Explorando a Arquitetura REST: Estudo de caso sobre APIs para sistemas de monitoramento web

Gabriel M. S. Zachari, André C. da Silva

Grupo de Pesquisa Mobilidade e Novas Tecnologias de Interação
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas
Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP)
Campus Hortolândia – SP – Brasil

gabriel.zachari@aluno.ifsp.edu.br, andre.constantino@ifsp.edu.br

Abstract. *In the era of increasing digital advances, web systems have become indispensable in several aspects, facilitating communication, access to information and the provision of services, such as monitoring services. However, for these systems to fully meet the demands of modern society, it is crucial that they follow standards that aim to optimize performance and performance. In this study, we explore the REST architecture and related concepts through a case study considering the development of APIs for integrations aimed at web systems, focusing on sustainable applications for data monitoring.*

Resumo. *Na era dos crescentes avanços digitais, os sistemas web tornaram-se indispensáveis em diversos aspectos, facilitando a comunicação, o acesso à informação e a prestação de serviços, como em serviços de monitoramento. No entanto, para que esses sistemas atendam plenamente às demandas da sociedade moderna, é crucial que sigam padrões que visem à otimização de desempenho e performance. Neste estudo, exploramos a arquitetura REST e conceitos relacionados por meio de um estudo de caso considerando o desenvolvimento de API's para integrações destinadas a sistemas web, focando em aplicações sustentáveis para monitoramento de dados.*

1. Introdução

Segundo Sommerville (2011, pág. 355), “Na década de 1990, o desenvolvimento *Web* revolucionou a troca de informações organizacionais. Os computadores de clientes poderiam acessar informações em servidores remotos fora de suas próprias organizações”. Com a evolução das tecnologias na *Web*, e com a necessidade cada vez maior de integração entre aplicações, surge a necessidade de desenvolver aplicações que forneçam uma API (*Application Programming Interface*), como afirmam Silveira *et al.* (2012, pág. 193): “Sistemas sem uma API externa vivem isolados em silos de dados, algo cada vez mais raro, já que a necessidade de se integrar com outros sistemas é crescente”. Esses dados estando disponíveis, torna possível realizar análises e monitoramento em tempo real, importante para uma rápida resposta da empresa como afirma Baquero *et al.* (2016): “A análise e monitoração atempada dos processos de negócio são essenciais para identificar situações de não conformidade e reagir imediatamente a essas inconsistências, de forma a responder rapidamente aos concorrentes".

Sobre API *REST*, Newman (2022) descreve que “é um serviço que pode ser atribuído de maneira independente com base em um domínio de negócio. Onde hospeda

funcionalidades de negócios em um ou mais *endpoints* [1]da rede, utilizando os protocolos mais apropriados". Contudo, ao desenvolver uma API, é importante considerar os requisitos não funcionais para garantir a qualidade do sistema e permitir as análises e monitoramentos desejados. Em uma arquitetura cliente-servidor onde as responsabilidades são separadas entre dois grandes componentes (o cliente e o servidor), o usuário tem como objetivo realizar a interação com a interface no cliente, enquanto o servidor é responsável pelo processamento e armazenamento dos dados dispondo seus serviços por intermédio de uma API.

Neste contexto, propõe-se expor os conhecimentos obtidos no estágio em andamento que envolve a construção de APIs e um protótipo *Mobile* para monitoramento de dados em painéis dinâmicos *(dashes)* com dados em tempo real. Essa solução busca atender às necessidades de serviços que utilizam o ENEM (Exame Nacional do Ensino Médio) como fonte de dados, a fim de oferecer uma plataforma acessível, ágil e customizável para análise de dados, possibilitando uma tomada de decisão mais assertiva, melhorando a experiência do cliente e otimizando os processos internos.

São objetivos específicos deste trabalho:

- Analisar a viabilidade e aplicabilidade do uso de APIs *REST* (*Representational State Transfer*) em conjunto com *Spring Boot* para o monitoramento de dados, sendo esses dados armazenados em um banco de dados relacional *(SQL Server*);
- Implementar um exemplo de API *REST* utilizando *Spring Boot* que disponibilize dados para painéis dinâmicos;
- Criação de um protótipo, a fim de demonstrar o uso de uma API para a criação de um painel de monitoração.

Neste trabalho, iremos abordar na Seção 2 os conceitos principais de desenvolvimento *web* utilizando APIs *REST* e a arquitetura BFF (*Back-end for Front-end*), além de discutir as ferramentas e tecnologias que possibilitam a análise e o monitoramento de dados em tempo real. Na Seção 3, apresentamos a metodologia empregada, o estudo de caso. Na Seção 4 apresentamos e analisamos exemplos práticos e correlatos que ilustram a implementação e os benefícios dessas tecnologias. Finalizamos com as considerações que são apresentadas na Seção 5.

2. Referencial Teórico

Nesta Seção abordaremos os conceitos básicos ao qual este trabalho articula, que são arquitetura *REST*, conceitos de API-BFF, e monitoramento de dados.

2.1. Arquitetura *REST*

A Arquitetura *REST* (*Representational State Transfer*) é um modelo para o desenvolvimento de software distribuído na *Web*, apresentado por Roy Fielding em 2001. Segundo Fielding e Taylor (2002), a estrutura distribuída da *Web* foi crucial para seu crescimento expressivo, permitindo aplicações em larga escala.

---

[1]Um *endpoint* é uma URL de uma API que permite enviar ou receber dados, sendo associado a um método HTTP e a uma funcionalidade específica, como buscar ou atualizar informações.

Esse modelo foi concebido para atender às demandas de sistemas distribuídos, baseando-se em princípios como interface uniforme, separação entre cliente e servidor, ausência de estado (*stateless*) e capacidade de *cache* nos recursos (*cacheable*). Esses princípios garantem que os sistemas sejam escaláveis, simples e eficientes.

A utilização desse tipo de arquitetura oferece diversas vantagens para sistemas *Web* modernos, como o uso de padrões abertos (HTTP), o que a torna uma escolha comum para o desenvolvimento de APIs. Conforme Newman (2022) aponta, essas APIs permitem uma comunicação leve e eficiente, essencial para ambientes *Web* dinâmicos.

Essa abordagem simplifica a criação de serviços *Web*, permitindo que diferentes sistemas se comuniquem de maneira uniforme. Fielding e Taylor (2002) destacam que a natureza distribuída da *Web* é ideal para aplicações em grande escala na Internet. Além disso, a arquitetura *REST* promove a reutilização de componentes e a modularidade, facilitando a comunicação entre sistemas. Richardson e Ruby (2007) observam que a adoção desse modelo permite construir sistemas flexíveis e fáceis de manter, aproveitando ao máximo os recursos da *Web*.

As APIs baseadas nesse modelo oferecem benefícios como escalabilidade, simplicidade, flexibilidade e desempenho. Segundo Newman (2022), a possibilidade de armazenar respostas em *cache* pode melhorar significativamente a performance das aplicações.

2.2. API - BFF (*Back-end for Front-end*)

As APIs, por padrão, são construídas de forma genérica para atender a diversas aplicações e maximizar a utilidade de suas funcionalidades. No entanto, essa generalização pode criar dificuldades para aplicações com características muito específicas. Nesse contexto, a arquitetura *Back-end for Front-end* (BFF) torna-se de grande importância.

De acordo com Richardson (2018), a arquitetura BFF é um padrão arquitetural que resolve as limitações das APIs tradicionais ao atender às necessidades específicas de diferentes interfaces de usuário. A BFF atua como um intermediário entre o *front-end* e os serviços de *back-end*, proporcionando uma camada adicional de abstração que facilita a personalização e otimização das funcionalidades oferecidas aos usuários finais.

A principal vantagem dessa abordagem é permitir que cada tipo de cliente (*web*, *mobile*, etc.) tenha uma API dedicada, adaptada às suas necessidades específicas. Fowler e Lewis (2019) destacam que, ao separar as preocupações de diferentes clientes, as equipes de desenvolvimento podem se concentrar em entregar a melhor experiência possível para cada interface, sem comprometer a performance ou a segurança do sistema.

Além disso, a arquitetura melhora a performance das aplicações ao processar e otimizar dados antes de enviá-los ao cliente. Isso reduz a carga sobre os serviços de *back-end* e melhora a eficiência da comunicação entre cliente e servidor. Conforme explicado por Thönes (2015), essa abordagem é especialmente útil em aplicações que requerem uma interface de usuário altamente responsiva e adaptável.

A implementação dessa arquitetura envolve a criação de APIs específicas para cada tipo de cliente, que agregam, filtram e transformam os dados conforme necessário. De acordo

com Gollob e Fenton (2020), essa abordagem promove a personalização, a flexibilidade e a modularidade, facilitando a manutenção e evolução do sistema. Em termos de segurança, a BFF oferece uma camada adicional de proteção, implementando controles rigorosos de autenticação e autorização antes que as solicitações cheguem aos serviços de *back-end*.

Portanto, a arquitetura *Back-end for Front-end* se apresenta como uma solução poderosa para superar as limitações das APIs tradicionais, oferecendo personalização, performance e segurança aprimoradas. Conforme destacam Richardson (2018) e Thönes (2015), a adoção dessa abordagem pode transformar a maneira como as APIs são desenvolvidas e consumidas, proporcionando uma experiência de usuário superior e uma maior eficiência operacional.

2.3. Monitoramento de Dados

A monitorização tornou-se um termo abrangente cujo significado varia conforme o contexto. De maneira geral, refere-se ao processo de obtenção de informações sobre o estado de um sistema. Segundo Sari e Honor (2012), o monitoramento é essencial para garantir a disponibilidade e o desempenho dos sistemas de TI, sendo definido como "o processo de manter a vigilância sobre a existência e magnitude das mudanças de estado e do fluxo de dados em um sistema". Esse processo visa identificar falhas e auxiliar na sua correção.

O monitoramento pode ser realizado de forma proativa e reativa. O monitoramento proativo envolve a observação de indicadores visuais, como gráficos de séries temporais e *dashboards*, para prever problemas antes que se tornem críticos. Conforme Sari e Honor (2012), essa abordagem permite aos administradores identificar padrões e tendências. O monitoramento reativo, por outro lado, utiliza mecanismos automatizados para enviar notificações, conhecidas como alertas, aos operadores sobre mudanças críticas no estado do sistema.

Turnbull (2014), em seu livro *The Art of Monitoring*, complementa que o monitoramento abrange ferramentas e processos pelos quais se mede e gerencia sistemas de TI, fornecendo "a tradução entre o valor do negócio e as métricas geradas pelos seus sistemas e aplicações". Isso significa que as métricas são convertidas em uma experiência mensurável para o usuário, oferecendo *feedback* tanto para a empresa quanto para a TI sobre a qualidade do serviço.

A interpretação do termo monitoramento pode variar. Em discussões técnicas, algumas pessoas usam o termo para se referir ao processo de medição, que pode não envolver interação humana. Sari e Honor (2012) destacam a importância de entender o contexto em que o termo é usado para aplicar corretamente as práticas de monitoramento. Entre os usos mais comuns estão a supervisão do fluxo de dados e o acompanhamento das mudanças no sistema, visando identificar falhas e auxiliar na sua posterior eliminação.

Em resumo, o monitoramento e o alerta são componentes essenciais de um sistema de gestão eficaz. O monitoramento contínuo permite a detecção precoce de problemas e a implementação de medidas corretivas antes que afetem os usuários finais. Conforme destacado por Sari e Honor (2012) e Turnbull (2014), um sistema de monitoramento bem implementado integra técnicas de processamento em tempo real, estatística e análise de dados

para fornecer uma visão abrangente do estado do sistema, garantindo assim sua confiabilidade e eficiência.

3. Metodologia

O método escolhido para possibilitar a discussão sobre a viabilidade de emprego de API *REST* para monitoramento de dados será um estudo de caso. O estudo de caso é um método de pesquisa que utiliza, geralmente, dados qualitativos, coletados a partir de eventos reais, com o objetivo de explicar, explorar ou descrever fenômenos atuais inseridos em seu próprio contexto (Eisenhardt, 1989). Com essa estratégia é possível se aprofundar no tema escolhido, ou seja, construção de API para monitoração de dados de diferentes aplicações.

Segundo Formighieri (2023), existem três tipos de estudo de caso:

- Exploratório: Se dá quando o tema é algo que está em seu começo, onde o objetivo é realizar levantamento de informações, para que posteriormente, caso haja continuidade, partir para uma pesquisa científica;
- Descritivo: É utilizado quando se requer uma pesquisa mais detalhada, captando dados qualitativos e quantitativos. Essas pesquisas são realizadas através de questionários, entrevistas e outros métodos;
- Analítico: Apresentado em cenários de que se deseja reunir dados para que seja gerada discussões, com o intuito de gerar *insights* e questionamentos sobre o mesmo.

Este trabalho empregará a vertente analítica, construindo cenários de implementação de API para disponibilizar dados para painéis dinâmicos. Na discussão pretende-se empregar *Spring Boot* e API *REST*.

Para a criação de um cenário fictício, devido a impossibilidade de expor dados da empresa concedente, foi construída uma API *REST* na linguagem de programação Java (*Spring Boot*), usando como fonte de dados um arquivo CSV (*comma-separated values*) com dados do ENEM de 2023 disponibilizado pelo INEP (Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira), e utilizando como ferramentas o Insomnia, *Json to Pojo*, Intellij IDEA, Android Studio, descritos brevemente a seguir o propósito de uso:

- Insomnia: Realizar os testes das saídas da API;
- *Json to Pojo*: Com essa ferramenta é possível fazer as modelagens de classes da API que será desenvolvida;
- Intellij IDEA: Ambiente de desenvolvimento para criar a API na linguagem de programação Java;
- Android Studio: Ambiente de desenvolvimento para criar o protótipo *mobile* na linguagem de programação Kotlin.

4. Desenvolvimento

Esta Seção tem como objetivo descrever as etapas do desenvolvimento deste trabalho, os protótipos criados e os objetivos da monitoração.

4.1. Visão geral da arquitetura

O presente trabalho tem como objetivo a construção de um protótipo de sistema de monitoramento do Exame Nacional do Ensino Médio (ENEM), utilizando uma arquitetura baseada em API *REST*. Para isso, foi desenvolvida uma API em Java com *Spring Boot*, responsável pela ingestão, processamento e disponibilização de dados extraídos de arquivos CSV fornecidos pelo INEP. Os dados são armazenados temporariamente em *cache* para otimizar o desempenho da aplicação e são expostos por meio de *endpoints* GET, garantindo acessibilidade ao *front end mobile*, desenvolvido para o sistema Android.

A arquitetura proposta segue os princípios do desenvolvimento desacoplado, onde o *back-end* e o *front-end* operam de maneira independente, se comunicando exclusivamente por meio da API *REST*. Isso garante maior flexibilidade e escalabilidade ao sistema, permitindo futuras expansões e adaptações.

4.2. Diagrama de sequência

O fluxo do programa (Figura 1) inicia quando a aplicação cliente realiza uma requisição HTTP GET para a API, que é recebida pelo *Controller* e repassada para a camada de *Service*, responsável por processar a solicitação e verificar se os dados já estão armazenados em *cache*. Caso não estejam, o DTO solicita os dados ao *Consumer*, que faz a coleta diretamente do arquivo CSV. Após a extração, os dados retornam ao DTO, que os armazena em *cache* e os encaminha para o *Service*, que por sua vez faz as regras de negócio e os envia ao *Controller*. Finalmente, o *Controller* retorna a resposta ao usuário via HTTP *Response*, garantindo um fluxo otimizado com menor latência e maior eficiência no acesso às informações.

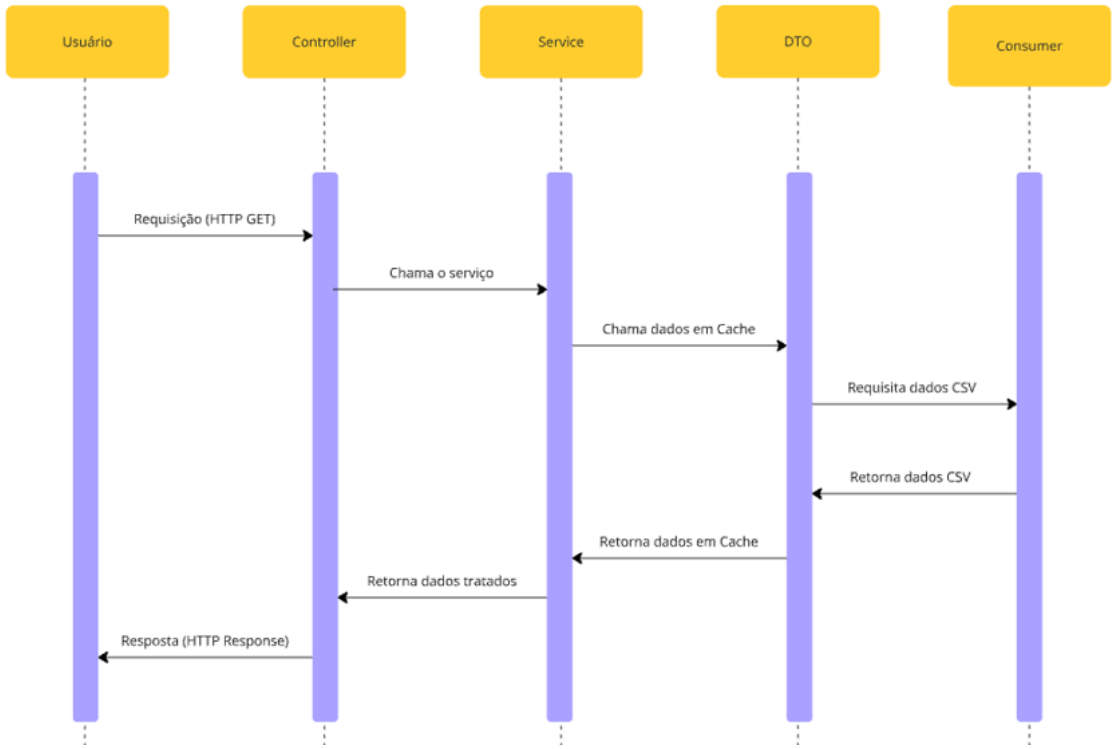


Figura 1. Diagrama de sequência exemplificando o tratamento de uma requisição realizada pelo cliente.

4.3. Base de dados

A coleta dos dados do Exame Nacional do Ensino Médio (ENEM) é realizada a partir de arquivos no formato CSV disponibilizados pelo INEP. Esses arquivos contêm 14 colunas e 1.048.576 linhas, sendo elas informações detalhadas sobre os participantes do exame, incluindo dados demográficos, presença nas provas e notas obtidas. A Figura 2 apresenta parcialmente os dados em formato de tabela ao abrir o arquivo CSV usando uma ferramenta de planilha eletrônica.

	A	B	C	D	E	F	G
1	NU_INSCRICAO	TP_FAIXA_ETARIA	TP_SEXO	TP_ESTADO_CIVIL	TP_COR_RACA	TP_NACIONALIDADE	TP_PRESENCA_CN
2	2,10059E+11	14	M	2	1	1	0
3	2,1006E+11	12	M	2	1	0	0
4	2,10061E+11	6	F	1	1	1	1
5	2,1006E+11	2	F	1	3	1	1
6	2,1006E+11	3	F	1	3	1	1
7	2,10058E+11	6	F	1	3	1	0
8	2,1006E+11	11	F	1	3	1	0
9	2,10058E+11	11	M	1	3	1	0
10	2,10059E+11	5	F	1	2	1	0
11	2,10061E+11	11	M	1	1	1	1
12	2,10059E+11	8	M	1	3	1	1
13	2,10061E+11	3	M	1	3	4	1
14	2,1006E+11	6	M	1	3	1	0
15	2,1006E+11	3	F	1	1	1	1
16	2,10059E+11	11	F	1	2	1	1
17	2,1006E+11	7	F	1	3	1	1
18	2,10058E+11	4	F	1	1	1	1
19	2,1006E+11	11	F	1	1	1	1
20	2,10058E+11	11	M	1	1	1	1
21	2,10059E+11	11	M	1	1	1	0
22	2,10061E+11	12	F	1	2	1	1
23	2,1006E+11	4	F	1	1	2	0
24	2,10061E+11	9	F	1	1	1	1
25	2,1006E+11	5	F	1	1	1	0
26	2,10059E+11	3	F	1	1	1	1
27	2,1006E+11	8	F	0	3	1	1
28	2,10058E+11	4	F	1	3	1	1
29	2,1006E+11	11	M	1	1	1	0
30	2,10058E+11	5	F	1	3	1	1
31	2,10059E+11	13	F	0	3	1	1

Figura 2. Planilha apresentando parcialmente os dados do ENEM de 2023.

O dicionário de dados é um repositório estruturado que documenta as informações essenciais de um sistema, como nome dos atributos, tipos de dados e restrições, garantindo padronização, integridade e fácil manutenção. No contexto deste projeto, ele é fundamental para interpretar corretamente a planilha CSV fornecida pelo INEP, pois descreve o significado de cada coluna, como inscrição, faixa etária, notas e presença nas provas. Essa documentação assegura que os dados sejam extraídos, processados e disponibilizados corretamente pela API *REST*, evitando erros e inconsistências no monitoramento do ENEM. A Figura 3 mostra parcialmente o dicionário de dados disponibilizado pelo INEP.

4.4. Coleta dos dados

Para garantir um processamento eficiente, foi desenvolvida a classe *Consumer*, que implementa a leitura, extração e estruturação dos dados antes de disponibilizá-los por meio da API *REST*. A implementação foi feita utilizando a biblioteca *Apache Commons* CSV, que permite uma manipulação otimizada dos registros. O fluxo do processamento pode ser dividido nas seguintes etapas:

1. Leitura do arquivo CSV, garantindo a codificação UTF-8 para evitar problemas com caracteres especiais;
2. Mapeamento dos cabeçalhos, assegurando a estrutura correta dos dados;

DICIONÁRIO DE VARIÁVEIS - ENEM 2023

NOME DA VARIÁVEL	Descrição	Variáveis Categóricas		Tamanho	Tipo
		Categoria	Descrição		
DADOS DO PARTICIPANTE					
NU_INSCRICAO	Número de inscrição[1]			12	Numérica
NU_ANO	Ano do Enem			4	Numérica
TP_FAIXA_ETARIA	Faixa etária[2]	1	Menor de 17 anos	2	Numérica
		2	17 anos		
		3	18 anos		
		4	19 anos		
		5	20 anos		
		6	21 anos		
		7	22 anos		
		8	23 anos		
		9	24 anos		
		10	25 anos		
		11	Entre 26 e 30 anos		
		12	Entre 31 e 35 anos		
		13	Entre 36 e 40 anos		
		14	Entre 41 e 45 anos		
		15	Entre 46 e 50 anos		
		16	Entre 51 e 55 anos		
		17	Entre 56 e 60 anos		
		18	Entre 61 e 65 anos		
		19	Entre 66 e 70 anos		
		20	Maior de 70 anos		
TP_SEXO	Sexo	M	Masculino	1	Alfanumérica
		F	Feminino		
TP_ESTADO_CIVIL	Estado Civil	0	Não informado	1	Numérica
		1	Solteiro(a)		
		2	Casado(a)/Mora com companheiro(a)		
		3	Divorciado(a)/Desquitado(a)/Separado(a)		

Figura 3. Dicionário de dados para o arquivo CSV com os dados do ENEM 2023.

3. Tratamento de inconsistências, como a remoção de colunas vazias e possíveis erros na formatação;
4. Extração e conversão dos registros, transformando-os em instâncias da classe CsvModel;
5. Armazenamento temporário em uma lista, possibilitando a manipulação eficiente dos dados antes de serem expostos via API.

A classe *Consumer* é responsável por processar o arquivo CSV e estruturar os dados para o sistema. O trecho de código da Figura 4 mostra a estrutura geral da classe.

```
18  @Service
19  public class Consumer {
        1 usage
20      public List<CsvModel> getDadosCsv(String filePath) throws IOException {
21
22          try (Reader reader = Files.newBufferedReader(Paths.get(filePath), StandardCharsets.UTF_8);
23               CSVParser csvParser = new CSVParser(reader, CSVFormat.DEFAULT
24                       .withDelimiter(';')  // Corrige o delimitador para ponto e vírgula
25                       .withFirstRecordAsHeader()
26                       .withIgnoreEmptyLines(true)
27                       .withTrim(true))) {
28
```

Figura 4. Classe para consumo dos dados da planilha.

A anotação *@Service* (linha 18) define a classe como um componente gerenciado pelo *Spring Boot*, permitindo que seja injetada em outros pontos da aplicação. A função *getDadosCsv()* (linha 20) recebe o caminho do arquivo como parâmetro e inicia a leitura

utilizando *Files.newBufferedReader()*, garantindo compatibilidade com diferentes sistemas operacionais. Na sequência, a biblioteca *Apache Commons* CSV é utilizada para instanciar um *CSVParser*, configurado com delimitador “;”, remoção de linhas vazias e utilização da primeira linha como cabeçalho.

Após a leitura do arquivo, os cabeçalhos das colunas são obtidos e validados para evitar inconsistências, como colunas vazias (Figura 5).

```
// Verifica se existe algum cabeçalho em branco e o remove
if (headerMap.containsKey("")) {
    System.err.println("Aviso: Uma coluna sem nome foi encontrada no CSV.");
    headerMap.remove( key: ""); // Remove colunas sem nome
}
```

Figura 5. Tratativa para espaços vazios.

A iteração sobre os registros do CSV ocorre dentro de um *loop for*, onde os dados são extraídos e armazenados na lista *batch* (Figura 6).

```
// Itera sobre os registros do arquivo CSV
for (CSVRecord record : csvParser) {
    try {

        // Extrai os dados do registro CSV e cria uma instância de CsvModel
        String nuInscricao = record.get("NU_INSCRICAO");
        String faixaEtaria = record.get("TP_FAIXA_ETARIA");
        String sexo = record.get("TP_SEXO");
        String estadoCivil = record.get("TP_ESTADO_CIVIL");
        String corRaca = record.get("TP_COR_RACA");
        String nacionalidade = record.get("TP_NACIONALIDADE");
        String escola = record.get("TP_ESCOLA");
        String presencaCn = record.get("TP_PRESENCA_CN");
        String presencaCh = record.get("TP_PRESENCA_CH");
        String presencaLc = record.get("TP_PRESENCA_LC");
        String presencaMt = record.get("TP_PRESENCA_MT");
        String notaCn = record.get("NU_NOTA_CN");
        String notaCh = record.get("NU_NOTA_CH");
        String notaLc = record.get("NU_NOTA_LC");
        String notaMt = record.get("NU_NOTA_MT");
        String ufProva = record.get("SG_UF_PROVA");
```

Figura 6. Trecho de código da classe *Controller* para a captura do nome das colunas.

Cada campo do CSV é acessado utilizando o nome da coluna, garantindo que os dados sejam corretamente mapeados para a estrutura do sistema. Os atributos extraídos incluem:

- Identificação do participante (NU_INSCRICAO);
- Características demográficas (TP_FAIXA_ETARIA, TP_SEXO, TP_ESTADO_CIVIL, TP_COR_RACA, TP_NACIONALIDADE);
- Presença nas provas (TP_PRESENCA_CN, TP_PRESENCA_CH, TP_PRESENCA_LC, TP_PRESENCA_MT);

- Notas obtidas (NU_NOTA_CN, NU_NOTA_CH, NU_NOTA_LC, NU_NOTA_MT).
- Unidade federativa da prova (SG_UF_PROVA).

4.5. Processamento e Disponibilização dos Dados

Após a coleta dos dados do arquivo CSV, o sistema segue um fluxo estruturado que inclui armazenamento temporário, aplicação de regras de negócio e exposição dos dados via API *REST* (conforme fluxo apresentado na Figura 1). O processo ocorre em três principais etapas:

1. Armazenamento em *Cache*: Os dados extraídos são armazenados temporariamente para otimizar a performance e reduzir leituras repetitivas do arquivo CSV;
2. Aplicação de Regras de Negócio: Os dados são processados para cálculos estatísticos, filtragem e organização conforme a necessidade do sistema;
3. Disponibilização via API *REST*: Os dados processados são acessíveis por meio de *endpoints*, permitindo que o *frontend mobile* os consuma de maneira eficiente.

4.5.1. Armazenamento Temporário em Cache

Para evitar leituras recorrentes do arquivo CSV e melhorar o desempenho da aplicação, os dados coletados são armazenados em *cache* utilizando o Spring *Cache*. Isso reduz o tempo de resposta da API, uma vez que as informações são recuperadas diretamente da memória em chamadas subsequentes.

A classe CsvDTO é responsável por essa etapa. O método *getDadosEnem()* carrega os dados apenas uma vez e os mantém disponíveis em cache (Figura 7).

```
18      @Cacheable(value = "dadosCsv")
19      public List<CsvModel> getDadosEnem() {
20
21          try {
22              String path = "D:\\faculdade\\6° sem\\microdados_enem_2023\\DADOS\\MICRODADOS_ENEM_2023.csv";
23              return consumer.getDadosCsv(path);
24          } catch (Exception e) {
25              e.printStackTrace();
26          }
27          return null;
28      }
```

Figura 7. Armazenar os dados em *cache*.

A anotação *@Cacheable (value = "dadosCsv")* (linha 18) garante que os dados fiquem armazenados na memória até que o *cache* seja invalidado ou atualizado.

4.5.2. Aplicação de Regras de Negócio

Com os dados carregados, são aplicadas regras de negócio para filtragem, cálculos estatísticos e organização dos registros. A classe *Service* (Figura 8) implementa essas regras, processando as informações e preparando os dados para exposição via API.

Um exemplo é o método *getTop10Regioes()*, que calcula a média das notas e retorna os 10 melhores desempenhos em uma determinada região.

```
20      @Cacheable(value = "dadosRegiao")
21      public Object getTop10Regioes(String regiao) {
22
23          List<CsvModel> listDado = new ArrayList<>();
24          List<CsvModel> dadosFiltrados = new ArrayList<>();
25          try {
26              listDado = dadosDto.getDadosEnem();
27
28              if (regiao.isEmpty()) {
29                  dadosFiltrados = listDado.stream()
30                          .filter(it -> isValidNotas(it))
31                          .sorted(Comparator.comparingDouble(this::calcularMediaNotas).reversed())
32                          .limit( maxSize: 10) //
33                          .collect(Collectors.toList());
34              } else {
35                  dadosFiltrados = listDado.stream()
36                          .filter(it -> Objects.equals(it.getUfProva(), regiao))
37                          .filter(it -> isValidNotas(it))
38                          .sorted(Comparator.comparingDouble(this::calcularMediaNotas).reversed())
39                          .limit( maxSize: 10)
40                          .collect(Collectors.toList());
41              }
42
43              return dadosFiltrados;
44          } catch (Exception e) {
45              e.printStackTrace();
46          }
47          return null;
48      }
```

Figura 8. Exemplo de método na classe *Service* para tratativas de dados.

Nesse fluxo:

- Os dados são carregados do *cache* (*dadosDto.getDadosEnem()*) (linha 26);
- São filtrados por região (se fornecida pelo usuário) (linha 36);
- Apenas registros com notas válidas são considerados (*isValidNotas()*) (linha 30 e 37);
- Os dados são ordenados pela média das notas (*calcularMediaNotas()*) (linha 31 e 38) e limitados aos 10 melhores (linha 32 e 39).

Essa lógica permite que os usuários acessem rapidamente os dados processados sem necessidade de cálculos repetitivos.

4.5.3. Aplicação de Regras de Negócio

A última etapa do fluxo é a exposição dos dados por meio da API *REST*. A classe *DadosController* define os *endpoints* HTTP que permitem ao *front-end* acessar as informações processadas.

O *endpoint /regioes-top-10* disponibiliza os 10 melhores desempenhos por região, cujo código está exposto na Figura 9. Esse método realiza as seguintes operações:

Figura 9. *Controller* para disponibilizar o *endpoint /regioes-top-10*.

1. Chama *service.getTop10Regioes(regiao)*, que processa os dados (linha 54);
2. Retorna os resultados com status HTTP 200 (OK) se a requisição for bem-sucedida (linha 57);
3. Em caso de erro, responde com status HTTP 500 (INTERNAL SERVER ERROR) (linha 59).

Esse *endpoint* permite que aplicações externas consultem os dados processados de forma rápida e eficiente. Demais *endpoints* estão expostos na Tabela 1.

Tabela 1. *Endpoints* para as funcionalidades disponibilizadas pelo *back-end*.

Funcionalidades	***Endpoint***	**Método HTTP**
Presença por matéria	/dados/presenca	*GET*
Cor/Raça	/dados/cor-raca	*GET*
Gráfico de idade	/dados/quantidade-idade	*GET*
Lista de Estados	/dados/estados	*GET*
Top 10 notas	/dados/regioes-top-10	*GET*
Separação por gênero	/dados/genero	*GET*

4.6. Resultados de Desempenho

Para avaliar a eficiência da API, foram realizadas medições no tempo de resposta das requisições antes e depois da implementação do *cache*. Inicialmente, as requisições dependiam da leitura direta do arquivo CSV, resultando em tempos mais altos devido ao grande volume de dados. Com a utilização do *Spring Cache*, os dados passaram a ser armazenados temporariamente, reduzindo significativamente o tempo de resposta nas chamadas subsequentes.

Os testes demonstraram que, após o primeiro carregamento, as requisições processadas via *cache* apresentaram uma redução expressiva na latência, tornando o sistema mais rápido e eficiente para consultas frequentes. Os resultados detalhados serão

apresentados a seguir (Figura 10), onde o tempo de resposta diminui de 14.2 segundos para 16.7 milissegundos, destacando o impacto positivo da otimização no desempenho da API.

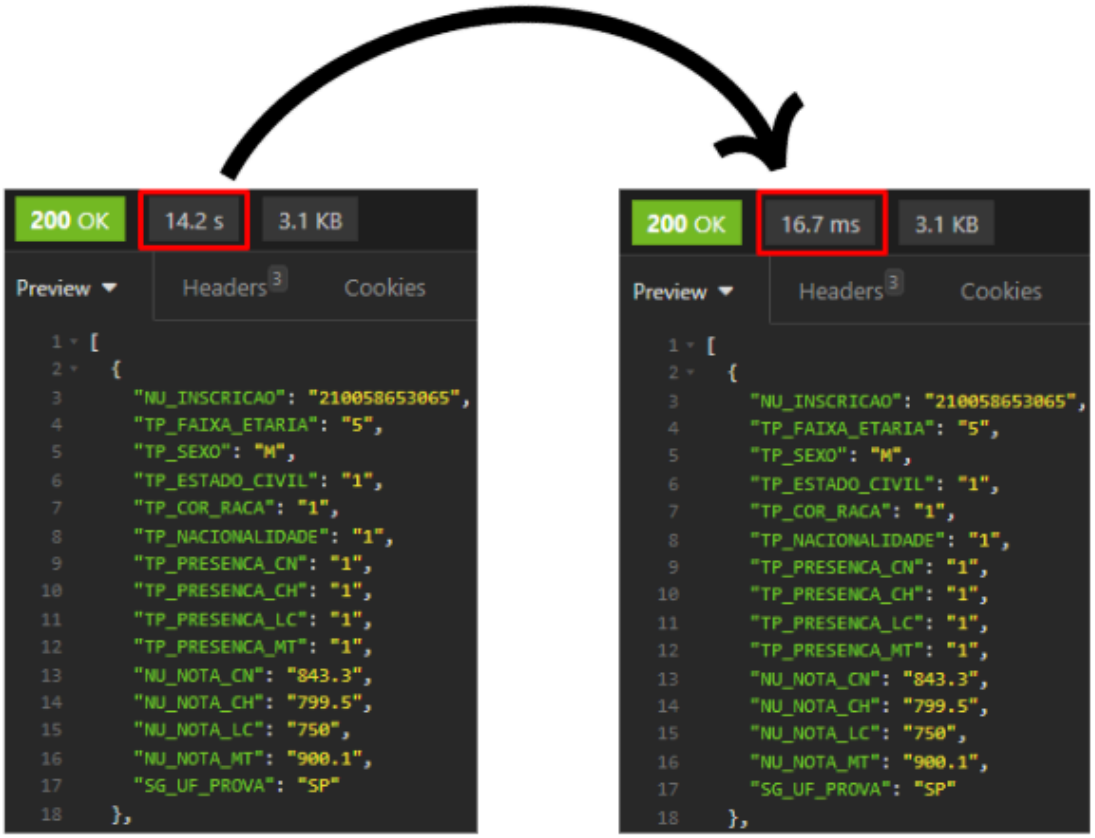


Figura 10. Tempo de resposta após os dados armazenados em *cache*.

4.7. Desenvolvimento do Aplicativo Mobile em Kotlin

O aplicativo *mobile* desenvolvido para Android tem como principal função consumir os dados da API *REST* e exibi-los de forma intuitiva para os usuários. Para isso, foi utilizado Kotlin, juntamente com a biblioteca Retrofit, que facilita a comunicação com serviços *web*.

A estrutura do consumo de dados no aplicativo segue três etapas principais:

1. Configuração do Retrofit: Inicializa a conexão com a API *REST* e define a base URL dos *endpoints*;
2. Definição da Interface de Comunicação (*ApiService*): Mapeia os *endpoints* da API para serem chamados no *app*;
3. Implementação do Repositório (*Repository*): Centraliza as chamadas da API e disponibiliza os dados para o restante da aplicação.

4.7.1. Configuração do Retrofit

A conexão com a API é gerenciada pela classe *NetworkService*, que configura o Retrofit para converter os dados recebidos no formato JSON (Figura 11).

Figura 11. *Retrofit* para fazer a requisição.

Essa configuração permite que o aplicativo se conecte à API *REST* sem necessidade de escrever código manual para converter respostas JSON.

4.7.2. Definição dos *Endpoints* no ApiService

A interface *ApiService* define as requisições HTTP para os dados monitorados pelo *app* (Figura 12). Cada função corresponde a um *endpoint* da API, retornando os dados necessários.

Figura 12. Caminhos das requisições.

Cada método utiliza a anotação @GET para indicar o *endpoint* correspondente, permitindo que os dados sejam recuperados com requisições HTTP GET.

4.7.3. Implementação do Repositório para Consumo de Dados

O *Repository* (Figura 13) centraliza as chamadas de API, facilitando a obtenção dos dados para serem utilizados em outras partes do aplicativo.

Com essa abordagem, qualquer parte do aplicativo pode acessar os dados apenas chamando os métodos do *Repository*, sem precisar configurar diretamente a API.

```
class Repository {

    private val service = NetworkService.getInstance()

    fun getTop10Regioes(regiao: String) = service.getTop10Regioes(regiao)
    fun getQuantidadeIdade() = service.getQuantidadeIdades()
    fun getCorRaca() = service.getCorRaca()
    fun getPresenca() = service.getPresenca()
}
```

Figura 13. Chamada dos *endpoints*.

4.7.4. Fluxo de Consumo de Dados no Aplicativo

O fluxo de consumo dos dados no aplicativo segue os seguintes passos:

1. O usuário solicita uma informação (exemplo: *ranking* das 10 melhores médias por região);
2. O *ViewModel* chama o *Repository*, que encaminha a requisição para a API via Retrofit;
3. A API *REST* responde com os dados processados, que são convertidos automaticamente para objetos Kotlin;
4. O aplicativo exibe os dados na interface de forma clara e acessível.

Esse fluxo garante que os dados sejam acessados de forma eficiente e organizada, mantendo o aplicativo responsivo e dinâmico.

4.8. Interfaces do Aplicativo

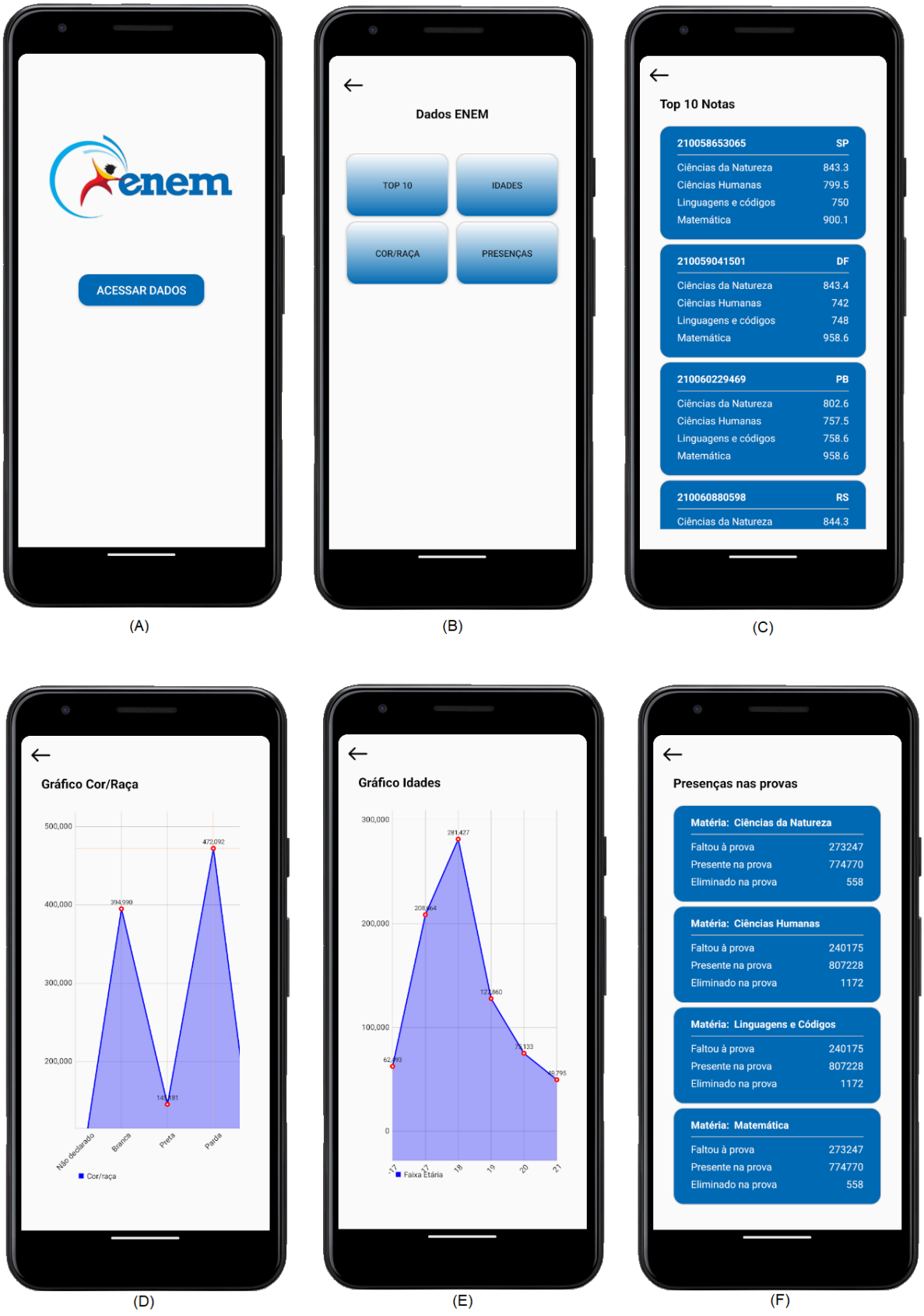
O aplicativo *mobile* foi desenvolvido com foco no monitoramento de dados educacionais do Enem, permitindo a visualização de estatísticas essenciais, como distribuição etária, presença, recorte de cor/raça e análise regional. A interface foi projetada para ser intuitiva e acessível, garantindo que os usuários possam interpretar as informações de maneira clara e eficiente.

Por meio de chamadas à API *REST*, os dados são coletados em tempo real, tratados e armazenados em cache para otimizar a experiência do usuário. A exibição das informações ocorre por meio de gráficos interativos e tabelas dinâmicas, facilitando a análise dos dados e possibilitando a tomada de decisões estratégicas com base nos padrões identificados.

A Figura 14 apresenta as telas do aplicativo, destacando os elementos visuais e funcionais que suportam o monitoramento contínuo dos dados educacionais.

5. Conclusões

O presente trabalho teve como objetivo o desenvolvimento de um protótipo de monitoramento de dados educacionais, demonstrando a eficiência das APIs *REST* na coleta, tratamento e exibição de informações relevantes. A arquitetura do projeto foi estruturada para garantir escalabilidade, flexibilidade e integração com diferentes clientes, permitindo que cada implementação seja adaptada conforme as necessidades específicas de cada usuário.



(A) (B) (C)

(D) (E) (F)

Figura 14. Telas do aplicativo que consome a API *REST* (A) tela inicial, (B) menu de dados, (C) melhores notas, (D) gráfico cor/raça, (E) gráfico idades, e (F) presenças nas provas.

A aplicação desenvolvida possibilita a extração e análise de dados através de um fluxo estruturado, onde as requisições do usuário são processadas pelo *controller*, encaminhadas para o service, armazenadas temporariamente no DTO (*cache*) e, por fim, acessam o *consumer*, que realiza a coleta de dados a partir de arquivos CSV. Esse fluxo garante eficiência na obtenção e atualização das informações, permitindo que os clientes tenham acesso a dados precisos em tempo real.

Os resultados obtidos demonstram que o uso de APIs *REST* proporciona um desempenho otimizado para aplicações de monitoramento, assegurando rapidez na comunicação entre sistemas e na disponibilização dos dados. Além disso, a modularidade da arquitetura facilita futuras expansões e adaptações do sistema, pois cada camada foi projetada de forma independente, permitindo que novos serviços sejam adicionados sem comprometer o funcionamento da aplicação como um todo. Dessa forma, novas fontes de dados podem ser integradas ao *consumer*, melhorias na estratégia de *cache* do DTO podem ser aplicadas para otimizar a performance, e novos *endpoints* podem ser incorporados ao *controller* sem impactar diretamente outras partes do sistema. Além disso, essa modularidade favorece a escalabilidade horizontal, permitindo que diferentes componentes sejam expandidos conforme a demanda, garantindo um alto nível de desempenho e adaptabilidade a diferentes contextos de uso.

Durante o desenvolvimento do projeto, alguns desafios foram enfrentados, principalmente no consumo dos dados, devido ao grande volume de informações presentes nos arquivos CSV, o que exigiu estratégias para otimizar a leitura e o processamento. Além disso, a definição e implementação das regras de negócio demandaram um planejamento criterioso para garantir que os cálculos e filtros aplicados refletissem corretamente os padrões e necessidades do monitoramento educacional.

Apesar do sucesso na implementação do protótipo, há possibilidades de aprimoramento para futuras versões. A integração com bancos de dados mais robustos, a implementação de mecanismos de segurança avançados para proteção dos dados e a criação de dashboards interativos são algumas melhorias que podem ser incorporadas para aumentar a usabilidade e confiabilidade do sistema. Além disso, a adaptação do protótipo para diferentes clientes exigiria um planejamento mais detalhado em conjunto com cada equipe, garantindo a personalização das funcionalidades conforme suas necessidades específicas.

Dessa forma, este trabalho demonstrou a viabilidade de um sistema de monitoramento eficiente baseado em APIs *REST*, destacando sua importância no cenário atual de análise de dados e sua aplicabilidade em diferentes contextos organizacionais. O protótipo desenvolvido representa uma base sólida para futuras melhorias e implementações, contribuindo para o avanço de soluções tecnológicas voltadas à gestão e análise de dados estratégicos.

Referências

EISENHARDT, K. M. Building theories from case study research. Academy of Management Review, New York, v. 14, n. 4, 1989.

FIELDING, Roy Thomas; TAYLOR, Richard N. Principled design of the modern web architecture. ACM Transactions on Internet Technology (TOIT), v. 2, n. 2, p. 115-150, 2002.

FOWLER, Martin; LEWIS, James. Microservices: A definition of this new architectural term. ThoughtWorks, 2019. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 23 jul. 2024.

GMZACHARI. Monitoramento Enem - Front Mobile. GitHub, 2024. Disponível em: https://github.com/GMzachari/Monitoracao-Enem-Front-Mobile. Acesso em: 12 mar. 2025.

GMZACHARI. Monitoramento Enem - Back-end. GitHub, 2024. Disponível em: https://github.com/GMzachari/Monitora-o-Enem. Acesso em: 12 mar. 2025.

GOLLOB, T.; FENTON, J. BFF: The Back-End for Front-End Pattern. Addison-Wesley, 2020.

INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA – INEP. ENEM. 17 nov. 2020. Disponível em: https://www.gov.br/inep/pt-br/acesso-a-informacao/dados-abertos/microdados/enem. Acesso em: 06 mar. 2025.

NEWMAN, Sam. Building Microservices: Designing Fine-Grained Systems. 2. ed. O'Reilly Media, 2022.

RICHARDSON, Chris. Microservices Patterns: With examples in Java. Manning Publications, 2018.

RICHARDSON, Leonard; RUBY, Sam. RESTful Web Services. O'Reilly Media, 2007.

SARI, Z.; HONOR, L. System Monitoring: Techniques and Tools. 2. ed. TechPress, 2012.

THÖNES, Joachim. Microservices. IEEE Software, v. 32, n. 1, p. 116-120, 2015.

TURNBULL, James. The Art of Monitoring. Turnbull Press, 2014.

Documento Digitalizado Público

Anexo I - artigo - TCC

Assunto: Anexo I - artigo - TCC
Assinado por: Andre Constantino
Tipo do Documento: Relatório
Situação: Finalizado
Nível de Acesso: Público
Tipo do Conferência: Documento Digital

Documento assinado eletronicamente por:

- **Andre Constantino da Silva**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 14/03/2025 22:43:05.

Este documento foi armazenado no SUAP em 14/03/2025. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifsp.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 1967849
Código de Autenticação: f1c913842e