja impessoal e que não viole a privacidade, sigilo ou posse dos dados do cliente.	Alta
3	O sistema deve implementar mecanismos de segurança para que as informações não possam ser adquiridas e ou interpretadas por terceiros durante a coleta e envio dos dados.	Alta
4	O sistema deve armazenar as informações em um banco	Alta

	de dados na nuvem com disponibilidade dos dados através da internet mediante credenciais autorizadas.	
5	O sistema deve apresentar as informações em uma interface que possibilite personalização de visualização entre gráficos e tabelas.	Média
6	A visualização das informações coletadas deve ser possível por meio da internet através de credenciais autorizadas e apresentar disponibilidade de uso durante horário comercial (8:00 às 18:00), salvo os momentos de atualização, manutenção e ou interrupção do serviço por motivos adversos e além do controle.	Média

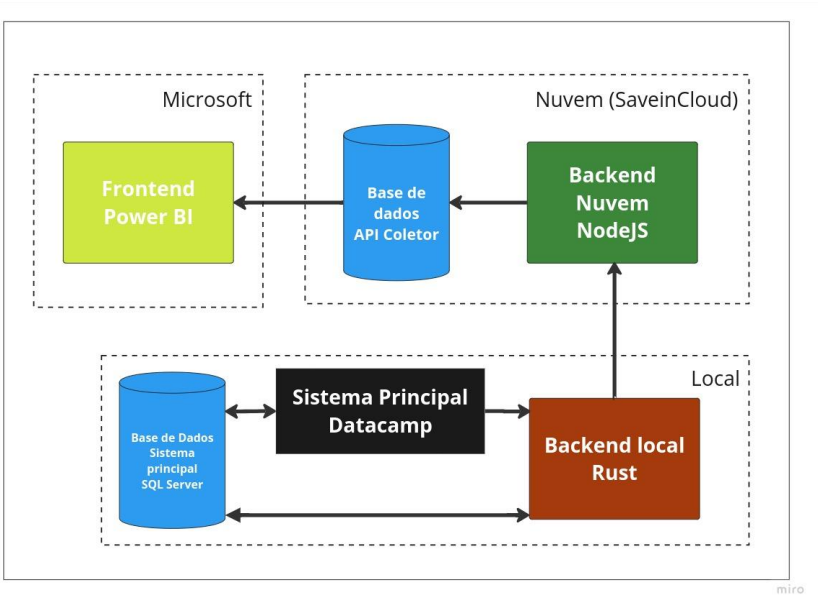
Fonte: Adaptada do documento original de requisitos, de propriedade da empresa *Datacamp Automação Comercial*

Todos os requisitos de prioridade “Alta” (funcionais e não funcionais) foram cumpridos e entregues já na primeira versão do sistema.

3.3 Arquitetura do API Coletor

Nesta seção, é apresentada a arquitetura do ambiente em que a solução *API Coletor* funciona e foi idealizada, a qual foi dividida em três partes principais (Figura 9): o *Local*, a *Nuvem* e um ambiente externo, que foi denominado *Microsoft*.

Figura 9 - Arquitetura do API Coletor



Fonte: Elaborada pelo autor

O ambiente “Local” é a fonte dos dados a serem coletados, que compreende a máquina do cliente e seu banco de dados. Nesse ambiente fica instalado um programa escrito em Rust, que se comunica com os componentes do ambiente “Nuvem”. No ambiente denominado “Nuvem”, que encontra-se hospedado no provedor *SaveInCloud*[5], está localizado o banco de dados do API Coletor, bem como a API implementada em NodeJS, para receber os dados advindos do ambiente “Local”. Por fim, o *frontend* do API Coletor é fornecido pelo ambiente “Microsoft”, por meio do serviço de *Power BI*, que se comunica diretamente com a base de dados do API Coletor, para apresentar as informações, sem necessidade de uma hospedagem própria.

Para melhor ilustrar o funcionamento do sistema e sua a arquitetura, considere o seguinte cenário: o dia de coleta chega[6] e o sistema em Rust, que fica armazenado no servidor local do cliente da *Datacamp*, é acionado para iniciar a coleta dos dados, realizar a encriptação dos mesmos e enviá-los para a API do coletor, hospedada da nuvem (SaveinCloud). Na nuvem, a aplicação em NodeJS tem a responsabilidade de decodificar os dados e então salvá-los no banco de dados. Por fim, o *Power BI* fica responsável por se conectar ao banco de dados da API e coletar os dados, de tempos em tempos, para então apresentá-los em sua interface web.

3.4 Detalhes de implementação do ambiente “Local”

Para o cliente da *Datacamp*, a aplicação foi idealizada como um executável “invisível”, a ser chamado pelo sistema principal de operação da empresa. Rust foi a linguagem utilizada para implementação desse executável. Por ser uma linguagem de mais baixo nível, ela oferece uma biblioteca padrão mais enxuta, assim, aplicações de alto nível, que envolvam protocolos de camada de aplicação, como o HTTP, requerem os chamados *crates*, as bibliotecas da linguagem Rust. Por isso, houve a necessidade de se utilizar algumas bibliotecas de terceiros, uma vez que a

[5] https://saveincloud.com/pt/

[6] Para clientes que nunca enviaram os dados, a coleta será feita na primeira vez que o sistema principal for utilizado. Para aqueles que já se comunicaram alguma vez com o API Coletor, é definido um dia de 1 a 28 para que a coleta seja realizada.

própria linguagem não oferecia suporte.

Primeiramente, foi necessário obter os dados oriundos do banco de dados do cliente, nesse caso, o *Microsoft SQL Server*. Para isso foi utilizada a biblioteca *tiberius*[7], um cliente que se comunica com o banco, por meio do protocolo *Tabular Data Stream* (TDS). Um exemplo de código para essa finalidade se encontra na Figura 10 logo abaixo:

Figura 10 - Código RUST que utiliza a biblioteca Tiberius

```rust
use tiberius::{Client, Config, Query, AuthMethod};
use std::{error::Error};
use tokio::net::TcpStream;
use tokio_util::compat::TokioAsyncWriteCompatExt;

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {

    let mut config = Config::new();

    config.host("127.0.0.1");
    config.port(1433);
    config.database("NOME_BANCO");

    config.authentication(
        AuthMethod::sql_server("usuario_autorizado", "senha_segura")
    );

    config.trust_cert();

    let tcp = TcpStream::connect(config.get_addr()).await?;

    let mut client = Client::connect(config, tcp.compat_write()).await?;

    Ok(())
}
```

Fonte: Elaborada pelo autor

Com exceção da linha 2, que refere-se à importação da biblioteca *standard* do Rust, as demais importações são de biblioteca de terceiros, *tiberius* e *tokio*[8]. *Tokio* é um runtime assíncrono; ele provê um ambiente orientado a eventos para execução de código Rust, a flag *#[tokio::main],* na linha 6, indica ao compilador que deve usar

[7] https://crates.io/crates/tiberius
[8] https://crates.io/crates/tokio

esse ambiente para execução do código. Nas linhas 9 a 23 estão as configurações para conexão com o banco, tais como *IP*, *porta*, *nome do banco de dados* e *usuário,* dentre outras. Uma observação é que a linha 19 não é usada em produção, pois ela ignora a verificação de certificado de segurança; em ambientes oficiais, é usada uma outra função, que valida o certificado. Por fim, na linha 25, é o momento de retorno da função *main*, indicando que o código foi executado com êxito.

A fim de garantir o sigilo dos dados e cumprir com as obrigações da LGPD, os dados movimentados entre o ambiente “Local” e a “Nuvem” têm o seu anonimato e integridade preservados. Para isso, foi utilizada uma biblioteca chamada *magicCrypt* [9], disponível em ambas as linguagens utilizadas, Rust e JavaScript (NodeJS).

Na Figura 11, há um exemplo de codificação de dados, no qual se cria uma instância da classe *magicCrypt* (linha 3), informando a chave secreta como primeiro parâmetro e o número de *bits* para o algoritmo de criptografia como segundo parâmetro. Na linha 5, é passado o valor para que seja então devolvido o mesmo já criptografado.

Figura 11 - Código RUST com biblioteca *magicCrypt*

```rust
use magic_crypt::{new_magic_crypt, MagicCryptTrait};

let mc = new_magic_crypt!("magickey", 256);

let base64 = mc.encrypt_str_to_base64("http://magiclen.org");

assert_eq!("DS/2U8royDnJDiNY2ps3f6ZoTbpZo8ZtUGYLGEjwLDQ=", base64);
```

Fonte: Retirado da documentação da biblioteca rust-magiccrypt

Em ambiente de produção, essa biblioteca foi utilizada para criptografar o nome e valor das propriedades do JSON enviado na requisição para a API na nuvem. Da mesma forma, o programa em NodeJS detém o conhecimento da chave secreta e pode descriptografar os dados recebidos.

Para a transmissão dos dados, por meio do protocolo HTTP, foi usada a

[9] https://crates.io/crates/magic-crypt

biblioteca *reqwest*[10]. Essa biblioteca oferece os métodos básicos deste protocolo, tais como *GET*, *POST*, *PUT* e *DELETE.* Além disso, ela oferece dois modos, um assíncrono e outro síncrono, para realizar requisições. A demonstração de uma requisição, utilizando a biblioteca *reqwest,* se encontra na Figura 12. Nesse exemplo, é utilizado um *HashMap* (linhas 11 e 12), estrutura que funciona como um conjunto de chave-valor, para conseguir representar dados no formato JSON. Na linha 9, é criado o cliente HTTP. Posteriormente, na linha 14 esse cliente é utilizado para enviar uma requisição do tipo POST, com os dados inseridos no *HashMap*, que serão convertido para JSON por meio da função *json()*. Por fim, a Figura 13 demonstra o fluxo de execução que o *backend* local executa sobre os dados.

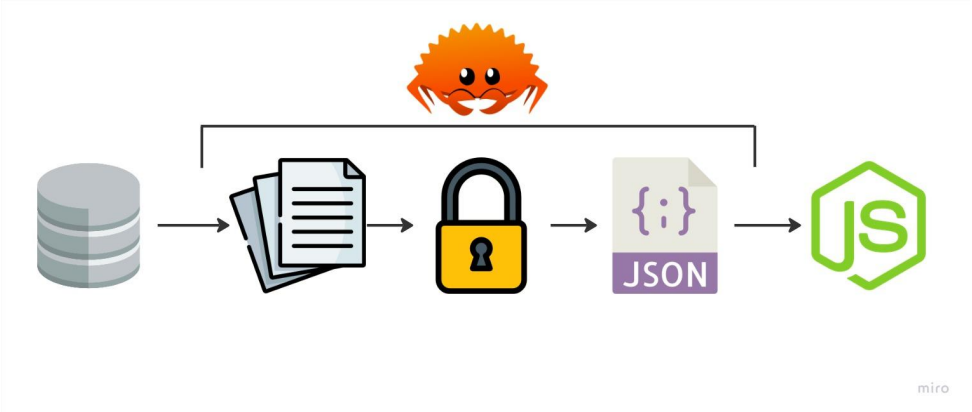
Figura 12 - Código RUST utilizando a biblioteca reqwest

```
1  use std::{collections::HashMap};
2  use reqwest;
3
4  let mut map: HashMap<&str, String> = HashMap::new();
5
6  let id : i32  = 999;
7  let name : &str  = "Ademar";
8
9  let client = reqwest::Client::new();
10
11 map.insert("id", id.to_string());
12 map.insert("name", name.to_string());
13
14 let res = client.post("http://localhost:3000")
15 .json(&map)
16 .header("CONTENT_TYPE", "application/json")
17 .send()
18 .await?;
19
20 res.status(); //StatusCode::OK (200)
21 res.text(); // Mensagem de sucesso
```

Fonte: Elaborada pelo autor

[10] https://crates.io/crates/reqwest

Figura 13 - Ilustração do fluxo de execução do ambiente "Local"



Fonte: Elaborada pelo autor

3.5 Detalhes de implementação do ambiente “Nuvem”

Para suprir a demanda de disponibilidade, escalabilidade e centralização dos dados, a parte central do API Coletor foi concebida na estrutura de nuvem. A fim de permitir um desenvolvimento mais ágil e não tão centrado em economia de recursos, foi utilizado o ambiente NodeJS e a hospedagem do provedor *SaveInCloud*.

A aplicação foi organizada baseando-se nos conceitos do padrão arquitetural MVC (*Model-View-Controller*), para delimitar a responsabilidade de cada parte do código. Também foi realizada a implementação de características do padrão de projeto *Chain Of Responsability*, mais fortemente notado no uso do *framework* ExpressJS, como será apresentado mais adiante.

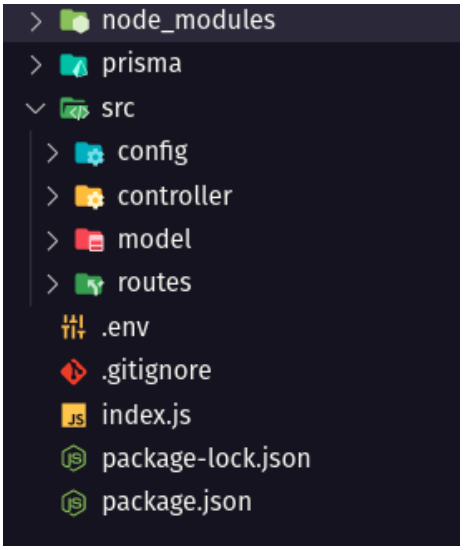
A Figura 14 apresenta a estrutura de pastas do projeto. Ela se inicia com a pasta *node_modules*, na qual são armazenadas as bibliotecas instaladas pelo NodeJS. Essa pasta agrupa todas as dependências e códigos das bibliotecas instaladas. A seguir, a pasta *prisma* armazena a estrutura do banco de dados e as *migrations*, que serão explicadas posteriormente. Na pasta *src*, abreviação para *source*, se encontra o código fonte principal da API do coletor. Dentro dela, há as configurações de bibliotecas e os componentes das camadas *models* e *controllers*. Os componentes do tipo *controller* são responsáveis por validações prévias dos dados e os do tipo *model* são relacionados ao contato direto com o banco de dados. Por último, na pasta *routes*, estão os componentes responsáveis pelo

roteamento/navegação da aplicação em si. Fora da pasta *src*, estão os arquivos contendo as variáveis de ambiente, configurações para controle de versão, dentre outros.

O redirecionamento de requisições para as funcionalidades da API em nuvem é feito pelo *framework* ExpressJS, explicado na Seção 2.2. Esse *framework* executa a partir dos mapeamentos dos métodos HTTP (tais como GET, POST, etc). Também é possível definir tratamentos gerais para todas as requisições, utilizando o método *use*, geralmente usado para autenticação, *logs* e coleta de dados para análises estatísticas de consumo.

A Figura 15 apresenta, das linhas 11 a 25, as rotas para o recurso “clientes”. Há também tratamentos gerais (linhas 7 a 9), válido para todos os métodos que tentam acessar o recurso "clientes". As funções passadas para os métodos HTTP são denominadas *arrow functions*, que dispõe de 3 parâmetros para que o desenvolvedor possa acessar os dados e funcionalidades de uma chamada à rota. O parâmetro *req*, abreviatura para *request*, é um objeto responsável por receber os dados advindos do cliente HTTP que faz a requisição. Por meio dele, é possível obter dados do cabeçalho da requisição; alguns deles são o método HTTP utilizado, *tokens* de acesso e o *body* ou corpo da requisição, contendo os dados enviados (ver Figura 16). Já o parâmetro *res*, abreviatura de *response*, é o objeto responsável pela resposta a ser enviada ao cliente.

Figura 14 - Estrutura de pastas da API hospedada na nuvem



Fonte: Elaborada pelo autor

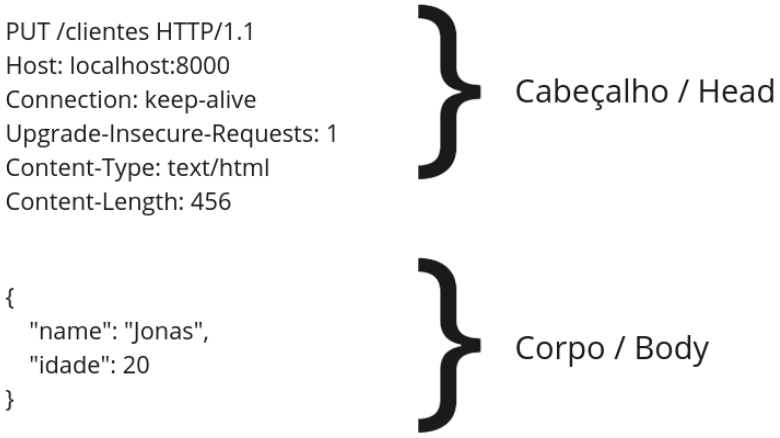
Figura 15 - Mapeamento de rotas com o framework ExpressJS

```
1  import express from 'express';
2
3  const app = express();
4
5  app.use(express.json());
6
7  app.use('/clientes', (req, res, next) => {
8    /** Tratamento geral para todas as rotas /clientes */
9  });
10
11 app.get('/clientes', (req, res) => {
12   /** Tratamento requisição GET */
13 });
14
15 app.post('/clientes', (req, res) => {
16   /** Tratamento requisição GET */
17 });
18
19 app.put('/clientes', (req, res) => {
20   /** Tratamento requisição GET */
21 });
22
23 app.delete('/clientes', (req, res) => {
24   /** Tratamento requisição GET */
25 });
```

Fonte: Elaborada pelo autor

Um exemplo de uso desses objetos se encontra na Figura 17. Na linha 2, é feita uma verificação, por meio da função *validaToken*, sobre o valor da propriedade *authorization*. Caso o retorno de *validaToken* seja verdadeiro, na linha 3, é enviado o código 200, indicando o sucesso, e uma mensagem em texto plano. Caso contrário, na linha 5, é enviado o código 500, evidenciando um erro, juntamente com uma mensagem em texto plano.

Figura 16 - Ilustração dos dados advindos de uma requisição HTTP



Fonte: Adaptado das documentações sobre HTTP da MDN Web Docs (2023)

Figura 17 - Exemplo de uso dos parâmetros de uma requisição

```
1  app.get('/clientes', (req, res) => {
2    if (validaToken(req.headers.authorization)) {
3      res.sendStatus(200).send("Autorizado o acesso.");
4    } else {
5      res.sendStatus(500).send("Erro na autorização.");
6    }
7  });
```

Fonte: Elaborada pelo autor

Por fim, o último parâmetro de uma requisição, chamado *next*, corresponde à próxima função que vai interceptar a chamada do recurso. Ele é mais usado no método *use*, uma vez que esse método realiza um tratamento padrão e chama a rota responsável por lidar com o conteúdo da requisição. Esse mecanismo de passagem de responsabilidade é conhecido como *middleware* e é uma variante de implementação do padrão de projetos *Chain Of Responsability* (GAMMA et al., 2000). Os *middlewares* podem ser encadeados, permitindo assim tratar os dados da requisição, até o envio da resposta ao cliente. No caso do API Coletor, o principal uso dos *middlewares* é para a descriptografia dos dados.

Na Figura 18, é apresentado um exemplo de funcionamento de um *middleware*. Na linha 2, é invocada uma função que altera o corpo da requisição para uma forma descriptografada. A linha 3, por sua vez, chama-se a próxima

função na cadeia (*next*). Na linha 6, o *middleware* declarado nas linhas 1 a 4 é utilizado em um *endpoint* do tipo POST, para o recurso “clientes”. Assim, no momento que uma requisição do tipo POST chegar ao recurso “clientes”, primeiramente, a requisição irá passar pela função *descriptografarConteudo* e só depois continuará a execução das linhas 7 a 10, já com os dados do “req.body” sem criptografia.

Figura 18 - Uso de middleware no ExpressJS

```
1  function descriptografarConteudo(req, res, next) {
2    req.body = decrypt(req.body);
3    next();
4  }
5
6  app.post('/clientes', descriptografarConteudo, (req, res) => {
7    /**
8     * Tratamento requisição POST já com os
9     * dados descriptografados
10    * */
11 });
```

Fonte: Elaborada pelo autor

Para a API do coletor, foram disponibilizados 5 recursos principais, cujas definições e sobre quais dados eles operam são apresentados na Tabela 3.

Tabela 3 - Recursos da API do coletor

#	Recurso	Função
1	Clientes	Lida com o cadastro e atualização dos dados dos clientes.
2	Versões	Armazena o número da versão e a data de atualização do sistema instalado no cliente.
3	Informações	Armazena as informações relativas ao uso dos recursos do sistema pelo cliente, como o número de documentos fiscais emitidos.
4	Cliques	Armazena os *menus* clicados e a quantidade de cliques realizados no sistema, pelo cliente.
5	Usuários	Armazena as informações de liberação de acesso dos usuários cadastrados para o cliente, como quais *menus* o

		usuário tem acesso e quais ações o mesmo pode realizar dentro do sistema.

Fonte: Elaborada pelo Autor

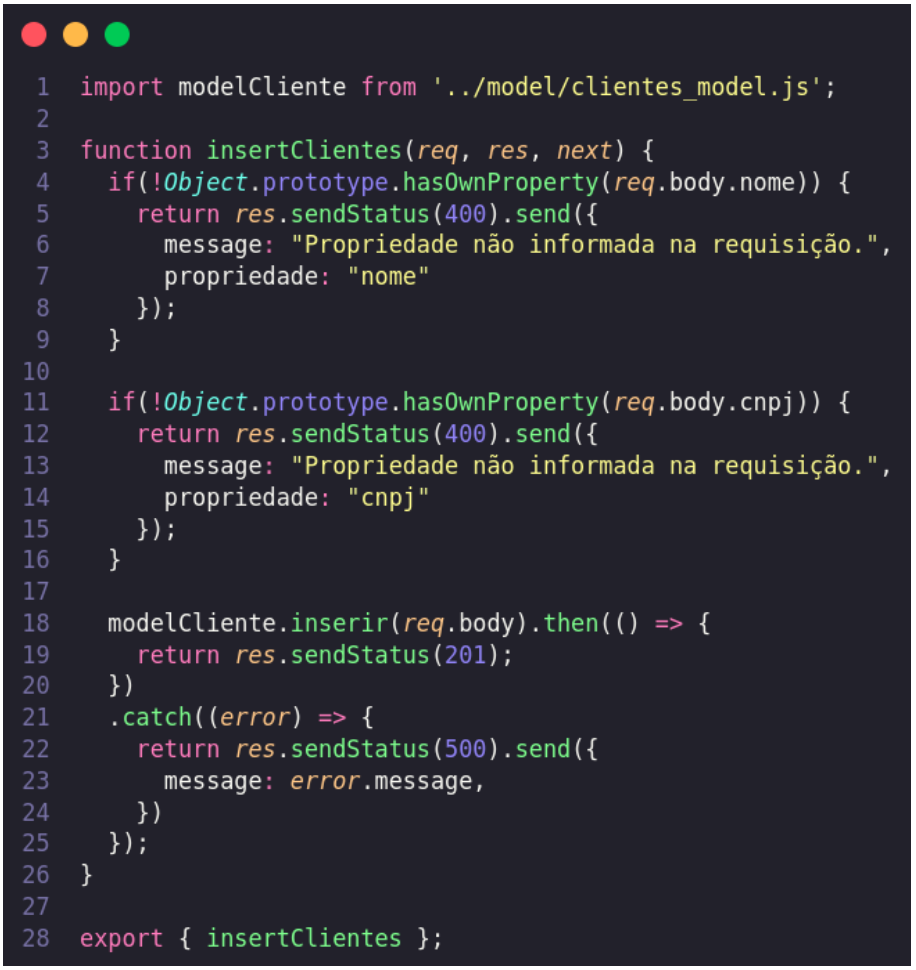
Para cada rota, é disponibilizado ao cliente HTTP apenas o método POST para cadastro dos dados, com exceção do recurso de “Clientes”, o qual oferece a possibilidade de alteração de dados e, por isso, também possui uma rota configurada para o método PUT.

Na Figura 19, é apresentado um fluxo de execução[11] da aplicação para o cadastro de dados de um cliente. Analisando-se a execução do código, nas linhas 4 e 11, é verificado se o corpo da requisição contém os atributos necessários para o cadastro de um cliente[12]. Em caso de negativa, é enviada uma resposta com o código 400 (esse é um erro padrão do protocolo HTTP, conhecido como “*Bad Request*”) e uma mensagem explicativa. Caso a requisição e o corpo estejam compatíveis, a linha 18 é executada, a qual invoca a função *inserir* da classe *modelClient*. Após a execução, é enviado ao cliente uma resposta positiva com o código 201 (“*Created”*), em caso de sucesso, ou uma resposta negativa com código 500 (“*Internal Server Error*”), caso contrário. A prática de retornar erros internos para o cliente não faz parte das boas práticas de desenvolvimento de API públicas e pode apresentar falhas de segurança, porém, como se trata de um projeto completamente interno essa decisão foi considerada aceitável.

[11] É importante salientar que o fluxo foi simplificado para que pudesse ser apresentado didaticamente neste trabalho, por isso, o código da Figura 19 omite algumas validações dos dados e também não trata de erros como duplicidade dos dados ou inconsistências.

[12] Para manter sigilo quanto ao código da solução da empresa *Datacamp*, os atributos e implementações reais da entidade “Cliente”, bem como sua lógicas de negócio foram omitidos.

Figura 19 - Código para cadastramento de clientes

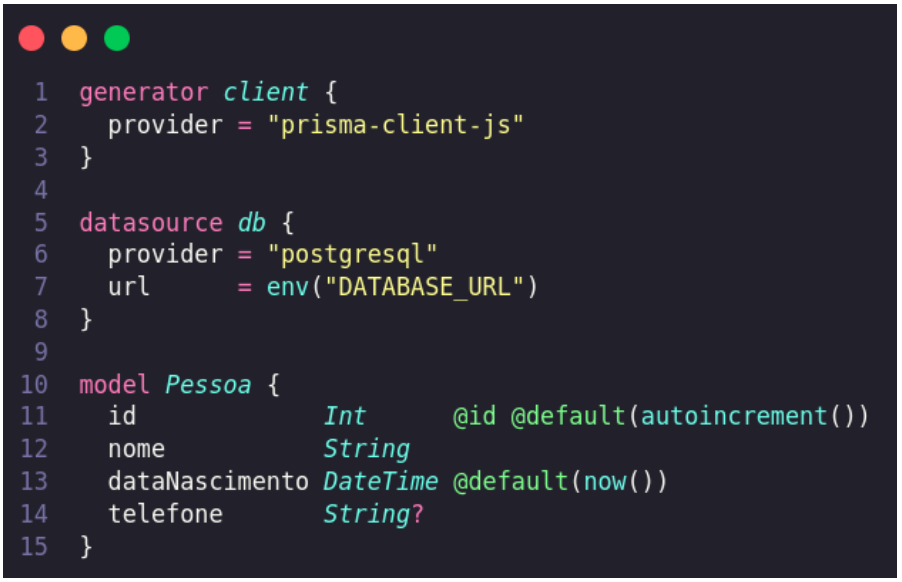


```
import modelCliente from '../model/clientes_model.js';

function insertClientes(req, res, next) {
  if(!Object.prototype.hasOwnProperty(req.body.nome)) {
    return res.sendStatus(400).send({
      message: "Propriedade não informada na requisição.",
      propriedade: "nome"
    });
  }

  if(!Object.prototype.hasOwnProperty(req.body.cnpj)) {
    return res.sendStatus(400).send({
      message: "Propriedade não informada na requisição.",
      propriedade: "cnpj"
    });
  }

  modelCliente.inserir(req.body).then(() => {
    return res.sendStatus(201);
  })
  .catch((error) => {
    return res.sendStatus(500).send({
      message: error.message,
    })
  });
}

export { insertClientes };
```

Fonte: Elaborada pelo autor

Conforme explicado na Seção 2.3, PrismaJS é o ORM que permite a manipulação dos dados da aplicação, utilizando programação orientada a objetos com JavaScript. Assim, a impedância entre os paradigmas relacional e orientado a objetos é reduzida e tende-se a aumentar a produtividade dos desenvolvedores. PrismaJS consegue estruturar tabelas do banco de dados por meio da definição de esquemas em uma notação mais próxima à linguagem JavaScript. Para isso, é necessário preencher um arquivo chamado *schema.prisma*, gerado após a instalação do PrismaJS (Figura 20).

Figura 20 - Arquivo de exemplo completo schema.prisma



```
1  generator client {
2    provider = "prisma-client-js"
3  }
4
5  datasource db {
6    provider = "postgresql"
7    url      = env("DATABASE_URL")
8  }
9
10 model Pessoa {
11   id             Int      @id @default(autoincrement())
12   nome           String
13   dataNascimento DateTime @default(now())
14   telefone       String?
15 }
```

Fonte: Elaborada pelo autor

O arquivo da Figura 20 pode ser dividido em 3 partes principais. A primeira parte é chamada de *generator* (linhas 1 à 3). Nela, é possível definir, por meio da propriedade *provider*, um programa a ser executado para ler um arquivo e gerar alguma saída específica. Por padrão, é chamado o *script* “prisma-client-js”. *Generators* costumam ser usados para gerar documentação, tradução para outras linguagens de programação, criação de esquemas, entre outros usos. Para o API Coletor, o *generator* padrão foi utilizado para gerar os esquemas do banco de dados e gerenciar as *migrations*. A segunda parte é o *datasource*, linhas 5 a 8, responsável por configurar a conexão com o banco de dados. O *datasource* possui um *provider*, que é o nome do *driver* utilizado para comunicação com o banco de dados, e a *url* de conexão, que possui dados de acesso ao banco, tais como usuário, senha, dentre outras informações. Por fim, tem-se os *models*, linhas 10 a 15. Eles representam as tabelas que vão ser criadas no banco de dados. Nas linhas 11 a 14, estão a definição das colunas da tabela, especificando tipos, *constraints*, entre outras informações. Nas linhas 12 e 13, são definidas outras colunas, da mesma forma que a anterior. Na linha 14 há uma particularidade: após o tipo da coluna há um ponto de interrogação (?), que indica que o campo pode ser nulo.

Após a definição do esquema, é realizada a geração das *migrations*. *Migration* é uma estratégia de gerenciamento de versão de banco de dados, a fim de

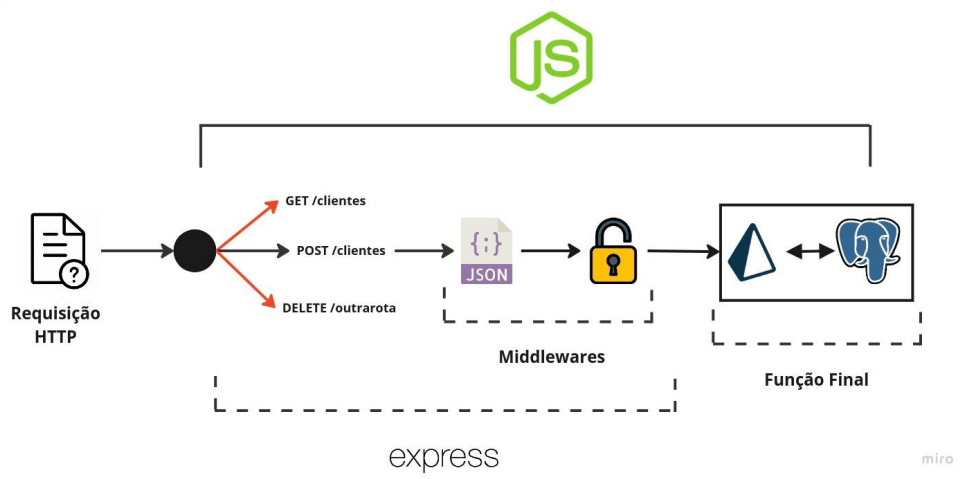
definir pontos no tempo para documentação e permitir “voltar atrás” com alterações, caso seja necessário. Através do comando “prisma generate”, gera-se uma pasta chamada *migrations*, com os comandos SQL puros, conforme pode ser visto na Figura 21. Dessa maneira, o fluxo geral do ambiente "Nuvem" pode ser ilustrado, de forma resumida, pela Figura 22.

Figura 21 - Arquivo SQL de saída do PrismaJS

```sql
-- CreateTable
CREATE TABLE "Pessoa" (
    "id" SERIAL NOT NULL,
    "nome" TEXT NOT NULL,
    "dataNascimento" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
    "telefone" TEXT,

    CONSTRAINT "Pessoa_pkey" PRIMARY KEY ("id")
);
```

Fonte: Elaborada pelo autor

Figura 22 - Ilustração do fluxo de execução do ambiente "Nuvem"



Fonte: Elaborada pelo autor

3.6 Detalhes de implementação do ambiente “Microsoft”

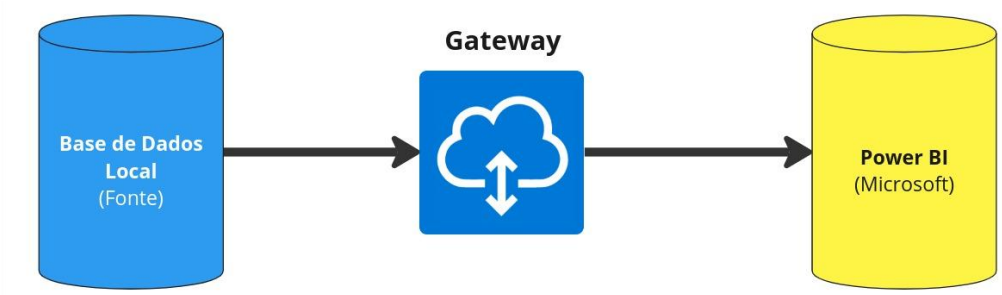
Para a solução de visualização dos dados, foi utilizado o *Microsoft Power BI*, que é a alternativa tida pelos gestores da empresa *Datacamp* como provisória e experimental. O Power BI, ou apenas BI, é uma ferramenta de *Business Intelligence*,

com foco em transformar os dados em informações que permitam melhor visualização de relatórios e geração de ideias.

Para se obter uma fonte de dados, são disponibilizadas duas maneiras: *Direct Query* e *Import*. A *Direct Query*, como o próprio nome sugere, faz conexão direta com um banco de dados e permite o tratamento de grandes volumes de dados, contudo tem suas transformações limitadas, o que afeta o número de possibilidades de relatório. Por essa razão, foi escolhido o modelo de *Import*, baseado em se fazer uma cópia de parte dos dados. Além das personalizações de visualização e transformação dos dados, o fator de custo foi incluído, uma vez que o BI tem serviços de atualização dos dados armazenados periodicamente que variam conforme os planos contratados. Como as informações são para análises gerais, não necessita-se de precisão em tempo real, portanto, as atualizações diárias do plano gratuito suprem a demanda de dados.

Por se tratar de um banco de dados fora dos domínios da Microsoft, mesmo sendo feita uma cópia momentânea dos dados, é necessário definir um ponto de entrada externo para o banco de dados, o chamado *Gateway.* O *Gateway* funciona como intermediário entre o banco de dados do sistema e os servidores da Microsoft. Dessa forma, o fluxo de informações é unidirecional, coletando os dados requisitados da origem e enviando-os para serem armazenados no serviço de Power BI, como mostra a Figura 23.

Figura 23 - Diagrama Gateway Power BI

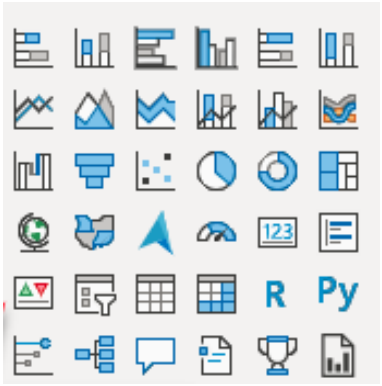


Fonte: Elaborada pelo autor

Para montar as visualizações, após configurar a conexão com o banco de dados, são selecionadas as tabelas relacionais e então são permitidas as

transformações dos dados, pivotagem[13] de tabelas, tratamento de campos para exibição de valores tratados e formatação de valores. Logo a seguir, é definido o tipo de visualização, conforme apresentado na Figura 24.

Figura 24 - Opções de visualização Power BI

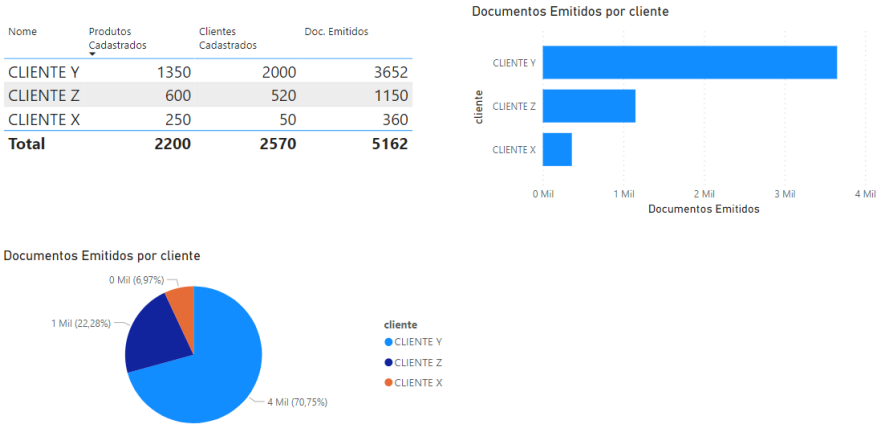


Fonte: Retirado do programa Power BI Desktop

Por meio das opções disponíveis, foi possível montar múltiplas visualizações, como por exemplo as visões em forma de mapa, conforme mostra a Figura 8. Ademais, foram incluídas visões de gráficos de pizza e barras horizontais (Figura 25), permitindo comparação de tamanho e de volume de movimentações entre os clientes. Por fim, também foram desenvolvidas visões de forma tabular, algumas representando puramente a tabela no banco de dados para conferência detalhada de valores, e também tabelas com acumuladores, para visualização numérica das dimensões dos clientes (Figura 25).

[13] Inverter linhas e colunas de uma tabela para ter um visão diferente dos dados

Figura 25 - Visualizações utilizadas no Power BI



Nome	Produtos Cadastrados	Clientes Cadastrados	Doc. Emitidos
CLIENTE Y	1350	2000	3652
CLIENTE Z	600	520	1150
CLIENTE X	250	50	360
Total	**2200**	**2570**	**5162**

Fonte: Elaborada pelo autor

4 ANÁLISE DE USO E RESULTADOS

Com os requisitos básicos para o funcionamento do API Coletor finalizados e após mais de 6 meses de coleta e análise, foi possível iniciar a avaliação do sistema.

4.1 Volume de dados coletados

O funcionamento do API Coletor teve início no mês de Outubro de 2022. Até o mês de Junho de 2023 (9 meses), haviam sido coletados dados de 688 clientes de 55 municípios distintos. Isso representa aproximadamente 63% dos clientes ativos da *Datacamp*. Em termos de volume de dados coletados, havia por volta de 42 mil registros no banco de dados do API Coletor, sendo 30% relativos ao uso das funções principais do sistema, 60% relativos ao uso geral por meio da coleta de cliques em *menus* e 10% correspondentes a outros dados coletados.

Progressivamente, o volume de dados fica muito denso, consequentemente as dimensões de escalabilidade ficam comprometidas em aplicações que necessitem do uso de todos os dados. Contudo, as análises são realizadas de forma periódica e sobre dados com escopo de data definido. Dessa forma, é possível continuar entregando informações de forma performática. Essa abordagem permite também a classificação de dados como obsoletos ao longo do tempo e, assim, é possível armazenar esses dados em forma de *backup* e manter o banco de dados de produção somente com as informações mais relevantes.

4.2 Tomadas de decisão

Após a transformação dos dados em informações, por meio dos relatórios gerados e das discussões sobre os mesmos, foram tomadas decisões sobre a manutenção do sistema principal da *Datacamp*. Das decisões tomadas pela empresa, foram destacadas 3 como principais:

Melhoria dos recursos mais utilizados: por meio da coleta de dados relativos aos cliques nos *menus* do sistema principal, foi possível a visualização

sobre as funcionalidades mais acessadas e a frequência de acessos. Dessa maneira, as funções mais utilizadas foram priorizadas a receberem melhorias de funcionamento e modernização da interface, com o objetivo de tornar o sistema mais fácil e agradável ao usuário final. Com base nos dados até junho de 2023, o cadastro de produto ficou em primeiro lugar em termos de utilização. Isso se deve, em parte, ao crescimento do número de clientes da *Datacamp* e, consequentemente, à necessidade de cadastramento das informações básicas para venda. Em segundo e terceiro lugares, ficaram o lançamento de informações fiscais e a entrada e saída de notas fiscais, respectivamente. Em conjunto com o cadastro de produtos, essas 3 funções receberam prioridade na construção de novas soluções e modernização de interface, buscando demonstrar ao usuário final a preocupação da empresa com a constante evolução do produto.

Remoção de funcionalidades não utilizadas: a partir da análise dos dados, também foi justificável a remoção de funções que não eram mais acessadas no sistema. Uma vez que mais de 50% dos clientes foram alcançados com o API Coletor, o julgamento de descontinuar e remover certas funcionalidades foi pautado nos dados de uso da maioria dos clientes. Com isso, foi possível minimizar o esforço de manutenção da equipe em partes que não estavam mais sendo usadas e não agregavam valor ao cliente.

Abordagem de clientes para acesso antecipado e pontos focais de pesquisa: por meio da análise de uso das funcionalidades por cliente, foi possível separar os clientes com base nessas informações, como também classificá-los para conhecer quem serão os mais afetados em caso de mudanças. Dessa forma, os clientes agora podem ser consultados previamente sobre as mudanças que virão a ser implementadas e, assim, garantir melhor aceitação das mesmas, melhorar as soluções e introduzir o cliente como agente direto sobre as mudanças no software.

4.3 Opinião dos gestores

Funcionários da empresa *Datacamp* foram consultados, por meio de entrevistas, acerca dos benefícios trazidos pelo API Coletor. Os membros da empresa entrevistados foram um gestor da empresa e o gerente da área de produto.

Ambos trabalham na área de produto e são os que mais usam a solução desde sua implementação. As questões realizadas na entrevista, juntamente com as respostas dos entrevistados encontram-se nas Tabela 4, 5 e 6. A entrevista se baseou nos impactos do API coletor para o trabalho dos envolvidos (Tabela 4), qual o benefício notado por meio da utilização do coletor (Tabela 5) e qual a oportunidade de melhoria é enxergada para o seu futuro (Tabela 6).

Tabela 4 - Respostas sobre o impacto do API Coletor no trabalho dos envolvidos

Pergunta: Qual o impacto trazido pelo API Coletor em seu trabalho ?	
Membro	**Resposta**
Gestor	O API Coletor possibilitou uma mudança relevante na forma como gerenciamos o produto e a carteira de clientes. Provê a sustentação necessária para a tomada de decisão, que antes era sobre o que achávamos que os usuários utilizavam, para ser sobre o que estes de fato utilizam.
Gerente de Produto	O API Coletor de forma geral transforma a abstração do cotidiano do cliente em dados analíticos, a grande vantagem de seu uso é a segurança que dados podem trazer para melhor análise do produto e seu uso. Também o armazenamento do perfil de uso do produto, por outro lado somente o coletor não transforma a usabilidade do produto, ou seja, seu uso se faz mais eficaz quando analisado junto a outros dados como números de suporte quanto a problemas em tela ou dúvidas de uso ou nível de satisfação do cliente junto ao produto.

Fonte: Elaborada pelo autor

Tabela 5 - Respostas sobre os benefícios do uso do API Coletor

Pergunta: Qual o principal benefício da API do Coletor notada por você?	
Membro	**Resposta**
Gestor	Tomada de decisão para o produto e suporte com base em métricas de uso do sistema.
Gerente de Produto	O maior benefício citado é a capacidade da visão macro do uso do produto e a possibilidade de visão micro de cada processo inserido no produto, porém esta visão micro de processo com N visões de empresas de ramos distintos.

Fonte: Elaborada pelo autor

Tabela 6 - Respostas sobre futuras melhorias no API Coletor

Pergunta: Qual o próximo passo de melhoria para o API Coletor?	
Membro	**Resposta**
Gestor	Incrementar as informações capturadas em termos de ações realizadas dentro do sistema, exemplo, tipos de relatórios gerados e filtros utilizados.
Gerente de Produto	A possibilidade de medir o tempo de uso em funcionalidade, vejamos, sabemos que o tempo de aprendizado das pessoas é variável. N motivos fazem uma pessoa aprender certa funcionalidade no software, seja por conhecimento de área, ou por não conhecimento de tecnologia, medir este tempo seria útil em tomadas de decisão quanto a treinamento, interfaces mais amigas a usuários ou até mesmo disposição de fluxo de informação dentro do produto, a fim de que o uso do produto se torne linear, auto explicativo e funcional.

Fonte: Elaborada pelo autor

Dessa forma, por meio das respostas obtidas, é possível concluir que a solução idealizada e implementada conseguiu satisfazer as partes interessadas. Além disso, o impacto causado no produto deixa claro que o sistema impactou nas decisões da empresa e também abriu novos pensamentos para melhorias, como o refinamento e diversificação dos dados a serem coletados.

5 CONSIDERAÇÕES FINAIS

O objetivo do API Coletor, em termos simples, era coletar dados sobre o uso do sistema principal da *Datacamp*, catalogá-los e tratá-los de forma que o usuário final da ferramenta fosse capaz de entender como o cliente da *Datacamp* usa o sistema. Mediante os requisitos, o sistema deveria então ter várias fontes de dados e conseguir canalizar essas fontes a um local comum, no qual a análise seria feita.

A solução implementada, denominada API Coletor, conseguiu atender às expectativas dos usuários, pois, por meio dela, foi possível traçar um perfil para o cliente da *Datacamp* e assim tomar decisões baseadas nas informações coletadas. A melhoria de áreas vitais ao sistema, assim como descontinuação de funcionalidades não usadas e que demandam manutenção para compatibilidade, foram os resultados mais evidentes de que a análise de informações impactou o rumo das decisões sobre os produtos e serviços oferecidos pela empresa.

Apesar de já atender à necessidade imediata dos gestores de produto, pontos de melhoria já foram percebidos, tais como coletar o tempo de uso de cada funcionalidade e monitorar quais os relatórios mais usados, a fim de melhorá-los. Além do mais, com a nova onda de inteligências artificiais, é possível pensar em utilizar modelos de aprendizado para catalogar as informações e explicitar padrões, refinando ainda mais as tomadas de decisões.

Quanto ao desenvolvimento profissional do autor deste trabalho, pode-se destacar que, para a implementação da solução, foi necessário aprender a trabalhar com a interoperabilidade de sistemas, assim como conhecer e estudar sobre as ferramentas e modelos para propor soluções viáveis e dinâmicas, como o mercado de software demanda. Dessa forma, foi notável a importância das disciplinas de graduação sobre a concepção de um software e sua utilização. A existência de mais de um software interagindo com outro, como visto em Sistemas Distribuídos, em consonância com o levantamento de requisitos e planejamento de arquitetura, conforme Engenharia de Software. Além disso, o próprio uso e motivação da ferramenta entra em contato com os conteúdos estudados na disciplina de Sistemas de Informação, ao elucidar que esse tipo de sistema obtém dados e deve então prover informações para seus usuários.

REFERÊNCIAS

BUNGART, José Wagner. **Redes de computadores: Fundamentos e protocolos**. Editora SESI-Serviço Social da Indústria, 2017.

BUSINESSWIRE. **Crescimento mais rápido da nuvem no Brasil expande o ecossistema AWS.** 2023 Disponível em: <https://www.businesswire.com/news/home/20230118005280/pt>.

DA SILVA, JOSE HUMBERTO NUNES **Os benefícios da implantação de um software de automação comercial: um estudo de caso em uma microempresa de autopeças na cidade de Timon-MA**. 2018. 72 p. Monografia (Bacharelado em Engenharia de Produção)–Centro Universitário UNINOVAFAPI, Teresina, 2018.

DA SILVEIRA, Paulo R; SANTOS, Winderson E. **Automação e Controle Discreto**. [Digite o Local da Editora]: Editora Saraiva, 2009. E-book. ISBN 9788536518145. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788536518145/. Acesso em: 23 dez. 2022

DE OLIVEIRA, Henrique Eduardo Machado. Aplicativo Cliente-Servidor multicamadas para controle de uma rede de lojas via WEB utilizando Java. **Departamento de Informática e Estatística-Universidade Federal de Santa Catarina (UFSC). Pág**, p. 10-29, 2003.

EXPRESS. **Home** 2023. Disponível em: <https://expressjs.com/pt-br>.

FERRARI, Alberto; RUSSO, Marco. **Introducing Microsoft Power BI**. Microsoft Press, 2016.

FYDORENCHYK, Tetiana. **Tornando o complexo organismo Jelastic em algo simples**. 2017. Disponível em: <https://imasters.com.br/cloud/tornando-o-complexo-organismo-jelastic-em-algo-sim

ples>.

GAMMA, Erich. **Padrões de projetos: soluções reutilizáveis**. Bookman editora, 2009.

GITHUB. **Npm is joining github** 2020. Disponível em: <https://github.blog/2020-03-16-npm-is-joining-github/>.

GOLDEN. **Rust (programming language)** 2022. Disponível em: <https://golden.com/wiki/Rust_(programming_language)-E4RE3M>.

KLABNIK, Steve; NICHOLS, Carol. **The Rust programming language**. No Starch Press, 2023.

MERCADOECONSUMO. **Maioria das empresas tem dificuldades na gestão de dados**. 2019. Disponível em: <https://mercadoeconsumo.com.br/01/07/2019/gestao/maioria-das-empresas-tem-dificuldades-na-gestao-de-dados>.

MICROSOFT. **Power BI Overview** 2023. Disponível em: <https://learn.microsoft.com/pt-br/power-bi/fundamentals/power-bi-overview>.

NODE. **About**. 2023. Disponível em: <https://nodejs.org/en/about>.

POSTGRESQL. **About**. 2023. Disponível em: <https://www.postgresql.org/about>.

PRISMA. **Concepts**. 2023. Disponível em: <https://www.prisma.io/docs/concepts>.

SABINO, Vanessa; KON, Fabio. **Licenças de software livre: história e características.** 2009.

TEIXEIRA, Pedro. **Professional Node. js: Building Javascript based scalable software**. John Wiley & Sons, 2012.

WIKIPEDIA. **Virtuozzo (company)** 2023. Disponível em: <https://en.wikipedia.org/wiki/Virtuozzo_(company)>.