UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

REINALDO HASSEL NETO

API DE LEITURA, COMPRESSÃO E DISPONIBILIZAÇÃO DE DADOS PARA *BUSINESS INTELLIGENCE*

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO - PR
2022

REINALDO HASSEL NETO

API DE LEITURA, COMPRESSÃO E DISPONIBILIZAÇÃO DE DADOS PARA *BUSINESS INTELLIGENCE*

API FOR READING, COMPRESSION AND DATA AVAILABILITY FOR BUSINESS INTELLIGENCE

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso, do curso superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná (UTFPR), *Campus* Pato Branco, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. Érick Oliveira Rodrigues.

PATO BRANCO - PR

2022

REINALDO HASSEL NETO

API DE LEITURA, COMPRESSÃO E DISPONIBILIZAÇÃO DE DADOS PARA BUSINESS INTELLIGENCE

Trabalho de conclusão de curso de Graduação apresentado como requisito obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná (UTFPR).

Data da aprovação: 29 de novembro de 2022.

Érick Oliveira Rodrigues
Doutorado
Universidade Tecnológica Federal do Paraná (UTFPR)

Rúbia Eliza de Oliveira Schultz Ascari
Doutorado
Universidade Tecnológica Federal do Paraná (UTFPR)

Eliane Maria De Bortoli Fávero
Doutorado
Universidade Tecnológica Federal do Paraná (UTFPR)

PATO BRANCO - PR
2022

Dedico este trabalho à minha esposa, aos meus pais e aos amigos que sempre me incentivaram e apoiaram.

AGRADECIMENTOS

Agradeço primeiramente ao meu orientador pela dedicação, direcionamento e compromisso durante a realização deste trabalho.

Agradeço a minha esposa e aos meus pais pelo incentivo, apoio incondicional e suporte emocional.

Agradeço aos meus amigos por todo suporte fornecido durante a realização deste trabalho.

Agradeço aos professores componentes da banca por disponibilizarem seu tempo para auxiliar na construção do estudo.

E por fim, agradeço a todos que diretamente ou indiretamente fizeram parte da minha formação acadêmica.

“Tente uma, duas, três vezes e se possível tente a quarta, a quinta e quantas vezes for necessário. Só não desista nas primeiras tentativas, a persistência é amiga da conquista. Se você quer chegar aonde a maioria não chega, faça o que a maioria não faz.”

(Bill Gates)

RESUMO

O *Business Intelligence* (BI) é utilizado para determinar as tomadas de decisões em vários ramos de negócios como saúde, educação, financeiro, comércio e a indústria alimentícia, por exemplo. Esse processo geralmente consiste em basear as decisões em análises de dados já existentes de cenários anteriores e semelhantes ao encontrado no momento atual ou até mesmo entregando o resultado da decisão a ser tomada de forma breve, clara e objetiva, utilizando de gráficos e *dashboards* (painéis de controle) para alcançar o resultado esperado. Diante disso, há pouca exploração dos dados governamentais abertos, que são dados reais de órgãos governamentais disponibilizados para análise e também podendo ser utilizado para análises de BI. O trabalho proposto visa resolver uma problemática muito frequente no armazenamento de arquivos para BI. Em uma única solução a aplicação realizará a compressão de arquivos de fontes de dados para BI e disponibilização do resultado da compressão em uma *cloud* pública, para futura descompressão e utilização dos dados.

Palavras-chave: *business Intelligence;* tomada de decisão; compressão de dados.

ABSTRACT

Business Intelligence (BI) is used to determine decision-making in various branches of business such as health, education, finance, commerce and the food industry, for example. This process usually consists of basing decisions on analysis of existing data from previous scenarios and similar to those found at the current time or even delivering the result of the decision to be taken in a brief, clear and objective way, using graphics and dashboards (panels). control) to achieve the expected result. Given this, there is little exploration of open government data, which are real data from government agencies made available for analysis and can also be used for BI analysis. The proposed work aims to solve a very frequent problem in file storage for BI. In a single solution, the application will compress files from data sources for BI and make the compression result available in a public cloud, for future decompression and use of the data.

Keywords: business Intelligence; decision making; data compression.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface.*
AWS	*Amazon Web Services.*
BI	*Business Intelligence*.
CPU	*Central Processing Unit*.
CQRS	*Command Query Responsibility Segregation*
CRM	*Customer relationship management.*
CSV	*Comma-separated values.*
CVM	Convenção de valores mobiliários
DDD	*Domain Driven Design.*
FPS	*Frames* por segundo.
HTTP	*HyperText Transfer Protocol*.
JSON	*JavaScript Object Notation.*
LZW	*Lempel–Ziv–Welch.*
MNP5	*Microcom Networking Protocol 5*.
PDF	*Portable Document Format*.
REST	*Representational State Transfer.*
RLE	*Run Length Encoding.*.
SOAP	*Simple Object Access Protocol*
SQL	*Structured Query Language.*
S3	*Simple Storage Service.*
URL	*Uniform Resource Locator.*
UFRGS	Universidade Federal do Rio Grande do Sul
UTFPR	Universidade Tecnológica Federal do Paraná
WebAPI	*Web Application Programming Interface.*
XML	*eXtensible Markup Language.*

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

O meio governamental desempenha funções básicas dos mais variados tipos como segurança, saúde, educação, dentre outras. Desta forma, há um grande campo para melhorias processuais e orçamentárias.

Analisando os processos e orçamentos governamentais, vemos que há grande brecha para a informatização destes e isto fica ainda mais evidente após a criação e ampla utilização do termo *“eGovernment”* (em tradução livre “governo digital”) que busca formas de informatizar e melhorar os processos governamentais. Para aumentar a produtividade e reduzir custos, órgãos governamentais podem utilizar cada vez mais o auxílio tecnológico como de sistemas.

Considerando esse contexto, a proposta deste trabalho consiste em desenvolver uma API e um *Worker Service* (Serviço de trabalho) visando a melhoria de custos de armazenamento de fontes de dados de órgãos governamentais. Este sistema irá realizar a compressão dos arquivos enviados e irá armazená-los no Amazon S3 sendo que posteriormente será possível realizar a busca por esses arquivos para descompressão e leitura seja qual for a finalidade.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolver uma aplicação web constituída de uma *WebAPI* para receber as requisições de compressão e um *Worker Service* para realizar a compressão e envio dos arquivos comprimidos para a cloud pública da *Amazon Web Services* (AWS).

1.1.2 Objetivos Específicos

Definir a estrutura da aplicação.

Realizar o download de qualquer fonte de arquivos desde que hospedado na web independente do formato.

Realizar a compressão dos arquivos recebidos e armazená-los de forma eficiente e segura.

Disponibilizar os arquivos comprimidos para consulta e descompressão do lado cliente na nuvem pública da AWS.

1.2 Justificativa

As aplicações web estão em constante evolução e aperfeiçoamento. Diante deste cenário, as aplicações de análises gerenciais vêm se tornando cada vez mais recorrentes no dia-a-dia de gestores.

Desta forma, cada vez mais é falado nas áreas de administração e sistemas a respeito de aplicações e análises de *Business Intelligence* (BI) ou Inteligência de Negócios.

Utilizando a definição proposta por Solomon "*Business Intelligence* é um sistema de suporte à decisão orientado a dados, armazenamento de dados e gestão do conhecimento com análise para fornecer subsídios ao processo decisório".

Isso se torna evidente para Hänel e Felden (HÄNEL; FELDEN, 2017) uma vez que os tomadores de decisão utilizam sistemas de tecnologia da informação para enriquecimento e tomada de decisões por meio de *Business Intelligence*, além do cumprimento de tarefas. Essas ferramentas amparam os gestores na tomada de decisões, expondo dados de forma agrupada em gráficos, análises entre outras formas de disposição de grandes massas de dados, classificando-os por tempo, localização, quantidade, entre outros.

O BI pode ser aplicado em praticamente todas as áreas, desde que tenhamos insumos para a análise deste negócio. Isso pode ser exemplificado pela definição de Solomon (2004) que diz que "sistemas de inteligência de negócios devem fornecer informações acionáveis, entregues no tempo, local e forma certas para auxiliar os tomadores de decisão."

Pensando nisso, hoje há um grande movimento referente ao *eGovernment*, isto é, a forma de governar baseada em sistemas, utilizando-se, por exemplo, de análises de dados fornecidas por meio do BI. Como exemplo prático do uso do BI e da aplicabilidade do *eGovernment* há o Painel COVID-19 da Prefeitura Municipal de Joinville - Santa Catarina. Nele é realizado um trabalho de BI para a apresentação dos dados obtidos durante a Pandemia de COVID-19 no município como número de casos, divisão dos casos por bairros, casos separados por sexo, ocupação dos leitos do município entre diversos outros indicadores (Secretaria de Saúde de Joinville, 2021).

Visualizando isso da ótica do Governo Federal do Brasil, atualmente sua grande quantidade de dados fica distribuída em diversos provedores e órgãos governamentais. Um exemplo prático é o de quantitativo de alunos de graduação nas universidades, tanto a Universidade Federal do Rio Grande Do Sul - UFRGS como a Universidade Federal do Paraná - UFPR têm um conjunto de dados no Portal Brasileiro de Dados Abertos[1] sobre este quantitativo, porém estão em conjunto de dados diferentes, organizações diferentes e também não estão comprimidos, sendo extraídos diretamente de arquivos JSON estáticos ou de APIs, tornando a análise desses dados de forma unificada mais complexa e custosa. Assim, há a necessidade de um recurso em que seja possível comprimir os conjuntos de dados para otimização de custos, indiferente da extensão de arquivo utilizada para o conjunto de dados e que seja possível obter o resultado da compressão de forma simplificada.

Nesse contexto, a API receberá as requisições de compressão e irá enviar uma mensagem para uma fila de mensagens do RabbitMQ (Um software de mensagens de código aberto). Além disso, será realizada a leitura destas mensagens pelo *Worker Service*, este irá realizar o download do arquivo que será comprimido e após realizará a compressão e *upload* do resultado da compressão para o AWS S3 que irá armazenar os arquivos comprimidos de forma otimizada e segura. Após *upload* será possível realizar a consulta pela URL para *download* do resultado da compressão via WebAPI e após realizar a descompressão do arquivo gerado no lado do cliente da API, uma vez que o algoritmo de compressão Brotli tem implementação em diversas linguagens de programação.

1.3 Organização do texto

Este texto está organizado em capítulos. O capítulo 2 traz a revisão de literatura, abordando *Business Intelligence*, API, *Worker Service* e Compressão. O Capítulo 3 aborda os materiais e o método utilizado para realização deste trabalho. No Capítulo 4 serão apresentados os resultados obtidos e resultados esperados, além da experiência esperada pelos utilizadores do mesmo. Por fim, são apresentadas as referências bibliográficas utilizadas neste trabalho.

1 https://dados.gov.br/

2 REFERENCIAL TEÓRICO

Durante a pesquisa realizada neste trabalho foi documentada uma definição genérica de Business Intelligence. Na subseção 2.1.1, é abordada a principal forma para aplicação de BI e as principais fontes de dados e na subseção 2.1.2 uma breve justificativa pela abordagem utilizando API. Na subseção 2.2 é apresentada uma definição de API e na subseção 2.2.1 é apresentada a definição de REST. Já na subseção 2.3 é apresentado o conceito de *Worker Service* e sua correspondência com serviços em segundo plano já existentes nos sistemas operacionais Windows e Linux. Por último na subseção 2.4 são apresentados alguns algoritmos de compressão de arquivos e um comparativo entre os mesmos, sendo que na subseção 2.4.1 é justificado o uso do algoritmo Brotli para o problema abordado.

2.1 *Business Intelligence*

Business Intelligence, popularmente conhecido como BI, é responsável por utilizar de forma estratégica os dados em um processo de tratamento e análise com a finalidade de possibilitar as melhores decisões para os negócios.

Hänel e Felden (2017) discutem como os tomadores de decisão utilizam sistemas de TI para enriquecimento e tomada de decisões por meio de *Business Intelligence*, além do cumprimento de tarefas.

O BI pode ser aplicado em praticamente todas as áreas, desde que haja insumos para a análise deste negócio. Um exemplo disso é a definição de Solomon (2004) que diz que “sistemas de inteligência de negócios devem fornecer informações acionáveis...” ou seja, com alta disponibilidade e “...entregues no tempo, local e forma certas para auxiliar os tomadores de decisão.”

Já existe um grande movimento de uso do BI em áreas como indústrias, manufatura, saúde e educação, isso porque estas áreas, em sua maioria, têm uma grande massa de dados e registros provenientes de seus processos e produtos, e por consequência, grandes possibilidades de analisar essa massa de dados para promover melhorias de custos, ganho de mercado, aumento da quantidade vendida e qualidade.

Na área da saúde o BI trouxe um ganho, evidenciado no estudo do impacto do BI na entrega de sistemas de cuidado à saúde nos Estados Unidos da América, onde é feita uma análise da entrega de cuidado à saúde e de como o *Business Intelligence* vem impactando esta área nos EUA (ASHRAFI; KELLEHER, 2014).

Neste estudo, é definido que as informações fornecidas por *Business Intelligence* na área da saúde devem ser obtidas em tempo hábil, devem ter relevância em sua análise e devem ser precisas. Cumprindo estes requisitos, os principais benefícios com a utilização do BI na saúde americana estão baseados em melhores resultados com pacientes, uma vez que há uma análise de casos passados, redução de custos médicos, utilizando decisões mais assertivas e até mesmo a garantia do futuro da indústria de cuidado à saúde.

Em estudo sobre a aplicabilidade do *Business Intelligence* no chão-de-fábrica (FORTULAN; FILHO, 2005) há a definição de que com a grande massa de dados gerada em todas as operações realizadas, as mesmas ficam espalhadas muitas vezes em diversos bancos de dados, o que pode dificultar a análise dos dados. Porém, a partir do momento que esses dados ficam organizados, utilizando como uma das formas de organização a prática de *Data Warehouse* (Banco de dados com a centralização dos dados quantitativos para análise), é possível obter dados referentes ao negócio e realizar estimativas, observar a capacidade de produção, dentre outras métricas (FORTULAN; FILHO, 2005).

Outra área de estudo onde o BI pode ser aplicado é na educação. Conforme estudo realizado por Casartelli (2010) o *Business Intelligence* é aplicado aos negócios, mas não exclusivamente a eles, e pode (é recomendável) que onde há tomada de decisões, seja usado o BI.

Para exemplificar a possibilidade de uso do BI, foi produzido um ensaio de análise dos dados de “Alunos”. No estudo foi utilizado uma base de dados como fonte de dados e o software Pentaho para realizar as análises (MUSSA; DA HORA, 2018). Neste ensaio foi observado que a aplicabilidade do BI na educação traz resultados satisfatórios para iniciar um processo de tomada de decisões orientado por BI, uma vez que, com a estrutura de BI finalizada os tomadores de decisão (como coordenadores) podem consultar dados das mais variadas formas para auxiliar em seu processo de escolha, podendo realizar uma análise assertiva e auxiliando na tomada de decisões.

Desta forma, é perceptível que todas as áreas dadas como exemplificações até o momento demonstraram que se houver dados disponíveis sobre o processo realizado, instituição de estudo ou produto é possível realizar trabalhos de *Business Intelligence*, utilizando uma fonte de dados (Banco de dados ou integração) e a forma de apresentação desses dados (*Dashboard*, gráficos ou aplicações Web). Assim, há uma melhora na compreensão dos processos realizados, nos indicadores referentes ao produto ou serviço.

2.1.2 Principal abordagem para aplicação de BI e fontes de dados

Abordando agora o BI da óptica de fonte dos dados e principal abordagem para aplicação do mesmo é possível analisarmos que os dados podem vir de diversas fontes da própria instituição que passará pelo processo de análise de seus dados.

Isso fica ainda mais evidente no estudo *Data sources for Business Intelligence* (HARIHARAN, 2018), onde é afirmado que a principal abordagem para análises de BI é o *data warehouse* e também são apresentadas as três principais categorias de dados referidos na literatura.

Data warehouse em tradução livre significa armazém de dados e de acordo com o livro Tecnologia e Projeto de *Data Warehouse* (MACHADO, 2006), é a evolução natural do ambiente de apoio à decisão. Além disso, é definido também que *data warehouse* tem por definição ser uma arquitetura que “proporciona uma sólida e concisa integração dos dados da empresa, para realização de análises gerenciais e estratégicas de seus principais processos de negócio.” ademais, tem como característica “integrar e consolidar informações de fontes internas e fontes externas, sumarizando, filtrando e limpando esses dados, preparando-os para análise e suporte a decisão”.

Isso coloca em foco as fontes de dados para aplicar o BI e *data warehouse.* Essas fontes de dados são divididas em três categorias: Dados internos, dados externos e dados pessoais, sendo que cada uma dessas categorias tem origem dos dados diferentes entre si.

Pela definição de Hariharan (2018), dados internos são dados armazenados pela organização geralmente de forma já organizada como dados provenientes de

sistemas de gerenciamento, sistemas de CRM, transações bancárias dentre outros. Dessa forma é possível apresentar os dados internos como dados de origem interna à organização ou dados gerados pela mesma.

Já dados externos são dados provenientes de provedores externos, diversas vezes as organizações contratam externos como ferramentas de marketing ou utilizam de dados de sistemas governamentais externos, estes são categorizados como dados externos, uma vez que não estão armazenados internamente pela organização.

Já dados pessoais são dados gerados geralmente pelos tomadores de decisão, que em suas análises prévias já conseguem obter informações relevantes para análises futuras.

O Quadro 1 apresenta as origens de dados de cada uma das categorias citadas anteriormente.

Quadro 1 - Fontes de dados e suas origens

Fonte de dados	Origem dos dados
Dados internos	Dados de transações bancárias e de pontos de venda, dados de sistemas de relacionamentos com os clientes (CRM), dados provenientes de bancos de dados internos, registros internos (Provenientes de formulários, documentos Word, PDFs, XMLs etc.), outras aplicações empresariais e dispositivos sensorizados (Usando *gadgets* com sensores e conectividade).
Dados externos	Mídias sociais, dados do governo, Google entre outras fontes de dados externas à organização.
Dados pessoais	Pastas de trabalho e bancos de dados locais nos computadores dos tomadores de decisão.

Fonte: Autoria própria (2022).

2.2 Definição de API

API (*Application Programming Interface*) é um tipo de sistema que visa realizar a integração com outro sistema com base em uma forma de comunicação comum. De acordo com o MDN Web Docs (2021) “No desenvolvimento Web, uma API é um conjunto de métodos padronizados, propriedades, eventos e URLs [...]” estes que podem ser utilizados para “[...] interagir com componentes do navegador da Web de um usuário ou outro software / hardware no computador do usuário ou sites e serviços de terceiros.”

Desta forma é possível utilizar APIs para realizar a comunicação entre sistemas. Essa comunicação utiliza uma padronização de requisições para disponibilizar ou receber dados e tarefas a serem realizadas.

Assim, uma API se resume a um sistema que disponibiliza formas de executar determinadas tarefas por meio de requisições padronizadas (geralmente requisições utilizando a estrutura JSON e o protocolo HTTP).

Outra característica definida para uma API fica evidenciada pela Red Hat (2017). Além de estabelecer a comunicação entre sistemas e serviços, não deve haver a necessidade de saber como aquela API ou serviço foi implementado, do ponto de vista do desenvolvedor.

Desta forma, de maneira breve podemos caracterizar uma API como um contrato, que contém documentações representando o acordo entre as partes interessadas. Ou seja, “Se uma dessas partes enviar uma solicitação remota estruturada de uma forma específica, isso determinará como o software da outra parte responderá.” Red Hat (2017).

Ademais, há alguns estilos arquiteturais que definem o conjunto de restrições para criação de serviços web, sendo dois dos mais conhecidos o SOAP e o REST.

2.2.1 Definição de REST

A sigla REST representa *Representational State Transfer*, em tradução livre significa Estado Representacional e é um estilo de arquitetura proposto por *Roy Fielding* (2000) para servir aplicações na *Web*.

Este padrão oferece um conjunto de regras e convenções para desenvolver um *Web service* (Serviço para web) com escalabilidade, padronizado, que não dependa de outros padrões arquiteturais ou determinadas tecnologias (como linguagens de programação ou ferramentas específicas).

Para apresentação do padrão arquitetural são apresentados os seguintes princípios (FILEDING, 2000):

- Cliente-Servidor;
- *Stateless;*
- *Cache*;
- Interface uniformizada;
- Sistema em camadas;
- Código sob-demanda.

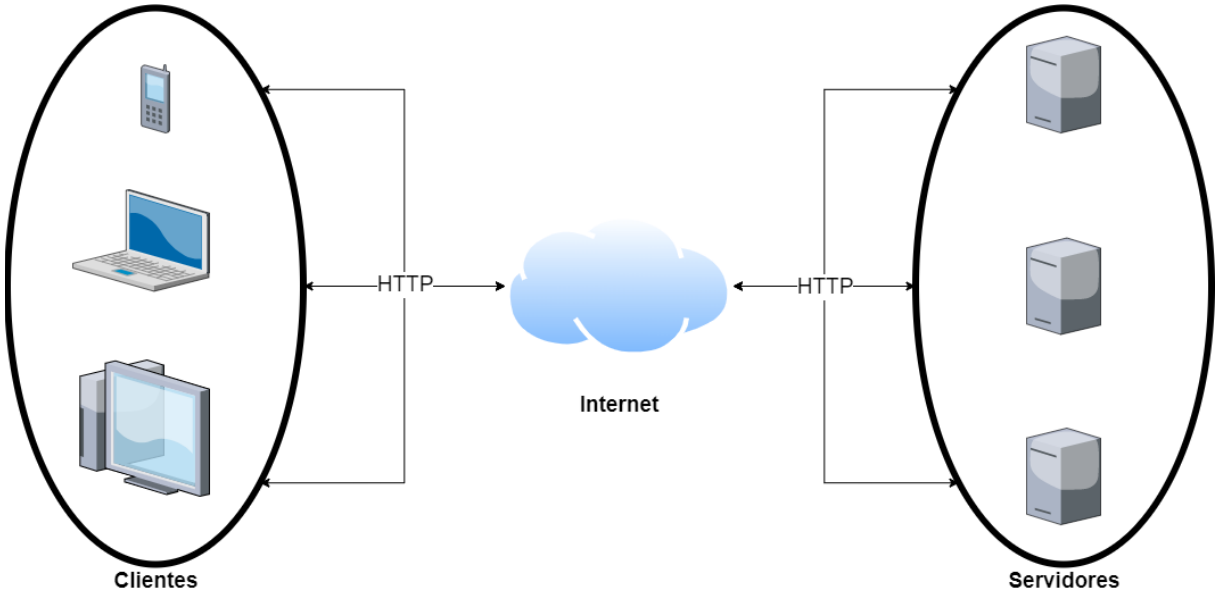
Cliente-Servidor:

Para o princípio de cliente-servidor (FIELDING, 2000), é proposto que as responsabilidades devem ser segregadas. Assim sendo, a interface do usuário deve ser separada (pode ser uma página na web ou um aplicativo para dispositivos móveis por exemplo) das responsabilidades de armazenamento dos dados e outras tarefas, que devem ficar a cargo do servidor.

Aplicando o princípio de cliente-servidor é possível melhorar a arquitetura em diversos aspectos como a portabilidade do sistema para outras plataformas, sua escalabilidade simplificando os componentes do servidor e também permitir que cada parte da aplicação (cliente e servidor) evoluam individualmente.

Na Figura 1 há um diagrama exemplificando como funciona a comunicação cliente-servidor na *web*.

Figura 1 - Diagrama de padrão de comunicação cliente-servidor



Fonte: Autoria própria (2022).

É possível notar na Figura 1 que há diferentes dispositivos, como computadores e dispositivos móveis se comunicando com os servidores. Além disso, na figura é utilizado o protocolo HTTP para comunicação.

Outro ponto importante de ressaltar é que no lado dos servidores é possível que haja mais de um servidor para atender as demandas dos clientes, como um servidor exclusivo para banco de dados, um apenas para armazenamentos de arquivos etc.

Stateless:

Incrementando o princípio de cliente-servidor há o princípio *Stateless* que em tradução livre significa sem estado.

Neste princípio (FIELDING, 2000) é definido que as requisições realizadas utilizando o padrão REST não devem conter estado, ou seja, a requisição deve conter todos os dados necessários para ser compreendida pelo servidor. Além disso, a requisição não deve ter qualquer vantagem de dados salvos previamente no servidor.

Esse princípio acaba trazendo melhorias em alguns aspectos de serviços web conforme apresentado por *Roy*, são eles:

- Visibilidade;
- Confiabilidade;
- Escalabilidade.

Do ponto de vista de visibilidade o princípio *Stateless* apresenta melhorias pois ao monitorar um serviço web você não precisa analisar uma requisição em específico, já que as requisições não tem qualquer estado armazenado no lado do servidor.

Já do ponto de vista de confiabilidade há melhorias uma vez que para se recuperar de falhas, não dependendo do estado armazenado no servidor, basta iniciar a rotina de recuperação e a operação estará sendo executada.

E do ponto de vista de escalabilidade existem ganhos também, uma vez que não tendo que armazenar o estado do cliente no lado do servidor, assim como não tendo que manter a conexão e o fluxo de dados ativos, o mesmo pode liberar recursos de forma muito mais rápida.

Além disso, torna-se muito mais simples a tarefa de implementação já que o servidor também não precisa gerenciar recursos entre as requisições.

Cache:

Em detrimento da melhoria de eficiência de rede, *Roy* adicionou o princípio de *Cache*.

Cache nada mais é que um espaço de armazenamento de acesso rápido para consulta de ações já realizadas anteriormente, como acessar uma página na web, onde partes da página que mudam com menor frequência como a logo do site por exemplo, podem ser armazenadas no cache para consultas futuras.

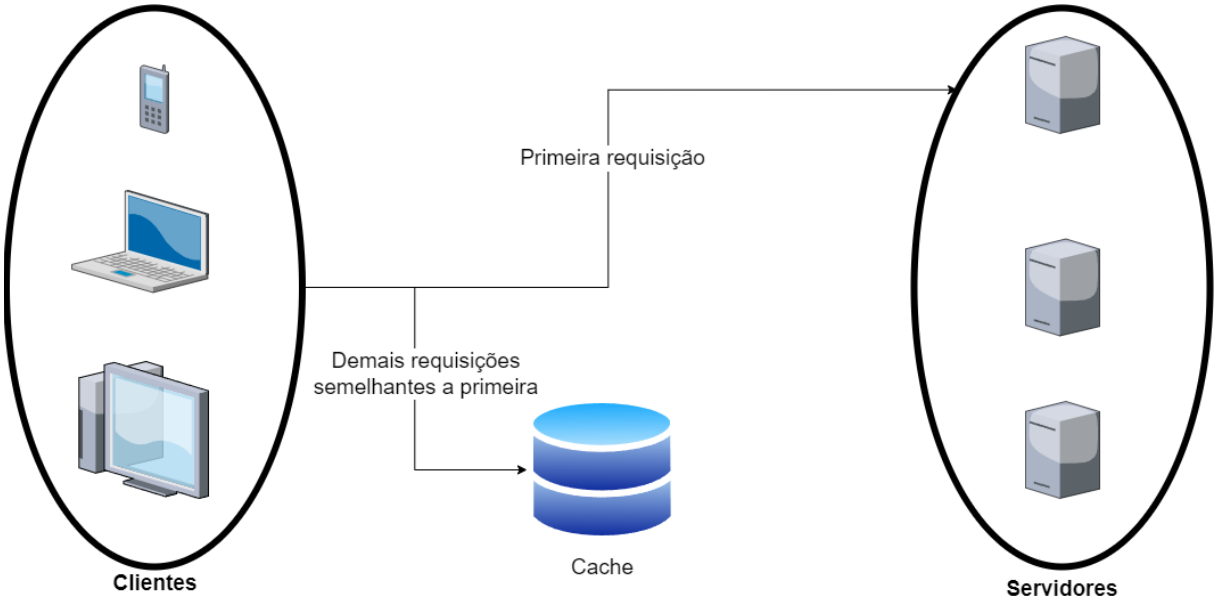
Sendo assim, este princípio tem como objetivo o reuso de respostas enviadas pelo servidor ao cliente caso o servidor rotule a resposta como podendo ser armazenada em *cache*.

Se essa resposta estiver rotulada dessa forma o cliente poderá armazenar o resultado da requisição em seu *cache* e consultar do mesmo caso haja uma requisição semelhante.

Na Figura 2 há um diagrama que exemplifica de forma simplificada o fluxo realizado quando há o *cache* das requisições em operações de cliente servidor.

Na primeira requisição clientes fazem diretamente aos servidores e após a primeira requisição caso seja realizada uma requisição semelhante a primeira será consultado o resultado do cache.

Figura 2 - Diagrama de requisição utilizando *cache* entre clientes e servidores



Fonte: Autoria própria (2022).

Interface Uniformizada:

Fielding definiu também características que determinam regras bem definidas para que a comunicação fique mais uniforme e normalizada (FIELDING, 2000).

Abaixo são apresentadas as regras definidas como padrão de arquitetura:

- Cada recurso é identificado por meio de um URI específico, único e coeso, geralmente estes URIs são identificados pela nomenclatura *endpoint* (em português: ponto final), que são os endereços para acessar determinado recurso;
- Uma URI representa recursos e não ações. De acordo com o padrão proposto a URI deve representar o recurso que o cliente deseja acessar, por exemplo: Acessar os alunos em um sistema acadêmico, nesse caso os alunos são o recurso e acessar é a ação;
- As ações são representadas pelos verbos HTTP (Os mais comuns sendo POST, GET, DELETE, PATCH, OPTIONS e PUT). Aqui são especificadas as ações a serem realizadas em um determinado recurso. Sendo assim,

utilizando-se do exemplo anterior a ação “Acessar” seria substituída por um verbo HTTP, nesse caso o verbo GET (Obter);

- O formato do recurso solicitado que será devolvido ao cliente é escolhido pelo servidor que irá devolver a requisição sendo os mais comuns o JSON e o XML;
- O formato de comunicação cliente-servidor deve ser definido no *Content Type*, este é um cabeçalho que deve ser enviado juntamente com a requisição HTTP;
- O cliente deverá receber todas as informações necessárias na resposta vinda do servidor e a partir desta resposta deve conseguir utilizar os outros recursos do serviço.

Sistema em camadas:

A aplicação deve ser desenvolvida em camadas e essas camadas devem ser fáceis de manter e prestar manutenção, serem alteradas e adicionar ou remover recursos. Cada camada processa as informações entre o cliente e o servidor da sua maneira.

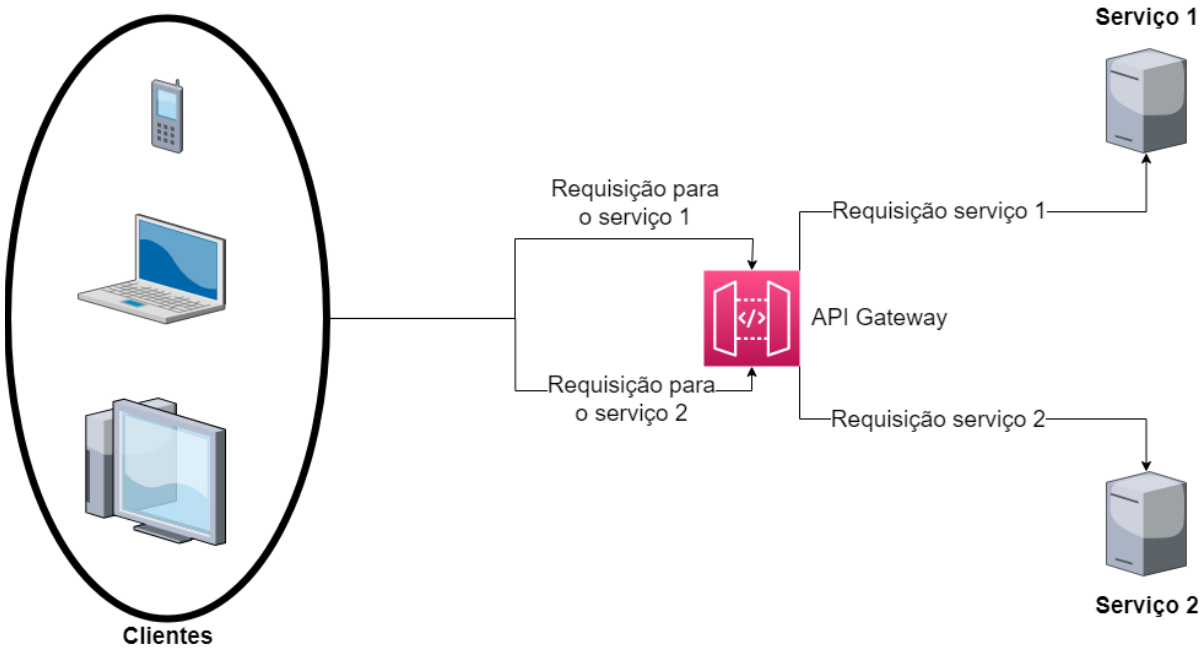
É muito comum encontrar uma camada intermediária entre os serviços web e o cliente. No mundo da programação moderna e serviços em servidores na nuvem são muito utilizados os *API Gateways* que são uma camada intermediária entre o cliente e o servidor.

Os *API Gateways* são serviços *web* intermediários e servem para fazer a distribuição das chamadas HTTP entre os demais serviços. Isso é de extrema importância, pois dessa forma os serviços web não ficam diretamente expostos na rede sendo que, do ponto de vista da segurança isso é excelente pois previne ataques dos mais variados tipos.

Além disso, os *API Gateways* acabam desempenhando o papel de distribuidores das requisições, em um cenário de arquitetura distribuída, por exemplo. Onde há mais de um serviço *web* para realizar as tarefas, é papel do *API Gateway* enviar cada requisição para seu respectivo serviço.

Na Figura 3, há um exemplo aplicando o *API Gateway*. Os clientes (geralmente sistemas web) realizam requisições ao *API Gateway* e o mesmo encaminha a requisição para o serviço solicitado.

Figura 3 - Diagrama de requisição a um *API Gateway* e requisições sendo redirecionadas pelo mesmo



Fonte: Autoria própria (2022).

Código sob-demanda:

Esse é o único princípio opcional da arquitetura REST pois reduz a visibilidade. Isso ocorre pois nesse princípio (FIELDING, 2000) define que a aplicação pode ser estendida baixando ou executando código na forma de scripts, pacotes ou *applets* (que é um pequeno *software* executando em um *software* maior).

Uma vantagem utilizando código sob-demanda é que a extensibilidade e agilidade para desenvolvimento e implantação do *software* aumenta significativamente.

Por outro lado, caso haja um código com erros em alguma dessas extensões também irá prejudicar o sistema que o utiliza e consequentemente isso prejudica a visibilidade.

2.3 *Worker Service*

Worker Service é uma abstração de serviços de longa duração do *framework* de desenvolvimento .NET da Microsoft.

Em uma definição presente no site de documentação da Microsoft (2022) é possível constatar que a criação de um serviço a ser executado junto ao sistema operacional é benéfica em casos de dados processados intensamente na CPU ou realizando a execução de tarefas em formato de fila em segundo plano no sistema operacional.

Ou seja, o serviço ficará realizando as operações enviadas a ele sem qualquer intervenção humana, ou até mesmo operações em que há um agendamento para a tarefa ser realizada.

Desta forma, a nível de sistema operacional e focando nos dois principais ecossistemas nesse assunto, Windows e Linux, há dois principais tipos de aplicações que se encaixam nesse critério.

No ecossistema Windows há o Windows Service, que é um serviço executado em segundo plano no sistema operacional executando a lógica definida em seu código.

No ecossistema Linux há a definição de Linux Daemons, que contém a mesma finalidade que o Windows Service, a execução de tarefas em segundo plano.

O principal motivo para utilizar um *Worker Service* e não a definição direta de Windows Service ou Linux Daemon é orientado ao desacoplamento de sistema operacional. Pensando em grandes aplicações de grandes corporações quanto maior o nível de desacoplamento de uma determinada ferramenta, melhor. Desta forma, caso haja a necessidade de trocar de sistema operacional basta hospedar o serviço no novo servidor.

Uma das formas de informar tarefas a serem realizadas a um *Worker Service* é por meio do uso de *brokers* de mensageria, no caso do trabalho realizado será utilizado o *broker* RabbitMQ. O RabbitMQ trabalha com estruturas de fila onde as mensagens são agrupadas por ordem de inserção e as primeiras mensagens inseridas são processadas primeiro.

No RabbitMQ de acordo com sua documentação (2021) são definidos os conceitos de produtor e consumidor. Produtor é alguém que envia mensagens a serem processadas para a fila e consumidor quem irá receber essas mensagens. Essa abordagem se mostra interessante uma vez que, toda vez que uma nova mensagem for inserida por um produtor na fila todos os consumidores inscritos nesta fila são notificados e recebem a mensagem para realizar suas tarefas.

2.4 Compressão

Compressão é a arte de comprimir os dados ou em outras palavras, um arquivo quando comprimido pode ser reduzido (KODITUWAKKU; AMARASINGHE, 2020). Olhando do ponto de vista da tecnologia da informação técnicas de compressão são amplamente utilizadas para melhor aproveitamento de recursos e estão amplamente aplicadas em aplicações famosas como o site de *streaming* de vídeo YouTube, que comprime os vídeos enviados para a plataforma visando otimizar o armazenamento.

Com essa prática é possível atacar um dos principais entraves para escalonar um produto na web, armazenamento. Utilizando o caso do YouTube, são enviadas por dia cerca 576 mil horas de vídeo, 400 horas por minuto, utilizando dados de agosto de 2019. Considerando vídeos na qualidade 1080p a 60 fps há uma média de 176MB por minuto enviado (GUEST, 2019). Considerando o cenário sem compressão em um dia seria um total de 6082560000MB a ser armazenado em um único dia, ou cerca de 6 milhões de *gigabytes*.

Considerando a quantidade de dados transacionado pelo YouTube seria inviável a operação sem utilizar nenhuma técnica de compressão dos dados. No sentido de compressão de dados, existem algumas alternativas como o RLE, LZW e Brotli, que serão detalhados em seguida.

RLE:

A ideia básica por trás do algoritmo de compressão RLE se baseia na substituição da repetição de determinados itens repetidos no fluxo de dados de entrada, ou seja, se um item *d* ocorre *n* vezes é realizada a substituição de toda a sequência de repetição pelo número de vezes da repetição e o item repetido.

Um exemplo simples é a frase utilizada por David Salomon (2004):

“all is too well.”

Aplicando a técnica RLE para compressão o resultado seria:

“a2l is t2o we2l.”

Porém, aplicando essa técnica não há como saber se os algarismos “2” estavam presentes no texto original ou não, portanto é necessário um caractere delimitador. Utilizando o @ como caractere delimitador teríamos o seguinte resultado:

“a@2l is t@2o we@2l.”

Porém, a compressão RLE implica o risco de aumentar o tamanho original do texto e consequentemente o tamanho em espaço de armazenamento ocupado.

Há um protocolo chamado MNP5, implementado originalmente por fabricantes de modems (SOLOMON, 2004) que utiliza o RLE sendo aplicado de uma forma diferente. Neste protocolo toda vez que é encontrado uma repetição de 3 ou mais caracteres o caractere repetido é escrito três vezes seguido pelo número de repetições.

Por exemplo uma sequência de três caracteres da letra “b” seria escrito da seguinte forma no protocolo MNP5:

“bbb0”

Já uma sequência de 4 caracteres da letra “a” seria escrita da seguinte forma:

“aaa1”.

Esse protocolo minimiza o risco de expansão do resultado original durante o processo de compressão, mas não o anula.

LZW:

O algoritmo LZW se caracteriza por ser um algoritmo de compressão da classe dos que utilizam dicionários como ferramenta para aplicar a compressão. Isso significa que é criada uma estrutura que armazena os padrões no objeto a ser comprimido e um identificador para esse padrão.

Tratando de compressão de texto utilizando o algoritmo LZW, primeiramente é criado um dicionário com todos os 256 caracteres da tabela ASCII. Após isso, uma iteração é executada no texto informado, passando caractere por caractere e verificando se esse caractere ou sequência de caracteres existe no dicionário.

Por exemplo, vamos considerar que **AT** é o caractere atual e **AN** o caractere anterior, utilizando o texto **“wabbawabba”** serão aplicados os seguintes passos:

Na primeira iteração como já existem todas as letras no dicionário o algoritmo irá encontrar a mesma, porém será considerado um cenário reduzido apenas com as letras que aparecem no texto informado.

A tabela então está inicializada da seguinte forma:

Quadro 2 - Dicionário no início da compressão LZW.

Chave	Dicionário
1	a
2	b
3	w

Fonte: Autoria própria (2022).

Sendo assim, serão atribuídos os seguintes valores para **AT** e **AN**:

AN = nulo (Pois não há um caractere anterior, já que é o início da compressão)

AT = w (Já que o primeiro caractere é o w).

Logo após, o valor de **AN** receberá **AN + AT**, ou seja, receberá **w** e é verificado se já existe no dicionário a junção de **AN + AT**.

Na segunda iteração os valores estarão dispostos da seguinte forma:

AN = w

AT = a

AN + AT = wa, como **wa** não existe no dicionário o mesmo será inserido, ficando da seguinte forma:

Quadro 3 - Dicionário na segunda iteração utilizando o algoritmo LZW

Chave	Dicionário
1	a
2	b
3	w
4	wa

Fonte: Autoria própria (2022).

Nos pontos em que são adicionadas sequências de caracteres na tabela teremos as saídas da compressão, com base nas chaves do dicionário. Quando há a ocorrência de inserção no dicionário a chave de **AN** será atribuída para a saída. Ou seja, a saída terá como primeiro algarismo o número 3 do **w** em **AN**.

Seguindo essa sequência até o final do texto informado a tabela ficará da seguinte forma:

Quadro 4 - Dicionário ao término da compressão utilizando LZW

Chave	Dicionário
1	a
2	b
3	w
4	wa
5	ab
6	bb
7	ba
8	aw
9	wab
10	bba

Fonte: Autoria própria (2022).

E o resultado final da compressão será: 31221461 (LZW Coding 1, 2014).

Brotli:

Brotli é um algoritmo desenvolvido pelo Google, foi desenvolvido inicialmente para o uso em servidores *web* para comprimir o conteúdo HTTP, visando maior velocidade no carregamento na *web* (SZABADKA, 2015).

O Brotli é caracterizado como um algoritmo do tipo que utiliza um dicionário, em sua concepção inicial ele utiliza como base a combinação do algoritmo LZ77 e *Huffman coding*.

2.4.1 Discussão acerca do algoritmo de compressão

Comparando o algoritmo Brotli com outro algoritmo muito utilizado para comprimir as respostas HTTP na web, o Gzip, o Brotli tem uma vantagem de cerca de 15% em redução de tamanho em arquivos JavaScript, cerca de 20% em redução de tamanho de arquivos HTML e para arquivos CSS a redução é de cerca de 16% (PANDJAROV, 2021).

Atualmente o algoritmo Brotli já é utilizado de forma difundida para compressão. No repositório de código aberto mantido pelo Google o algoritmo já tem mais de 1000 alterações, contando com portes para diversas linguagens de programação como C, Go, C#, Java, dentre outras (Google, 2022).

3 MATERIAIS E MÉTODO

A seguir estão o método e os materiais utilizados para a implementação do sistema desenvolvido como resultado que será obtido como resultado da realização deste trabalho.

3.1 Materiais

O Quadro 5 apresenta a lista de ferramentas e tecnologias utilizadas para o desenvolvimento deste trabalho.

Quadro 5 - Lista de ferramentas e tecnologias

Ferramenta/Tecnologia	Versão	Finalidade
Visual Studio Code	1.58.2	Editor de texto.
Visual Studio	2022	IDE de desenvolvimento.
PostgreSQL	14.0	Banco de dados SQL, utilizado para armazenar de forma mais fácil os relacionamentos.
Docker	20.10.8	Virtualizador, utilizado para criação de pacotes chamados *containers*.
DBEaver	21.2	Cliente SQL.
C#	10.0	Linguagem de programação
RabbitMQ	3.9	*Broker* de mensageria.
.NET	6.0	Framework de desenvolvimento.

Fonte: Autoria própria (2022).

3.2 Método

A seguir são apresentados os capítulos de método utilizado no desenvolvimento deste trabalho.

3.2.1 Elaboração da estrutura do projeto

É necessário estruturar o projeto a nível de código e aplicações necessárias para maior performance, melhor experiência do usuário e maior qualidade de manutenção. Desta forma, pensando em performance, há a necessidade de realizar uma abstração do processo de compressão, que muitas vezes pode demorar dependendo do tamanho do arquivo que está sendo comprimido, não deixando apenas a aplicação da API encarregada de realizar todas as operações.

Consequentemente, há a necessidade de utilizar o agendamento da tarefa de compressão, que será realizada de forma assíncrona, permitindo que a API receba requisições e agende e enfilere as ordens de compressão.

Utilizando de um tipo de estrutura de código do framework .NET chamado *Worker Service*, podemos desacoplar a lógica de processamento do mesmo, fazendo com que haja um ganho de performance na API.

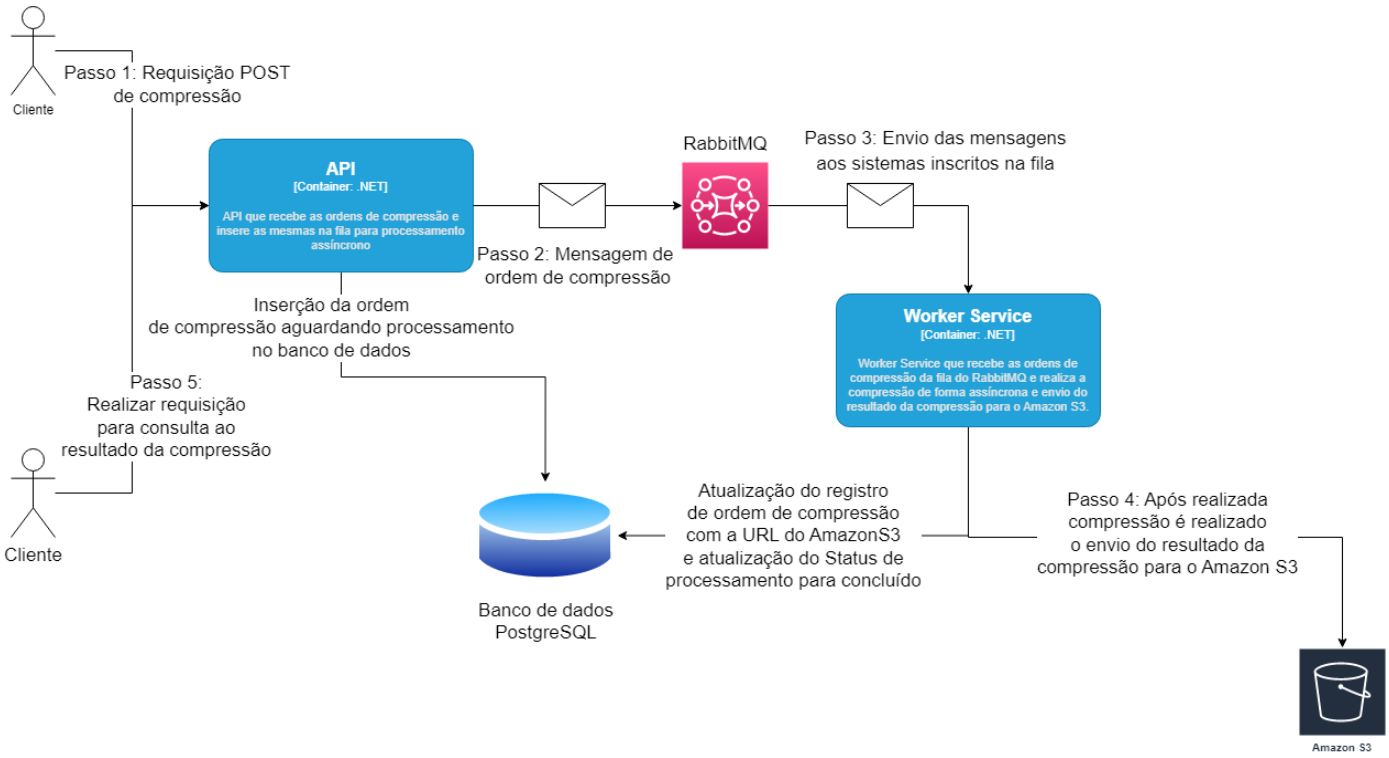
A comunicação entre a API e este serviço é realizada através de mensageria, utilizando o *broker* de mensagens RabbitMQ, onde é possível “inscrever” a API como produtor de mensagens para uma fila de mensagens e o serviço como consumidor desta fila. A cada nova mensagem todos os consumidores são notificados para realizarem suas tarefas.

Com base nesta fila de mensagens é possível realizar o processamento assíncrono das requisições enviadas para a API, já que a API tem a responsabilidade de apenas inserir as mensagens na fila do RabbitMQ.

Já o *Worker Service*, fica com a operação de receber as mensagens da fila e processar as ordens de compressão. Cada ordem acarretará na compressão dos arquivos provenientes da requisição realizada à API.

A Figura 4 apresenta um fluxograma resumido da aplicação.

Figura 4 - Fluxograma da aplicação



Fonte: Autoria própria (2022).

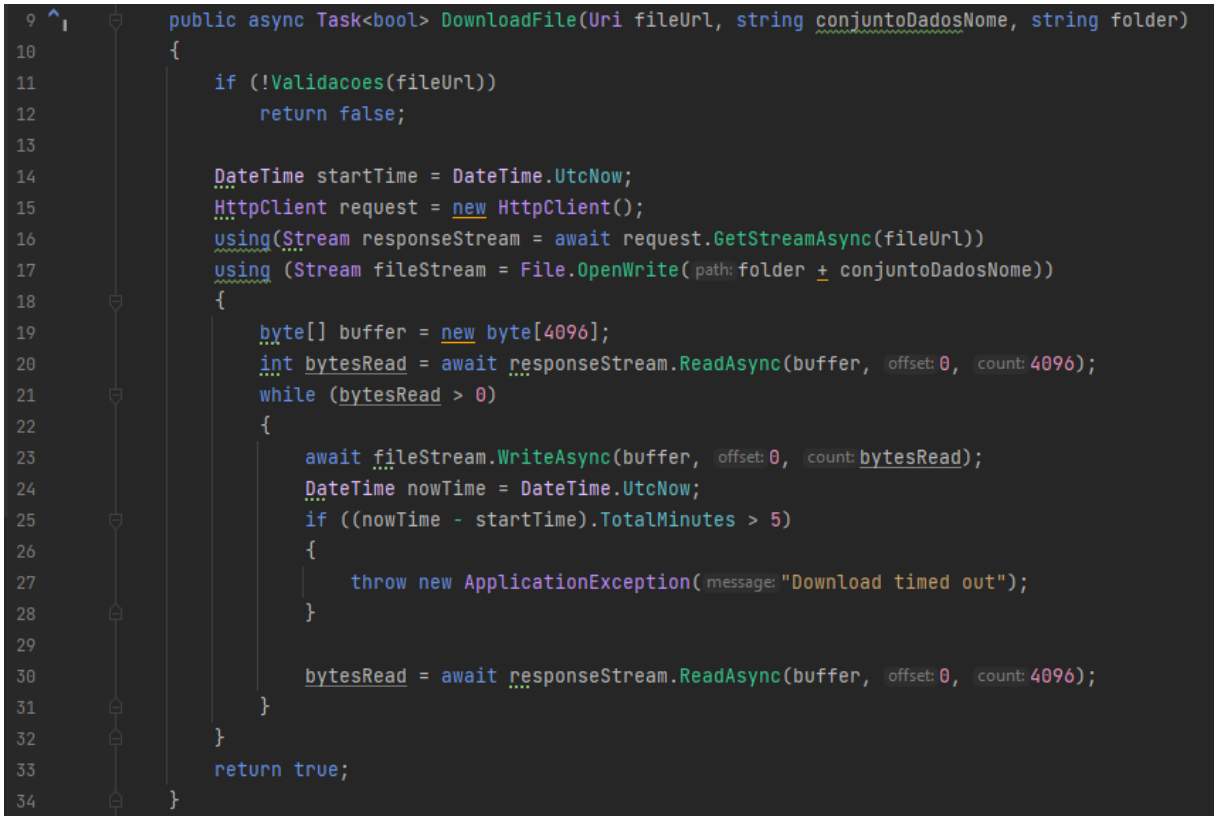
O primeiro passo no canto superior esquerdo é a requisição de compressão feita pelo cliente, nesta requisição deverá ser informado o ID no banco de dados da organização detentora daquele conjunto de dados e a URL na qual o conjunto está localizado.

No segundo passo, no centro da imagem, a API publicará uma mensagem na fila de mensagens do RabbitMQ e este consequentemente irá notificar todos os sistemas inscritos nesta fila, conforme passo três, em que a mensagem é encaminhada para o *Worker Service*.

No canto inferior direito ocorre o passo quatro que irá iniciar o processamento da mensagem recebida, acessando a fonte de dados, realizando o download, realizando a sua compressão e fazendo o upload ao AWS S3.

Para realizar o upload para o AWS S3 a mensagem recebida contém o link do arquivo que deverá ser comprimido, a partir disso é executada uma função que realiza o download do arquivo para o servidor. O código da função de download na linguagem de programação C# pode ser visualizado na Figura 5.

Figura 5 - Código C# de download de arquivos externos para o servidor



```
9   public async Task<bool> DownloadFile(Uri fileUrl, string conjuntoDadosNome, string folder)
10  {
11      if (!Validacoes(fileUrl))
12          return false;
13
14      DateTime startTime = DateTime.UtcNow;
15      HttpClient request = new HttpClient();
16      using(Stream responseStream = await request.GetStreamAsync(fileUrl))
17      using (Stream fileStream = File.OpenWrite( path: folder + conjuntoDadosNome))
18      {
19          byte[] buffer = new byte[4096];
20          int bytesRead = await responseStream.ReadAsync(buffer, offset: 0, count: 4096);
21          while (bytesRead > 0)
22          {
23              await fileStream.WriteAsync(buffer, offset: 0, count: bytesRead);
24              DateTime nowTime = DateTime.UtcNow;
25              if ((nowTime - startTime).TotalMinutes > 5)
26              {
27                  throw new ApplicationException( message: "Download timed out");
28              }
29
30              bytesRead = await responseStream.ReadAsync(buffer, offset: 0, count: 4096);
31          }
32      }
33      return true;
34  }
```

Fonte: Autoria própria (2022).

Na linha 11 do código apresentado na Figura 5 é executada uma função que valida se a entrada de dados é válida, caso não seja não executa a função de download.

Na linha 15 é criado uma variável do tipo *HttpClient*. Na linguagem C# essa classe tem a responsabilidade de realizar operações HTTP na *web*, neste cenário essa classe será responsável por fazer o *download* do arquivo contendo o conjunto de dados.

Nas linhas 16 e 17 são criados fluxos de dados para realizar a leitura do arquivo solicitado e para realizar a escrita desse arquivo no servidor, respectivamente.

Na linha 19 é criado uma variável com tamanho de 4096 *bytes* para realizar a leitura e escrita em pacotes, sendo que esses pacotes terão tamanho máximo de 4096 *bytes*. Essa prática foi adotada pois os arquivos que serão solicitados podem apresentar tamanho consideravelmente grande.

Na linha 21 é realizada a execução de um laço de repetição para que enquanto houver arquivos a serem lidos o código de leitura e escrita (Linhas 23 e 30 respectivamente) sejam executados.

Por fim, cada pacote de dados lido é escrito no arquivo que é salvo no servidor.

Após realizar o *download* do arquivo para o servidor é executada outra função, essa por sua vez irá realizar a compressão do arquivo. O código da função de compressão está disposto na Figura 6.

Figura 6 - Código C# de compressão de arquivos baseados em texto

```
14  public async Task<bool> CompressTextFile(string fileName, string folder, Guid processamentoId)
15  {
16      try
17      {
18          await using var stream = File.OpenRead( path: folder + fileName);
19          await using var fileStream = File.Create( path: folder + processamentoId + fileName);
20          await using var brotliStream = new BrotliStream(fileStream, CompressionLevel.SmallestSize);
21          await stream.CopyToAsync(brotliStream);
22
23          return true;
24      }
25      catch (Exception e)
26      {
27          _logger.LogError(e, e.Message);
28          return false;
29      }
30  }
31
```

Fonte: Autoria própria (2022).

Na função da Figura 6 primeiramente é realizada a leitura do arquivo na pasta informada e com o nome informado e criado um fluxo de dados de leitura desse arquivo na linha 18.

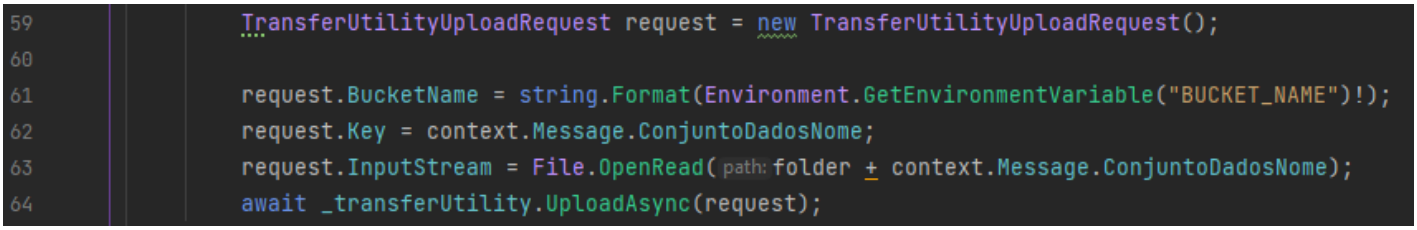
Após é criado um fluxo de dados para criação do arquivo comprimido na linha 19.

Logo abaixo é criado um fluxo de compressão, onde é utilizada a classe *BrotliStream* para compressão na linha 20.

E após os passos anteriores é realizada a cópia dos dados comprimidos para o arquivo que está sendo criado no servidor na linha 21.

Depois de o arquivo original já estar comprimido é realizado o envio para o AWS S3, utilizando as funções disponibilizadas pela própria AWS para envio, conforme apresentado na Figura 7.

Figura 7 - Código C# para envio do arquivo comprimido para o AWS S3



```
59        TransferUtilityUploadRequest request = new TransferUtilityUploadRequest();
60
61        request.BucketName = string.Format(Environment.GetEnvironmentVariable("BUCKET_NAME")!);
62        request.Key = context.Message.ConjuntoDadosNome;
63        request.InputStream = File.OpenRead( path: folder + context.Message.ConjuntoDadosNome);
64        await _transferUtility.UploadAsync(request);
```

Fonte: Autoria própria (2022).

Na linha 59 é criada uma variável do tipo *TransferUtilityUploadRequest* que é disponibilizada pela biblioteca de acesso ao AWS S3 criada pela própria AWS.

Nessa variável são atribuídos o nome do bucket (linha 61), a chave do arquivo que será enviado para o S3, que é um identificador único daquele arquivo (linha 62) e o fluxo de dados para acesso ao arquivo que será enviado (linha 63). Após, na linha 64 é executada a função do objeto *TransferUtility,* também da biblioteca da AWS, que irá realizar o envio do arquivo para o S3.

3.2.2 Experiência Esperada

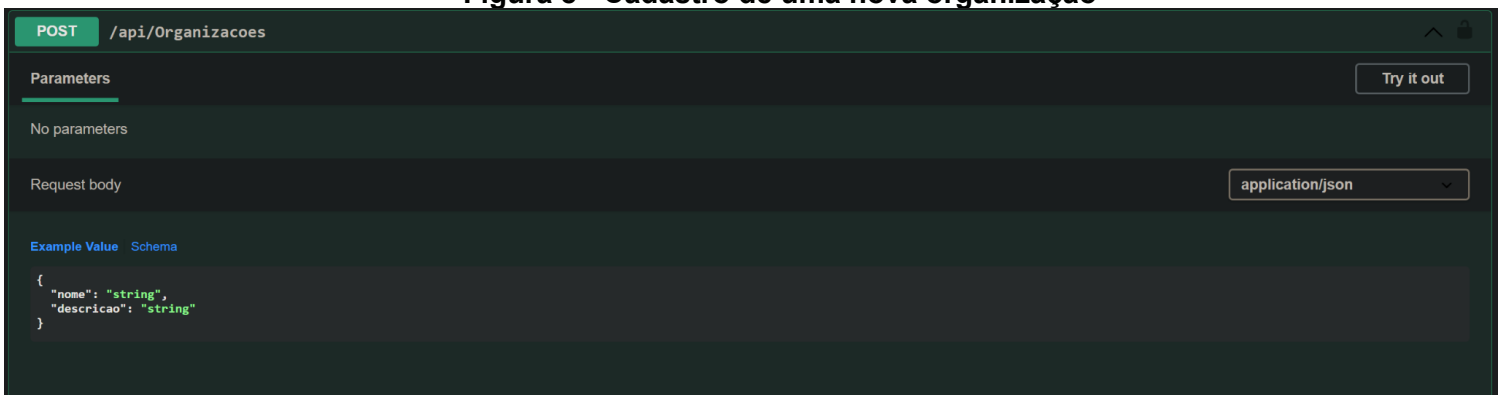
A experiência esperada se refere a de que forma os utilizadores deste projeto irão se relacionar e quais as operações possíveis a serem realizadas dentro da aplicação.

Desta forma as operações ficam divididas em duas principais *controllers*: Organização e Data.

A primeira *controller* chamada Organização se refere às operações realizadas com as entidades de Organizações governamentais, sendo 4 as operações possíveis nessa controller, visando a possibilidade de executar os principais comportamentos no banco de dados como criar o registro de organização, buscar por ID, buscar todos, atualizar e excluir.

A primeira operação dessa controller é feita utilizando o verbo HTTP POST onde deverá ser informado o nome da organização governamental e uma descrição (que é opcional) por meio da estrutura JSON para armazenamento no banco de dados. Na Figura 8 é apresentada a documentação dessa rota gerada pelo *Swagger*.

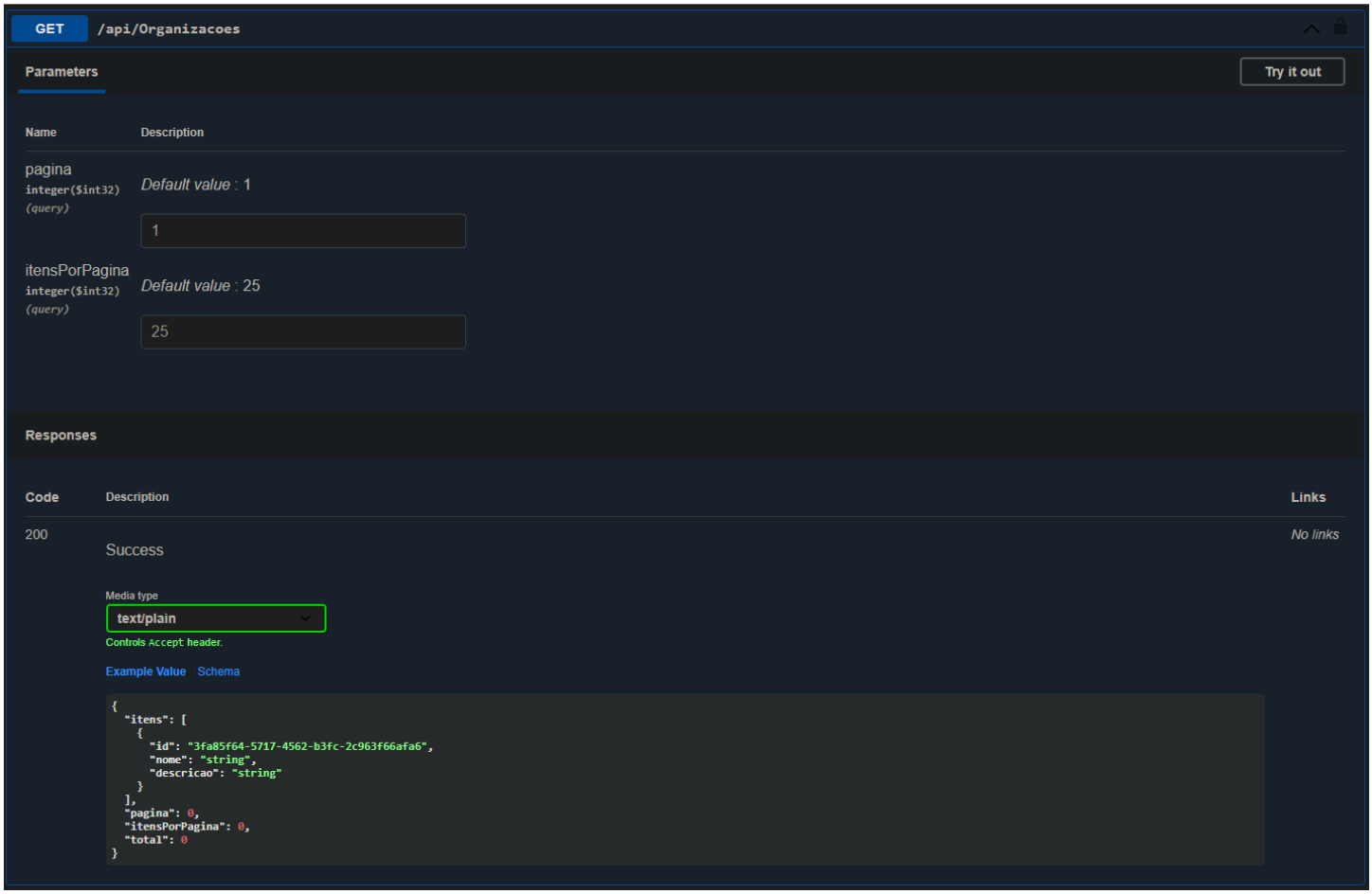
Figura 8 - Cadastro de uma nova organização



Fonte: Autoria Própria (2022).

Após, são executadas duas operações com o verbo HTTP GET. A primeira que obtém toda a listagem de organizações cadastradas com seus nomes e IDs apresentada na Figura 9, essa operação tem uma particularidade, por questões de performance é realizada uma busca paginada, passando a página que deve ser consultada e a quantidade de itens por página, caso não sejam informados estes parâmetros é atribuído por padrão a página 1 e 25 itens por página, conforme apresentado na Figura 9.

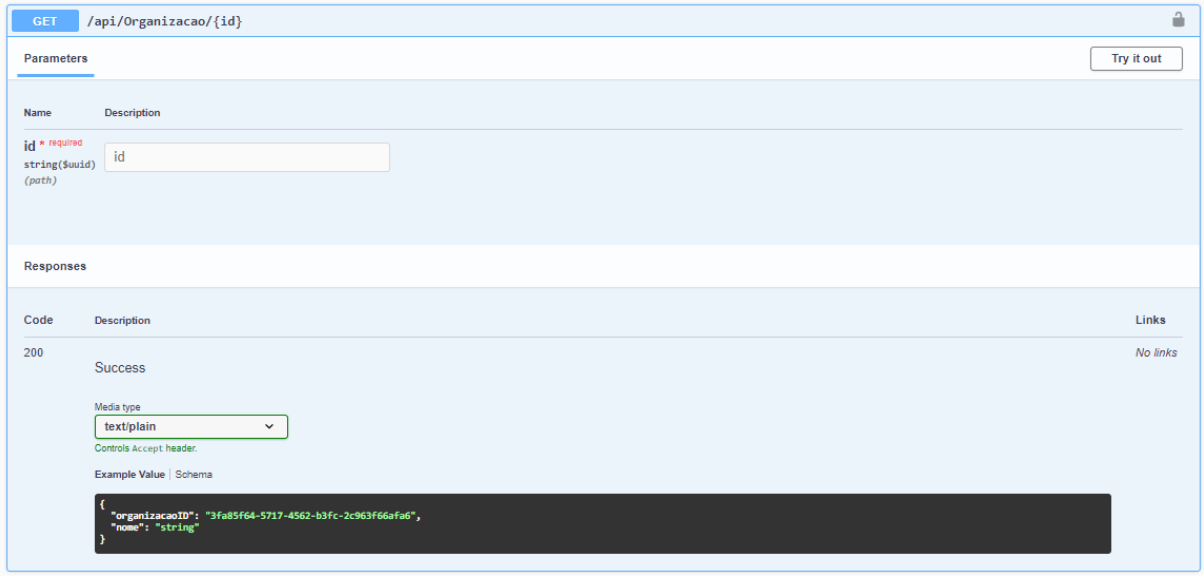
Figura 9 - Obtém todas as organizações



Fonte: Autoria própria (2022).

A segunda operação utilizando o verbo HTTP GET é a que obtém os dados de uma organização usando seu ID como parâmetro de busca, apresentada na Figura 10.

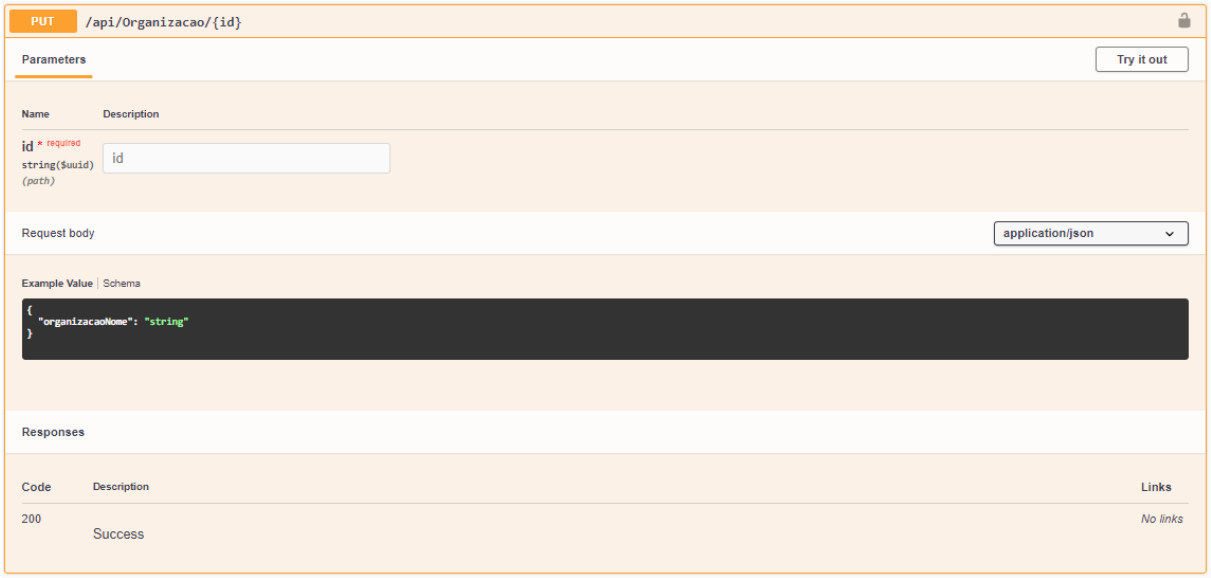
Figura 10 - Obtém organização por ID



Fonte: Autoria própria (2022).

Por último na *controller* Organizações há a operação usando o verbo HTTP PUT, para realizar a atualização do nome de uma organização governamental utilizando como parâmetro seu ID, apresentado na Figura 11.

Figura 11 - Atualiza o nome da organização

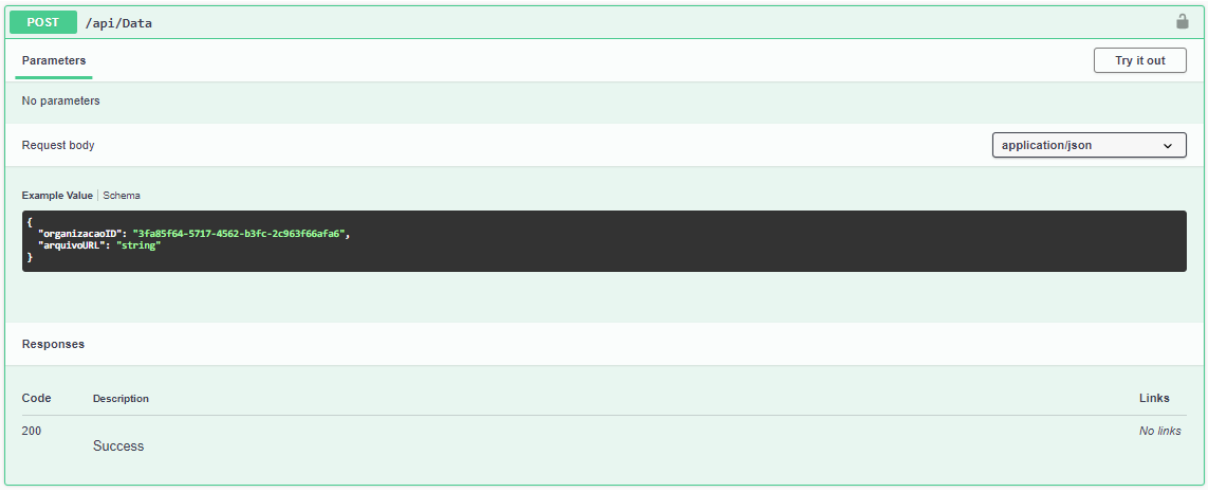


Fonte: Autoria própria (2022).

Já na controller de Data há apenas duas operações possíveis: a de agendar uma compressão de conjunto de dados ao *Worker Service* e a de buscar o endereço de um conjunto de dados já comprimido.

Na primeira operação é utilizado o verbo HTTP POST, informando na requisição o ID da organização detentora daquele conjunto de dados e a URL em que o *Worker Service* deve buscar os dados a serem comprimidos e armazenados no S3, apresentado na Figura 12.

Figura 12 - Agendamento de compressão



Fonte: Autoria própria (2022).

Já a operação de busca de conjunto de dados já comprimidos é realizada informando o ID da organização detentora do conjunto de dados, o ID do documento que contém os dados comprimidos anteriormente.

3.2.3 Desenvolvimento e testes

O desenvolvimento foi realizado por meio das tecnologias dispostas no quadro de materiais. Os testes por sua vez foram unitários e de integração, codificados a partir da biblioteca de teste xUnit.

Será apresentada a arquitetura do projeto e também a documentação na subseção 3.2.3.1.

3.2.3.1 Arquitetura do projeto, padrões utilizados e documentação da API

Visando melhor facilidade para manutenção e também boas práticas de programação foram aplicados padrões de projeto, arquitetura de código e arquitetura de aplicação, abaixo será detalhado as práticas utilizadas em cada etapa do desenvolvimento.

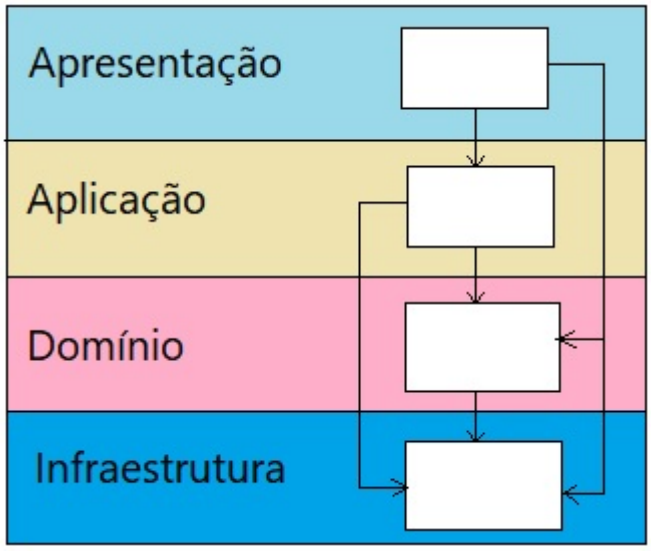
Arquitetura em camadas DDD:

DDD é uma forma de *design* de *software* e o termo foi utilizado pela primeira vez por Eric Evans, autor do livro “Domain-Driven Design: Atacando as complexidades no coração do *software*”.

Nele Evans apresenta práticas e padronizações para desenvolvimento de *software*, trazendo o foco para as entidades de domínio (Entidades que geralmente representam uma ação ou objeto real).

A escolha pelo DDD é baseada no foco que essa forma de *design* ganhou pela indústria, onde atualmente, é uma das formas de *design* mais aplicadas. Na Figura 13 é apresentada uma das formas mais comuns de aplicação do DDD.

Figura 13 - Diagrama de camadas DDD



Fonte: José Carlos Macoratti (2020).

Entrada de dados utilizando padrão CQRS:

CQRS (FOWLER, 2011) é um padrão arquitetural de *software* e visa separar a responsabilidade da consulta e da escrita. Sendo assim, esse padrão foi aplicado em todas as rotas existentes na API.

As operações de manipulação dos dados como criações, atualizações ou exclusões são realizadas por *Commands* (Comandos) enquanto operações de consultas são realizadas por *Queries* (Consultas).

Cada comando ou consulta tem uma função responsável por realizar a operação, essa função é chamada de *Handler* (manipulador, responsável por realizar a operação).

Operações no banco de dados realizadas pelo padrão *repository*:

O padrão *repository* é um padrão de projeto que visa por meio de funções manipular e aplicar comportamentos nas operações no banco de dados.

Sendo assim, geralmente cada entidade de domínio tem seu repositório específico, já que não é algo incomum a prática de ter comportamentos específicos para diferentes entidades.

Documentação da API:

Geralmente toda API tem uma documentação com exemplos de requisições para que o cliente possa começar a se familiarizar com o padrão das requisições.

A API desenvolvida neste trabalho utiliza o padrão REST para disponibilização dos dados e rotas de acesso e para apresentar uma documentação de forma organizada e dentro dos padrões do REST é utilizada a biblioteca *Swagger.*

Swagger é uma biblioteca que gera de forma automatizada páginas de documentação de APIs para consulta por outros clientes. Sendo assim com apenas algumas configurações na API a mesma passará a ter uma rota customizada para a documentação gerada (geralmente definida como o domínio em que a API está hospedada /swagger).

Conforme apresentado na subseção 3.2.2 são apresentadas figuras contendo as rotas disponíveis na API, a visualização apresentada nessas figuras é a documentação gerada pelo *Swagger*.

4 RESULTADOS

4.1 Resultados Obtidos

Os resultados obtidos estão dispostos nas subseções 4.1.1 onde é abordado o comparativo entre o algoritmo Brotli e o GZip e a subeção 4.1.2, onde é descrita a aplicação do trabalho realizado em um cenário real de uso.

4.1.1 Comparativo entre o algoritmo Brotli e o GZip

GZip é um algoritmo de compressão muito utilizado para compressão de arquivos para a *web*. O GZip é baseado em outros dois algoritmos de compressão, o LZ77 e o *Huffmann coding*.

Frequentemente são realizadas comparações entre Brotli e GZip, pois o Brotli foi apresentado como uma nova alternativa para compressão de arquivos para *web* mais eficiente que o GZip e de fácil implementação.

Comparando GZip e Brotli em arquivos Javascript há uma compressão de cerca de 14% a menos do Brotli em relação ao GZip, arquivos HTML uma redução de cerca de 21% do Brotli em relação ao GZip e arquivos CSS uma redução de cerca de 17% do Brotli em relação ao GZip (PANDJAROV, 2021).

Trazendo para a realidade do trabalho, realizando testes com 10 arquivos do tipo JSON obtidos do portal de dados abertos do Brasil com tamanhos entre 100MB e 10GB a redução foi entre 15 a 20%, indo de encontro com os dados apresentados anteriormente. No Quadro 6 é possível visualizar a disposição com o tamanho do arquivo original, o tamanho obtido com GZip, o tamanho obtido com Brotli e a porcentagem de redução do Brotli em relação ao GZip.

Quadro 6 - Comparativo entre GZip e Brotli

Número	Tamanho Original	Tamanho GZip	Tamanho Brotli	Redução Brotli em relação ao Gzip
1	153MB	7,65MB	6,36MB	16,85%
2	1460MB	102,2MB	83,44MB	18,35%
3	10345MB	2069MB	1699,89MB	17,84%
4	4300MB	378,4MB	326,60MB	13,68%
5	452MB	81,36MB	69,26MB	14,87%
6	3145MB	241,86MB	205,38MB	15,08%
7	6421MB	2119MB	1738MB	17,98%
8	9691MB	592,12MB	573,76MB	18,36%
9	6401MB	1714,82MB	1374,59MB	19,84%
10	5495MB	597,63MB	486,05MB	18,67%

Fonte: Autoria própria (2022).

No Quadro 6 foram dispostos os dados referentes a 10 testes de compressão utilizando GZip e Brotli, evidenciando a vantagem do Brotli em relação ao GZip em redução de tamanho final de arquivo, tendo média de 17,15% em relação aos 10 cenários.

Com base nos objetivos propostos é possível avaliar que todos foram atendidos.

O objetivo de definir a estrutura da aplicação foi alcançado e apresentado na subseção 3.2.1 e 3.2.3.

O objetivo de realizar o *download* de arquivos da *web* independente do formato foi alcançado e apresentada a função que realiza os *downloads* dos arquivos.

Já a compressão dos arquivos armazenados foi realizada utilizando o algoritmo Brotli e os resultados foram apresentados nesta subseção, obtendo ganhos expressivos.

Com base nos resultados de compressão foi notada uma redução considerável no tamanho de arquivos de cerca de 15% a 20% em média (arquivos baseados em texto como JSON e CSV).

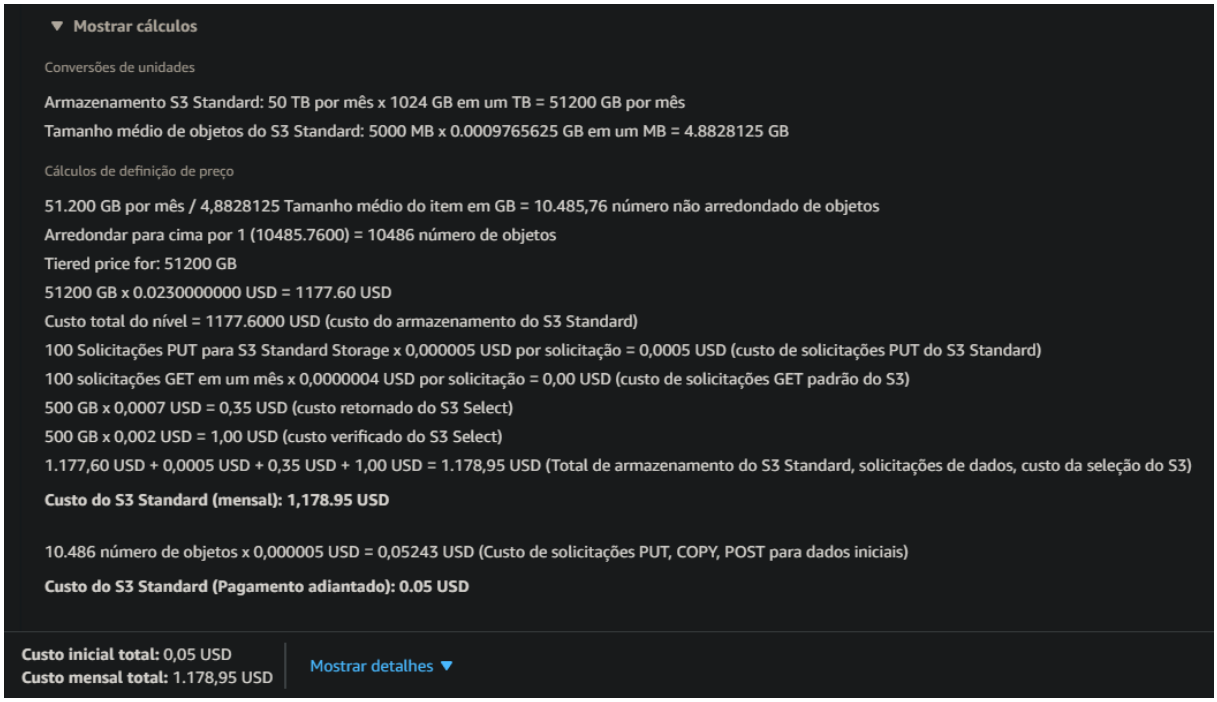
Porém, mesmo com a redução, um impeditivo pode ser o custo de armazenamento dos arquivos no AWS S3. Foi realizada uma estimativa de custos apenas de armazenamento no S3 para 50TB por mês (valor baixo visto que conjuntos de dados podem ocupar vários *gigabytes* de armazenamento). O valor estimado foi de $1178,95 dólares americanos mensais, conforme Figura 14.

Sendo que são cobrados $0,023 dólares americanos por *gigabyte* por mês, o cálculo realizado é 51200GB x $0,023, com um total de $1177,60 dólares. Além disso, deve ser considerado o número de acessos, atualizações e criações de arquivos, uma vez que cada operação dessas tem um custo.

Cada operação PUT tem um custo de $0,000005, ou seja, cada criação ou atualização de arquivo.

Porém, o custo mais relevante além do custo de armazenamento do arquivo é o custo de obter os arquivos, que é cobrado por quantidade de dados transferidos. Para uma transferência de 500GB é realizada uma cobrança de $1,00 dólar americano.

Figura 14 - Estimativa de custos de armazenamento AWS S3



Fonte: Autoria própria (2022).

4.1.2 Aplicando o trabalho realizado

Utilizando a aplicação desenvolvida foi cadastrada a organização com o nome “Agência Nacional de Telecomunicações - Anatel”, nela foi realizada uma solicitação para o método POST para cadastro de uma ordem de compressão do conjunto de dados de Acessos - Banda Larga Fixa (densidade por 100 habitantes)[2] em formato JSON.

Foi alcançado uma redução de 563MB para 457,77MB, uma redução de 18,69%. Um fator determinante para a capacidade de compressão é a repetição de sequências de caracteres dentro do arquivo. Quanto maior a repetição, maior será a compressão. Observando conjuntos de dados da CVM (Comissão de Valores Imobiliários), por exemplo, é possível observar que há diversas repetições de termos no arquivo, uma vez que são termos de índices financeiros observados.

2 https://dados.gov.br/dataset/densidade_banda_larga

5 TRABALHOS FUTUROS

Como trabalho futuro é possível criar uma aplicação para acesso visual a API desenvolvida, que tem um repositório criado no site de repositório de códigos online GitHub (HASSEL NETO, 2022), não necessitando realizar as chamadas HTTP diretamente a ela ou mesmo realizando uma integração direta com a API caso não seja o desejado.

Essa nova aplicação irá consumir a API desenvolvida e realizar todas as operações já disponíveis hoje, porém com uma interface visual, tornando mais prático o acesso e as operações a serem realizadas.

6 CONCLUSÃO

Neste trabalho, foi proposto uma aplicação para realizar a leitura, compressão e envio dos arquivos para o AWS S3 com a disponibilização do link para acesso. Os arquivos utilizados são públicos disponíveis no Portal de Dados Abertos do Brasil. Foram realizados experimentos a fim de explorar algoritmos de compressão, levando em consideração eficiência, utilização pela comunidade de programação, longevidade dentre outros aspectos.

Dentre os experimentos realizados, o algoritmo que obteve os melhores desempenhos dentre os escolhidos para análise foi o Brotli, evidenciando que a utilização de mais de uma abordagem para compressão é uma alternativa válida e com grande aplicabilidade, onde o método utilizado vai de encontro com o existente na literatura, mostrando que algoritmos de compressão podem beneficiar o armazenamento de arquivos para *Business Intelligence*.

REFERÊNCIAS

ASHRAFI, Noushin; KELLEHER, Lori; KUILBOER, Jean-Pierre. **The impact of business intelligence on healthcare delivery in the USA**. Interdisciplinary Journal of Information, Knowledge, and Management, n. 9, p. 117-130, 2014. Disponível em: http://www.ijikm.org/Volume9/IJIKMv9p117-130Ashrafi0761.pdf Acesso em: 25 jul. 2021.

CASARTELLI, A. O. et al. Inteligência estratégica em instituições de ensino superior. Rev. **Perspectivas em Ciência da Informação** [online], v.15, n. 2. 2010. Disponível em: http://portaldeperiodicos.eci.ufmg.br/index.php/pci/article/view/952/740. Acesso em: 25 jul. 2021.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** Disponível em : https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Acesso em: 03 nov. 2022.

FOWLER, Martin. **CQRS**, 14 jul. 2011. Disponível em: https://martinfowler.com/bliki/CQRS.html. Acesso em: 03 nov. 2022.

FORTULAN, Marcos Roberto; FILHO, Eduardo Vila Gonçalves. Uma proposta de aplicação de business intelligence no chão-de-fábrica. **Gestão & Produção**, [s. l.], v. 12, n. 1, p. 55-66, 25 fev. 2005. Disponível em: https://www.scielo.br/j/gp/a/ydtVGxxBtD65zcx4VmJDJGw/?format=pdf&lang=pt Acesso em: 25 jul. 2021

GUEST, Paul. **Video Resolution VS. Frames Per Second.** Disponível em: https://thinpigmedia.com/blog/decisions-decisions-video-resolution-vs-frames-per-second#:~:text=One%20minute%20of%20video%20is,MB%20with%201080p%20at%2060fps. Acesso em: 08 nov. 2022.

HÄNEL, Tom; FELDEN, Carsten. Design and evaluation of an analytical framework to analyze and control production processes. **10th CIRP Conference on Intelligent Computation in Manufacturing Engineering - CIRP ICME '16**, [s. l.], p. 141-146, 2017. DOI 10.1016. Disponível em: https://www.sciencedirect.com/science/article/pii/S2212827116306709. Acesso em: 25 jul. 2021.

HARIHARAN, Naveen Kunnathuvalappil. **DATA SOURCES FOR BUSINESS INTELLIGENCE**. International Journal of Innovations in Engineering Research and Technology. Disponível em: https://repo.ijiert.org/index.php/ijiert/article/view/2852. Acesso em: 24 out. 2022.

KODITUWAKKU, Saluka Ranasinghe; AMARASINGHE, Upali. **Comparison of lossless data compression algorithms for text data.** Disponível em: https://d1wqtxts1xzle7.cloudfront.net/61774586/Comp_com20200113-81589-6qysrz-

with-cover-page-v2.pdf?
Expires=1667962364&Signature=KFW~WRX2CvfQg9iEU3XFMEpsEkL5zoRi12Z9T
bDh4mIlb3C9l3ZzR8op--
cEnrzYHTx3Gwq9vgZuau6Gwq8DQR0dV57bTXCl3yBG6ryXuvMsBV~8~Q1~r7uRn
oiI2GghnEhieNb4dH6aOOOs9gguLO26kOuhQr29IKAXXCuc9HBtXLdZLHmuOtLGW
4bw8FsMgKOealS29EjLA02kIQQ6hGLu6c2PGA6xh0g3J7nizKsh5QPlnQU2NonfEJtf
ozA1TMPldpgbacJnmeTumfN9eVG0no2EzDSo5rTYxBAu2JhjNPgZKaUsh831Hb-
v6TX0nJcZ~r0Uf7AlWIng7LNHYA__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.
Acesso em: 08 nov. 2022.

MACHADO, Felipe Nery Rodrigues. **Tecnologia e Projeto de Data Warehouse.** Disponível em: https://books.google.com.br/books?hl=pt-BR&lr=&id=1YqwDwAAQBAJ&oi=fnd&pg=PT2&dq=data+warehouse&ots=P5DPrHCSY4&sig=GgtjvcqeJzwwzXJpX1rJP8l9na8#v=onepage&q=data%20warehouse&f=false. Acesso em: 23 out. 2022.

MUSSA, Munir de Sá; DA HORA, Henrique Rego Monteiro; FREIRE, Eduardo Francisco da Silva. **Business Intelligence in education on: An application of pentaho software**. Revista Produção e Desenvolvimento, [s. l.], v. 4, n. 3, p. 29-41, 2018. Disponível em: https://www.researchgate.net/profile/Henrique-Da-Hora/publication/326410416_BUSINESS_INTELLIGENCE_IN_EDUCATION_AN_APPLICATION_OF_PENTAHO_SOFTWARE/links/5b4b9e53a6fdccadaecefab4/BUSINESS-INTELLIGENCE-IN-EDUCATION-AN-APPLICATION-OF-PENTAHO-SOFTWARE.pdf. Acesso em: 25 jul. 2021.

PANDJAROV, Hristo. **More Site Speed Gains with Brotli Compression Algorithm, 13 jan. 2021.** Disponível em: https://www.siteground.com/blog/brotli-compression/. Acesso em: 10 nov. 2022.

SALOMON, David. **Data Compression. The Complete Reference 3rd Edition.** Disponível em: https://doc.lagout.org/Others/Information%20Theory/Compression/Data%20Compression%20The%20Complete%20Reference%203rd%20Ed%20-%20David%20Salomon.pdf. Acesso em: 09 nov. 2022.

SZABADKA, Zoltan. **Introducing Brotli: a new compression algorithm for the internet, 22 set. 2015.** Disponível em: https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html. Acesso em: 10 nov. 2022

V K. **LZW Coding 1. Youtube, 14 set. 2014. Disponível em:** https://www.youtube.com/watch?v=rZ-JRCPv_O8&ab_channel=VK. Acesso em: 10 nov. 2022.

O que é API?. Red Hat, 31 de out. 2017. Disponível em: https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces#:~:text=As%20APIs%20s%C3%A3o%20uma%20maneira,clientes%20e%20outros%20usu%C3%A1rios%20externos. Acesso em: 13 nov. 2022.

API. MDN Web Docs, 2021. Disponível em: https://developer.mozilla.org/pt-BR/docs/Glossary/API. Acesso em: 16 nov. 2021.

Painel COVID-19 Joinville. Secretaria de Saúde de Joinville. Disponível em: https://app.powerbi.com/view?r=eyJrIjoiMThiZGI5OTEtM2IxYS00OGI2LThkZTktZDE1Mzc5MTMyZDE3IiwidCI6IjRiMTAxNTc3LTMxY2EtNDQzNi04NjA5LWZmN2U1MDE1MDg3MiJ9. Acesso em: 16 nov. 2021.

Brotli. Google. Disponível em: https://github.com/google/brotli. Acesso em: 11 nov. 2022.

UtfprDados. Reinaldo Hassel Neto. Disponível em: https://github.com/reinaldohneto/UtfprDados. Acesso em 16 nov. 2022

ASP .NET Core - API com VS Code e conceitos do DDD, 03 nov. 2020**.** José Carlos Macoratti. Disponível em: https://www.macoratti.net/20/07/aspnc_ucddd1.htm. Acesso em: 17 nov. 2022

Serviços de Trabalho no .NET, 22 set. 2022**.** Microsoft. Disponível em: https://learn.microsoft.com/pt-br/dotnet/core/extensions/workers. Acesso em 16 nov. 2022.