UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VITÓRIA HELIANE PEREIRA DOS SANTOS SOBRINHA

HEALTHCHECKAPI:
MONITORANDO APIS GRPC

CAMPINA GRANDE - PB

2023

VITÓRIA HELIANE PEREIRA DOS SANTOS SOBRINHA

HEALTHCHECKAPI:
MONITORANDO APIS GRPC

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

Orientador: Professor Dr. João Arthur Brunet Monteiro.

CAMPINA GRANDE - PB

2023

VITÓRIA HELIANE PEREIRA DOS SANTOS SOBRINHA

HEALTHCHECKAPI:
MONITORANDO APIS GRPC

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. João Arthur Brunet Monteiro

Orientador – UASC/CEEI/UFCG

Professor Dr. Dalton Dario Serey Guerrero

Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em 14 de Fevereiro de 2023.

CAMPINA GRANDE - PB

ABSTRACT

Launched by Google in 2016, gRPC is a framework based on RPC (Remote Procedure Call) for creating APIs that has become increasingly popular among companies, startups and open-source projects. As it is a new technology, monitoring, observability and maintenance tools that support gRPC are quite scarce in the software development community. Furthermore, the few solutions available are often difficult to configure or do not produce desired metrics, such as call execution latency. Given this context, the purpose of this work is to develop HealthCheckAPI, a gRPC API monitoring platform. This application is capable of monitoring all endpoints available through the API through periodic calls to the service, obtaining availability and response time information, in addition to allowing the user to configure validations for the message returned by the API, as is done on the Assertible and Checkly platforms which are for monitoring REST APIs only.

HealthCheckAPI: Monitorando APIs gRPC

Vitória Heliane Pereira dos Santos Sobrinha
Universidade Federal de Campina Grande
vitoria.sobrinha@ccc.ufcg.edu.br

Orientador: João Arthur Brunet Monteiro
Universidade Federal de Campina Grande
joao.arthur@computacao.ufcg.edu.br

RESUMO

Lançado pelo Google em 2016, gRPC é um framework baseado em RPC (Remote Procedure Call) para criação de APIs que vem se tornando cada vez mais popular entre empresas, startups e projetos open-source. Por ser uma tecnologia nova, ferramentas de monitoramento, observabilidade e manutenção que deem suporte gRPC são bastante escassas na comunidade de desenvolvimento de software. Além disso, as poucas soluções disponíveis geralmente são difíceis de configurar ou que não produzem métricas desejadas, como a latência da execução das chamadas. Diante desse contexto, o propósito deste trabalho é desenvolver o HealthCheckAPI, uma plataforma de monitoramento de APIs gRPC. Essa aplicação é capaz de monitorar todos os endpoints disponíveis pela API através de chamadas periódicas ao serviço, obtendo informações de disponibilidade e tempo de resposta, além de permitir ao usuário configurar validações para a mensagem retornada pela API, assim como feito nas plataformas Assertible e Checkly que são para monitoramento apenas de APIs REST.

Palavras-chave

gRPC, protobuf, monitoramento, observabilidade, API.

Repositórios

https://github.com/healthcheckapi,
https://github.com/vitoria/healthcheckapi

Link de acesso à ferramenta desenvolvida

https://healthcheckapi-admin.vercel.app

1. INTRODUÇÃO

Criado pelo Google em 2015 com o propósito de desenvolver uma tecnologia que melhor integrasse os diversos microsserviços existentes na empresa, o Google Remote Procedure Call (gRPC) é um framework para criação de APIs rápidas e robustas baseado no protocolo Remote Procedure Call (RPC) [1][2]. Usando HTTP/2 e protocolo buffers, o gRPC consegue prover robustez, escalabilidade, desempenho e segurança para os serviços que fazem uso dessa tecnologia [2]. Por isso, mesmo sendo um framework recente - tendo sido lançado no ano de 2016 - tem se tornado cada vez mais popular entre as startups, projetos open-source e grandes empresas do mercado, como Netflix, Spotify e OpenIA, que vêm adicionando o gRPC no seu conjunto de tecnologias [2][3].

Após a criação de uma API ou serviço, é necessário garantir que os requisitos funcionais e não-funcionais estejam sendo respeitados à medida que novas alterações são feitas no sistema. Nesse sentido, podem ser utilizados softwares desenvolvidos com o propósito de monitorar, observar e analisar métricas da aplicação, como latência, tempo de funcionamento e quantidade de requisições. Esses softwares são chamados de Application Performance Management (APM) e são ferramentas comumente usadas tanto por engenheiros, a fim de acompanhar as métricas do sistema para atuar de forma rápida caso algum problema surja, como por pessoas de produto para conseguir mensurar a qualidade e operabilidade dos produtos oferecidos [4].

Sabendo da importância das ferramentas de APM, o mercado de software dispõe de diversas soluções nesse segmento. Segundo o ranking de ferramentas de APM mais bem avaliadas do Gartner, as principais são Dynatrace, New Relic One, AppDynamics, DataDog e Pingdom, porém nenhuma delas é capaz de fazer monitoramento de APIs gRPC, pois são todas voltadas para APIs REST [5]. Embora gRPC esteja se tornando bastante popular nos últimos anos, ainda são muito limitadas as ferramentas capazes de monitorar aplicações construídas em cima dessa tecnologia, pois o principal foco desse tipo de ferramenta é dar suporte ao modelo de APIs mais comum no mercado, o REST.

Algumas das poucas ferramentas que dão suporte a gRPC são Istio e Prometheus junto com o Grafana que possibilitam ter acesso a registro de logs, gráficos de tráfego e latência de uma perspectiva mais interna da aplicação [6][7]. Porém, essas ferramentas são difíceis de configurar, complexas para utilizar ou são pagas. Dentro dessa coleção de ferramentas, também existe a plataforma do Google Cloud criada pela empresa genitora do gRPC que também pode ser usada nesse propósito e promete uma boa usabilidade e visualização das métricas, porém também é uma ferramenta paga [8]. Mesmo assim, também apresenta métricas apenas de uma perspectiva mais interna das aplicações.

Por outro lado, existem ferramentas que foram desenvolvidas para APIs REST que realizam monitoramento dos endpoints da aplicação validando não somente o status e tempo de resposta das requisições, mas fazem validações mais elaboradas usando mecanismo de asserções, como Assertible e Checkly [9][10]. Dessa forma, as pessoas usuárias podem definir quais informações devem ser mandadas nas requisições de teste, além de poder configurar o que é esperado como resposta dessas solicitações. Assim, o monitoramento é garantido não somente para as métricas mais comuns de infraestrutura, mas também de regras de negócio.

Portanto, o propósito deste trabalho foi desenvolver o HealthCheckAPI, uma ferramenta open-source de monitoramento de API gRPC que seja fácil de configurar e que proporcione uma boa experiência para o usuário final. Assim como o Google

Cloud, essa ferramenta deve calcular as métricas de acordo com cada função gRPC definida no serviço através de chamadas periódicas configuradas. Essa plataforma oferece métricas calculadas do ponto de vista do usuário final dos serviços, promovendo a criação de valores calculados de acordo com o contexto realístico da aplicação. Assim como gRPC, essa plataforma será agnóstica ao ambiente que a API estará definida, tal como a linguagem de programação implementada. Com ela é possível ter acesso a métricas de latência, erros e tempo de disponibilidade da API.

2. Metodologia

Esta seção descreve as etapas e os processos que serão adotados para a realização deste projeto. Todo o projeto foi conduzido seguindo algumas práticas da metodologia ágil Scrum, com rituais de sprint de uma semana e alinhamentos periódicos feitos com o orientador do projeto a cada final de sprint, encontros similares a planning e retrospectivas da metodologia Scrum. Todas as atividades serão gerenciadas pelo Github Projects, uma ferramenta da Microsoft criada para auxiliar na gestão de projetos.

Abaixo encontram-se listadas todas as etapas do projeto.

Investigação do Estado da Prática. A primeira etapa definida para a condução deste projeto é investigar o estado-da-prática para entender melhor as ferramentas que já existem no mercado. Como resultado concreto desta atividade, será feita uma lista das ferramentas mais utilizadas para monitoramento de APIs gRPC e REST, elencando seus pontos positivos e negativos.

Especificação de requisitos da plataforma. Após feito o levantamento das principais ferramentas de monitoramento do mercado, foi criada a especificação dos requisitos da plataforma. De forma a tentar resolver os principais pontos negativos apresentados na listagem das ferramentas no tópico 1. Ao fim desta etapa foi produzido um documento completo englobando todos os requisitos funcionais e não-funcionais da plataforma desenvolvida.

Prototipação. Feito os requisitos da plataforma, nesta etapa foi desenvolvido o protótipo de como seriam os fluxos de uso e o design das telas da ferramenta. A prototipação teve foco nas principais funcionalidades do sistemas, priorizando a usabilidade e boa experiência do usuário final. O resultado desta tarefa foi a prototipação completa do projeto feita no Figma, uma ferramenta para construção de designs e protótipos.

Definição das tecnologias e arquitetura do sistema. Antes de iniciar a etapa de implementação da ferramenta, foi necessário definir quais seriam as tecnologias que seriam utilizadas (e.g. frameworks, linguagem de programação) e a arquitetura do sistema (e.g. monolito, microsserviços). O entregável desta etapa foi um documento descrevendo quais tecnologias foram utilizadas e o porquê de cada uma. Além de uma documentação junto de diagramas explicando como será a arquitetura do serviço.

Implementação do MVP. Já tendo todos os artefatos necessários para iniciar a fase de implementação (especificação, protótipo e tecnologias), esta etapa do projeto consistiu em desenvolver o Produto Viável Mínimo (MVP). O MVP deveria conter as funcionalidades principais da plataforma, tentando seguir ao máximo o que foi definido nas etapas anteriores. O entregável desta atividade foi o MVP funcionando e acessível para os possíveis usuários.

Teste da ferramenta com alguma organização que usa gRPC. Com o MVP pronto, a aplicação foi testada com alguns funcionários de uma organização real que utiliza gRPC em seus sistemas. Após esse teste, foi aplicado um questionário para avaliar o impacto da ferramenta e coletar informações de como a plataforma pode ser aperfeiçoada. Portanto, o entregável desta tarefa foram as avaliações feitas sobre a ferramenta testada.

Análise das avaliações. A partir das avaliações feitas pelos usuários do teste, foram feitas análises para extrair informações, como quais pontos devem ser melhorados na ferramenta e quais novas funcionalidades devem ser priorizadas. Por fim, o entregável foi um documento estruturado com as avaliações e as análises feitas a partir delas. Isso servirá como tomada de decisão para as próximas funcionalidades e correções que poderão ser implementadas na ferramenta.

3. Solução Proposta

Neste projeto foi desenvolvido o HealthCheckAPI, que consiste em uma plataforma de monitoramento de APIs gRPC capaz de fazer chamadas periódicas para os métodos RPCs definidos na configuração de um projeto criado pelo usuário na ferramenta. Para essas chamadas podem ser definidos o limiar de latência e a mensagem da requisição e o que é esperado que seja retornado como resposta, o que foi nomeado como *check*.

O resultado dessas requisições são exibidos para os usuários em uma dashboard onde pode-se ver gráfico de latência e quais foram as respostas retornadas nessas chamadas. Na Figura 1, é apresentado um exemplo da interface do HealthCheckAPI contemplando o dashboard de uma API de exemplo utilizada durante o desenvolvimento.

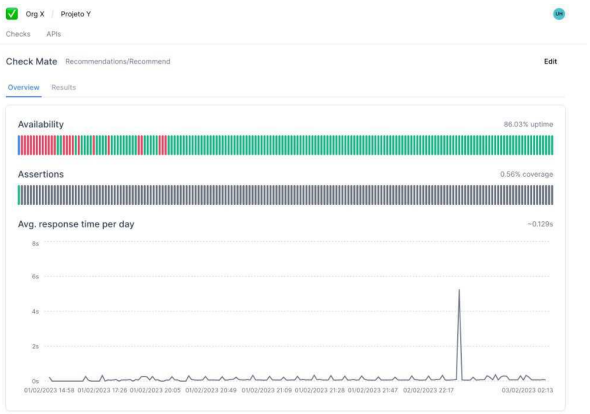

Figura 1: Screenshot de exemplo de UI, mostrando a dashboard de um check.

A arquitetura do HealthCheckAPI foi desenvolvida em alguns módulos, sendo eles o frontend, backend, CronService, CheckScript, usando o Supabase para armazenar os dados do sistema. A Figura 2 mostra um diagrama dessa arquitetura e como os módulos interagem entre si. Serão detalhados nesta seção cada aspecto arquitetural, especificidades e tecnologias utilizadas no desenvolvimento do HealthCheckAPI.

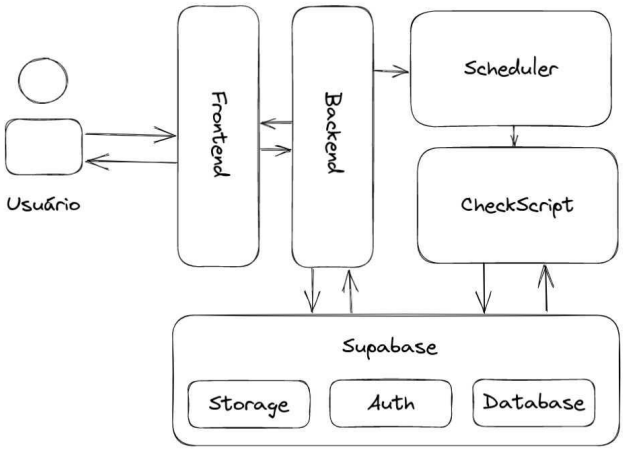

Figura 2: Arquitetura base do HealthCheckAPI.

3.1 Frontend

O módulo do *frontend* do HealthCheckAPI consiste em uma aplicação React implementada em Typescript usando Next.js, uma *framework* para aplicação web que tem se tornado cada vez mais popular e tem sido adotado por grandes empresas da indústria, como Netflix, Uber e OpenAI. O uso do Next.js além de tornar as aplicações web mais performáticas, facilita e agiliza bastante o desenvolvimento por já oferecer soluções práticas para roteamento e ser muito rápido para rodar a aplicação.

Para estilização, foi utilizada a biblioteca Tailwind. Essa diferente da maioria segue o princípio de não fornecer classes predefinidas para elementos como tabelas e botões, mas sim classes atômicas como alterar cor da margem ou mudar o modo de display de um elemento. Além disso, o Tailwind torna fácil a criação de designs responsivos oferecendo também as mesmas classes genéricas de acordo com os principais *breakpoints*. O radix-ui é uma biblioteca de componentes que fornece componentes básicos para criação de aplicações em React, cujo um dos seus princípios é que seus componentes tenham alta qualidade e ótimo suporte para acessibilidade, além de que seus componentes são sem estilo. Dito isso, neste projeto foi usado o radix-ui junto com o Tailwind para criar os elementos de visualização do HealthCheckAPI.

Este módulo do sistema é o único que o usuário tem interação e nele foram implementadas as principais funcionalidades do sistema, sendo elas: autenticação, criação de projeto, API e checks, e visualização dos resultados da execução dos checks criados. O diagrama da Figura 3 mostra como funciona o principal fluxo de interação do usuário com o sistema que é a criação de um check.

Como demonstrado no diagrama da Figura 3, para criar um check no HealthCheckAPI, primeiro o usuário precisa criar um projeto e registrar uma API no sistema. Para criar um projeto o usuário precisa apenas informar um nome e isso serve para agrupar os checks dentro de um contexto que seria o de projeto. Em seguida, é necessário que tenha sido registrada a API gRPC para a qual o check será criado. Uma API contém as informações necessárias para que o sistema consiga de fato fazer chamadas para ela. Desta forma, é preciso informar um nome - tem o propósito apenas de fácil identificação da API por parte do usuário - e URL de acesso. Para completar o registro da API, se faz necessário fazer também o *upload* dos arquivos *protobuf* que contenham o contrato da API (definição do serviço) e todas as mensagens usadas na mensagem de requisição e respostas.

Já estando com o projeto e API criados e selecionados, é possível criar os checks. Na criação de um check, o usuário precisa preencher os seguintes campos:

- **nome:** nome de identificação do check;
- **serviço:** nome do serviço implementado pela API;
- **método:** nome do método do serviço que deve ser executado no check;
- **intervalo:** intervalo em minutos em que o check deve ser executado;
- **mensagem da requisição:** preenchimento de cada campo da mensagem de requisição definida no contrato do serviço de acordo como o usuário deseja que seja enviada nas chamadas do check;
- **asserção:** mensagem de acordo com o que é esperado que seja retornado na chamada do check.

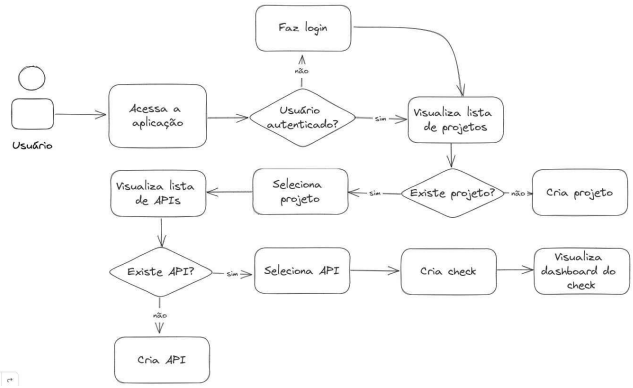

Figura 3: Diagrama de fluxo do usuário para criação de um check.

Logo após a criação do check, o usuário pode ter acesso a dashboard que contém as informações básicas conforme imputadas na criação, incluindo o resultado de todas chamadas que o HealthCheckAPI fez, assim como mostrado na Figura 1. O resultado das chamadas contém a latência de execução do método da API - consiste no delta do tempo em que a chamada foi feita e o tempo que o sistema obteve uma resposta ou erro - essa latência é mostrada em um gráfico de linha que relaciona o *timestamp* de quando a chamada foi feita e a latência (tempo de resposta).

Na dashboard do check também é possível visualizar um gráfico de disponibilidade da API para quando a chamada foi feita, fazendo também um cálculo aproximado de *uptime* para o serviço. As cores verde, vermelha e azul representam o código de status retornado na chamada. Azul significa sucesso, vermelho, erro, e azul, um erro não reconhecido conforme os tipos de status definidos pelo próprio gRPC.

Ainda no dashboard do check, é possível ver o resultado da validação da resposta da chamada comparada a mensagem esperada de acordo com o que foi definido na asserção do check. O resultado dessa validação pode ser “MATCHED” para quando a mensagem esperada e a retornada na chamada forem compatíveis e “NOT MATCHED” em caso contrário. No dashboard, o gráfico que representa esse resultado a cor verde é para “MATCHED”, vermelho “NOT MATCHED” e cinza para quando não havia asserção cadastrada para o check.

Todas as chamadas feitas pelo check para a API registram não apenas essas informações, como também qual foi a

mensagem da requisição e a resposta. Esse histórico pode ser visto na tela “Results” do dashboard do check conforme mostrado na Figura 4. Todos os gráficos do projeto foram criados usando a biblioteca Tremor, que é uma biblioteca de gráficos para React que utiliza Tailwind para estilização, além de oferecer diversas opções de gráficos que além de serem fáceis de configurar são rápidos na renderização trazendo uma boa experiência tanto no desenvolvimento quando na usabilidade.

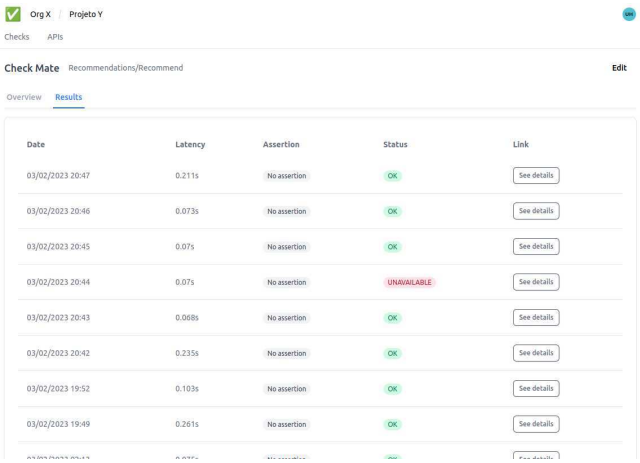

Figura 4: Aba de resultados do histórico de chamadas de um check.

3.2 Backend

Assim como o módulo do Frontend, o Backend também foi implementado em TypeScript usando o framework do Next.js. Esse framework já traz a possibilidade de junto com a aplicação web desenvolver a API necessária para lidar com a lógica de negócio do sistema. Desta forma, foi criada uma API RESTful disponível para o módulo do Frontend, os principais *endpoints* que essa API dispõe são os seguintes*:*

- [GET] /api/projects - lista os projetos do usuário autenticado;
- [GET] /api/project/:project_id - retorna os dados de um projeto específico;
- [GET] /api/project/:project_id/checks - lista os checks de um projeto específico;
- [GET] /api/project/:project_id/checks/:check_id - retorna os dados de um check em específico;
- [GET] /api/project/:project_id/checks/:check_id/results - retorna a lista dos resultados para um check;
- [GET] /api/project/:project_id/apis - lista as apis de um projeto específico;
- [POST] /api/project/:project_id/apis - cria uma nova api;
- [GET] /api/project/:project_id/apis/:api_id - retorna os dados de uma API específica e a lista de arquivos para essa API;
- [GET] /api/project/:project_id/apis/:api_id/details - lista os serviços e métodos da API;
- [POST] /api/project/:project_id/apis/:api_id/upload - faz upload dos arquivos *protobuf* da API.

Além de ser o responsável por acoplar as regras de negócio do HealthCheckAPI, esse módulo é responsável por armazenar e recuperar informações do Supabase, serviço que oferece soluções de autenticação, banco de dados e armazenamento. Neste projeto, todas essas soluções do Supabase foram utilizadas usando *supabase-js* - biblioteca fornecida pelo Supabase que torna extremamente fácil utilizar todas as suas soluções em um projeto implementado em JavaScript - e a seguir são apresentadas as motivações e ao que cada solução agregou ao projeto.

3.2.1 Banco de Dados

Todos os projetos no Supabase têm um banco de dados PostgreSQL dedicado e essa foi a solução para armazenamentos dos dados do HealthCheckAPI. Gerenciar o banco de dados, criar, apagar, alterar tabelas, adicionar e remover *policies* através do serviço do Supabase se tornam tarefas rápidas e fáceis de executar. Além de trazer todos os benefícios de um banco de dados PostgreSQL de ser robusto, rápido, relacional e um dos mais escaláveis bancos de dados existentes.

Abaixo, na Figura 5, é possível visualizar o diagrama relacional de como os banco de dados do HealthCheckAPI foi modelado.

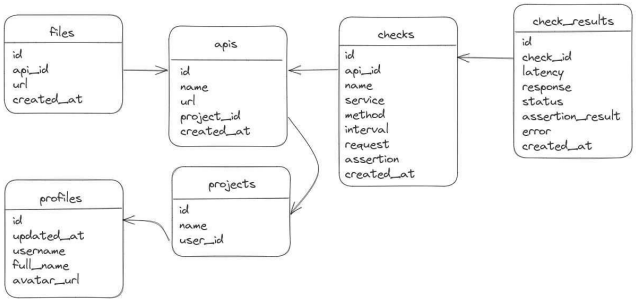

Figura 5: Diagrama relacional do banco de dados.

3.2.2 Armazenamento de arquivos

O HealthCheckAPI além de banco de dados, precisa de um sistema de armazenamento de arquivos para salvar os arquivos *protobuf* que o usuário oferece para as APIs. Por isso, foi utilizada a solução de *storage* também oferecida pelo Supabase que é capaz de armazenar arquivos de qualquer tipo, além de possibilitar a criação de políticas de permissão de acesso a esses arquivos de forma fácil de implementar. Além de salvar os arquivos proto das APIs, o storage também foi utilizado para salvar as fotos de perfil dos usuários.

3.2.3 Autenticação

Além de banco de dados e armazenamento de arquivos, o Supabase também oferece solução própria de autenticação de usuários para todos os seus projetos com um sistema completo de gerenciamento de usuários. Como o HealthCheckAPI precisa que o usuário esteja autenticado para ter acesso às funcionalidades do sistema, também foi utilizada essa solução do Supabase.

3.2.4 Deploy

Já que o framework utilizado para implementar tanto o módulo do Frontend quanto o do Backend foi o Next.js, o deploy a aplicação foi feita na plataforma da Vercel - criadora do Next.js. Nessa plataforma é possível fazer deploy rápido e eficiente de aplicações Next.js de forma gratuita, além de já oferecer integração com o Github para fazer deploy assim que alguma alteração no código for submetida para o repositório na *branch* principal. No entanto, a versão paga da Vercel apenas se aplica para repositórios criados em perfil pessoal no Github, repositórios pertencentes a alguma organização precisam de adesão a algum plano pago da

plataforma. Diante disso, o Frontend e o Backend ficaram em um repositório fora da organização do HealthCheckAPI para que pudesse ser feito deploy usando a Vercel.

3.3 Scheduler

Cada projeto pode definir vários checks que são chamadas periódicas que devem ser feitas para a API gRPC do usuário em um intervalo de tempo definido na sua configuração. Para isso, foi criado o Scheduler, um microsserviço em Python que faz uso do Crontab do Linux para fazer o agendamento de jobs para rodarem de forma periódica.

O Scheduler é usado pelo Backend através de chamadas para API RESTful definida no microsserviço. Essa API foi implementada usando o framework Flask do Python e expõe os seguintes endpoints. Um endpoint POST /scheduler para a criação de um novo agendamento quando um check for criado que recebe o ID do check e o intervalo em minutos para que ocorram as repetições.

Lidar com agendamento de tarefas em um sistema de forma robusta não costuma ser uma tarefa trivial e existem várias ferramentas no mercado que são capazes auxiliar nesse problema. Neste projeto foi utilizado o Cron que é um sistema de agendamento de tarefas do Unix. O Cron utiliza crontabs que são arquivos de texto para registrar e controlar a tarefas que devem ser executadas usando expressões de *cron* para definir os intervalos de execução.

Para Python existe uma biblioteca *python-crontab* que facilita o acesso aos arquivos de *crontab* do sistema permitindo a escrita e leitura de agendamento de tarefas do cron. Portanto, quando scheduler recebe uma chamada POST para o registro de um novo agendamento de checks, é criada uma nova entrada no crontab para executar o script de check que será detalhado na próxima seção. Essa nova entrada usa como identificador o ID do check que é usado no método de DELETE /scheduler da API para remover a entrada do crontab e assim para a execução do script de check para aquele check. O diagrama da Figura 3 traz uma forma visual de como é a arquitetura do Scheduler.

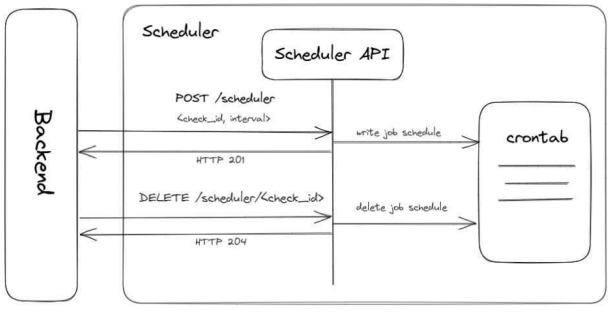

Figura 3: Arquitetura do Scheduler.

3.3.1 Deploy

De forma a facilitar as ações de *deploy* do Schedule, foram implementadas definições para conteinerização utilizando o Docker. Para isso foi criado um *dockerfile*, arquivo que contém a configuração da etapa de *build* e *startup* do microsserviço. Nesse *dockerfile* foi definido que a imagem a ser utilizada pelo container da aplicação seria o *Ubuntu 20.04* no qual é feita a instalação dos pacotes necessários para execução do Scheduler, sendo eles o *cron, Python* e o *PIP*. Logo após a instalação dos pacotes, é feita também a instalação das bibliotecas do Python utilizadas na aplicação que foram listadas no *requirements.txt,* arquivo que define quais bibliotecas e suas versões a aplicação requer. A última etapa é a execução de fato da API do Scheduler que é acessada pela porta 5000/tcp, por no arquivo de configuração é definido que o container expõe essa porta para acesso externo.

O deploy do Scheduler foi feito no *Railway.app,* uma plataforma de deploy simples e rápida para trazer aplicações para produção com poucas configurações. Fazer deploy no Railway.app é tão simples que basta conectar com o repositório no Github que a plataforma já faz toda a integração por conta própria fazendo com que a aplicação seja atualizada sempre que houver alterações no código da *branch* principal do repositório do Scheduler. Além disso, o *Railway.app* já identifica automaticamente o *dockerfile* e usa esse arquivo de configuração para fazer o *startup* da aplicação. O único passo extra necessário é adicionar as variáveis de ambiente e definir, também como variável de ambiente, qual é a porta que a aplicação estará rodando. A figura 6 mostra uma requisição feita para o Scheduler depois de ter feito deploy no Railway.app.

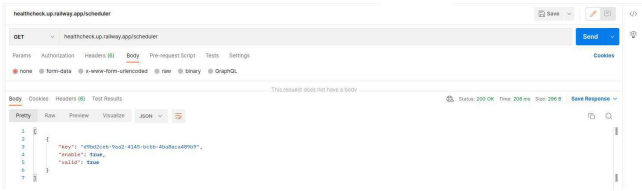

Figura 6: Exemplo de requisição para o Scheduler em produção no Railway.app usando o Postman.

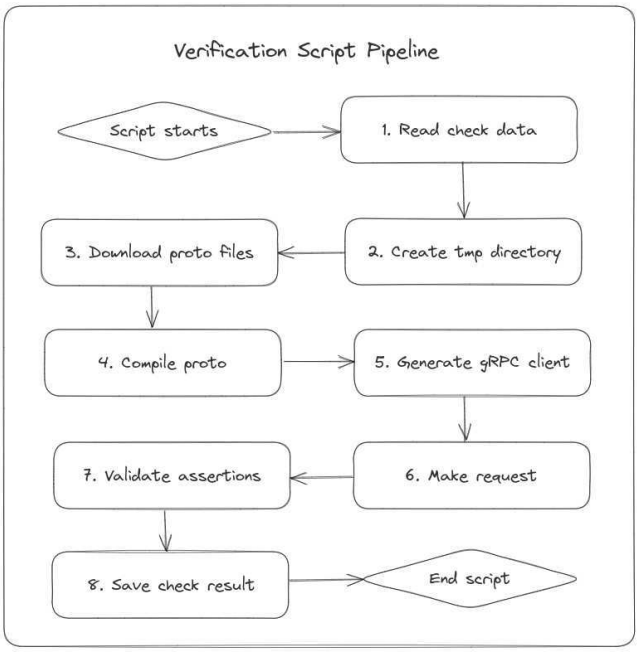

Figura 7: Diagrama do pipeline do script.

3.4 Script de verificação

A execução dos checks é feita através de um script em Python que dado o ID do check executa uma série de passos para fazer a chamada para o método RPC do serviço conforme definido na criação do check. O *pipeline* do script foi definido conforme a Figura 7 que mostra cada etapa e como é feita a interação entre elas. Em seguida cada etapa será detalhada.

3.4.1 Read check data

Em sua execução, o script precisa de algumas informações do check como URL da API, nome do serviço, método, mensagem da requisição e a mensagem esperada retornada pela API. No entanto, a única informação passada para o script é o ID do check, portanto a primeira etapa do script é consultar o *Supabase* para obter todas essas informações faltantes.

O *Supabase* não oferece suporte para *Python*, porém a comunidade desenvolveu uma biblioteca que permite contactar com o *Supabase* e fazer uso de todas as funcionalidades oferecidas pelo serviço, como autenticação, armazenamento e banco de dados. Nesta etapa do *pipeline* foi usada apenas a API de passar acessar o banco de dados do *Supabase* e obter as informações do check usando o ID fornecido como argumento na chamada de execução do script.

Desta forma, são feitas duas chamadas para o banco de dados, uma para pegar as informações na tabela de check filtrando pelo ID fornecido. E em seguida obtêm-se as informações da API configurada para o check em questão.

3.4.2 Criação do diretório temporário

Todos os arquivos gerados programaticamente para executar a chamada para a API são salvos em um diretório temporário específico para o check a ser executado. Esse diretório usa o ID para que não haja conflito entre execução de checks em paralelo. O padrão do diretório criado é "tmp/checks/<check_id>" e dentro dele serão salvos os arquivos do *protobuf* da API e o cliente que fará a chamada, a Figura 8 mostra como fica esse diretório na prática. Ao finalizar a execução do script, esse diretório é apagado a fim de não gerar conflito com a próxima execução desse check.

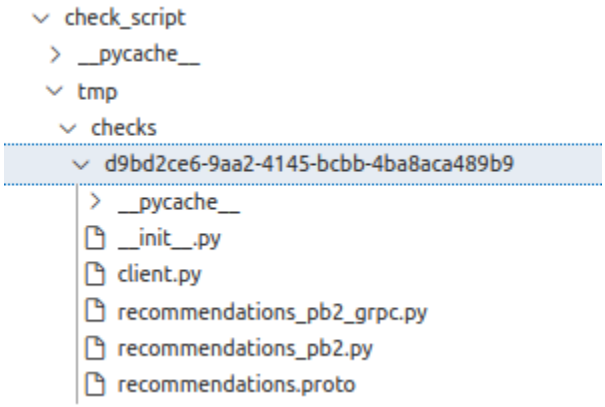

Figura 8: Exemplo de como o diretório temporário do script é criado.

3.4.3 Download protos

Qualquer chamada a uma API gRPC requer conhecer o contrato definido pelo serviço através de arquivos *protobuf.* Quando um check é criado, ele é associado a uma API já cadastrada na plataforma HealthCheckAPI. Essa API contém todos os arquivos *protobuf* salvos no armazenamento do *Supabase.* Então nesta etapa do script, também é feito uso do *Supabase* para baixar esses arquivos que são salvos no diretório temporário criados para o check.

3.4.4 Compilar os arquivos Protobuf

Para fazer uso do contrato definido por um *protobuf* é preciso gerar a sua representação na linguagem de programação desejada. Como nesse script a linguagem usada é Python, essa é a linguagem usada para criar essas representações. O próprio Google já oferece um compilador chamado *Protoc* que é capaz de ler proto e gerar suas representações para uma linguagem escolhida desde que essa esteja na lista de linguagens suportadas.

Usando o *protoc,* é passado o caminho do diretório temporário do check onde previamente foram salvos os arquivos ".proto". O compilador é capaz de ler todos os arquivos desse diretório cuja extensão seja ".proto" e nesse mesmo diretório ele cria as representações em Python. Abaixo na Figura 9 temos um exemplo desse comando que seria executado para um check.

```
/home/myuser/venv/bin/python -m grpc_tools.protoc --proto_path=tmp/checks/d9bd2ce6-
9aa2-4145-bcbb-4ba8aca489b9 --python_out=tmp/checks/d9bd2ce6-9aa2-4145-bcbb-4ba8aca
489b9 --grpc_python_out=tmp/checks/d9bd2ce6-9aa2-4145-bcbb-4ba8aca489b9 tmp/checks/
d9bd2ce6-9aa2-4145-bcbb-4ba8aca489b9/*.proto
```

Figura 9: Exemplo do comando de compilação dos *protobufs* do check.

3.4.5 Gerar o cliente gRPC

Além de conhecer e compilar os arquivos *protobuf,* é preciso criar um cliente que implemente a lógica para fazer requisição para o serviço definido na API. Esse cliente precisa ser gerado dinamicamente para que façam uso do protos que são conhecidos apenas em tempo de execução e também a definição da mensagem que será enviada na requisição.

Para isso, foi implementado um gerador para esse cliente que precisa apenas de alguns parâmetros como URL da API, nome do arquivo ".proto" no qual o serviço foi definido, nome do serviço, nome do método para o qual será feita a chamada e a mensagem da requisição definida na criação do check. Esse cliente também tem a função de calcular o delta do tempo entre o *timestamp* de quando a chamada foi feita e quando o método responder, essa informação é usada como o tempo de latência da execução desta requisição.

A seguir é mostrada a Figura 10 que contém o código gerado para o cliente de um check de exemplo.

```
healthcheck > check_script > tmp > checks > d9bd2ce6-9aa2-4145-bcbb-4ba8aca489b9 > client.py
import time
import grpc
from .recommendations_pb2 import RecommendationRequest
from .recommendations_pb2_grpc import RecommendationsStub


client = RecommendationsStub(grpc.insecure_channel("vitoria-example.fly.dev:9090"))
request = RecommendationRequest()


def make_request():
    start = time.time()
    try:
        start = time.time()
        response = client.Recommend(request)
        end = time.time()
        return {
            "error": None,
            "status": "OK",
            "response": response,
            "latency": end - start
        }
    except Exception as e:
        end = time.time()
        latency = end - start
        error = {
            "message": e.details(),
            "code": e.code().name,
            "debug_error": e.debug_error_string()
        }

        return {
            "error": error,
            "status": e.code().name,
            "response": None,
            "latency": latency
        }
```

Figura 10: Exemplo de cliente gerado dinamicamente.

3.4.6 Fazer requisição

Como pode ser observado na Figura 10, o cliente gerado na etapa

anterior define uma função que faz a requisição para o serviço. Já que foi gerado dinamicamente, esse cliente precisa ser importado dinamicamente também no fluxo principal do script.

Esse importação é feita usando a biblioteca do Python *importlib* que permite importar dinamicamente um módulo dado o diretório dele. Após a importação, basta chamar a função "make_request" que o cliente já encapsula toda a lógica de lidar com essa requisição para o serviço.

3.4.7 Validar asserções

A chamada da função "make_request" já retorna um dicionário que contém a resposta da requisição para o método do serviço RPC. Nessa etapa é feita a comparação entre a resposta real do método e o que é esperado pelo check como foi definido pelo usuário. O resultado dessa comparação define se o check foi sucesso ou falhou.

3.4.8 Salvar resultado do check

Já tendo todo o resultado da execução do check - código de status, mensagem de resposta ou erro, latência e resultado do match da resposta - basta salvar essas informações no banco de dados do sistema. Usando novamente a biblioteca do Supabase para Python essas informações são inseridas na tabela de "check_results" que serão utilizadas pela UI para mostrar ao usuário o desempenho da sua API conforme a execução do script.

4. Avaliação

Para avaliar o HealthCheckAPI foi desenvolvido um questionário com 8 perguntas com o objetivo de mensurar o impacto e a usabilidade da ferramenta. Esse questionário foi criado usando o Google Forms e distribuído através do Discord para membros de uma organização que usa gRPC com o principal framework para desenvolvimento das suas APIs.

No total, foram obtidas 13 respostas para o questionário, dessas respostas 76,9% das pessoas afirmaram possuir um serviço gRPC implementado ou rodando em ambiente de produção e 92,3% afirmaram já conhecer ou ter trabalhado com gRPC. Menos de 50% das pessoas afirmaram conhecer ferramentas que façam monitoramento de APIs gRPC, dessas as ferramentas que informaram ter conhecimento foram o Prometheus usado em conjunto com o Grafana e o Google Cloud, a Figura 11 mostra o gráfico de resposta para essa pergunta.

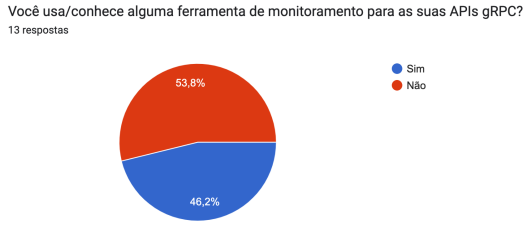

Figura 11: Gráfico em pizza das respostas para a pergunta "Você usa/conhece alguma ferramenta de monitoramento para APIs gRPC?".

Partindo para as perguntas específicas ao HealthCheckAPI, foram feitas perguntas para avaliar o impacto e a usabilidade da ferramenta do ponto de vista dos usuários de teste. Duas das perguntas foram elaboradas usando a escala Likert para medir a satisfação desses usuários com relação a usabilidade e impacto da ferramenta. As Figuras 12 e 13 mostram o resultado obtido para essas perguntas.

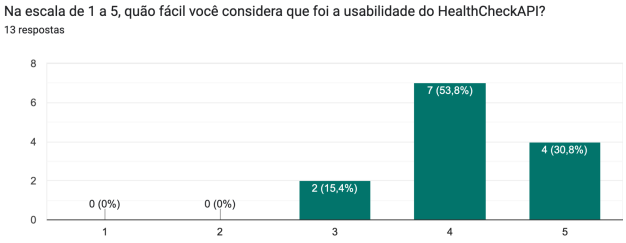

Figura 12: Gráfico em colunas das respostas para a pergunta "Na escala de 1 a 5, quão fácil você considera que foi a usabilidade do HealthCheckAPI?".

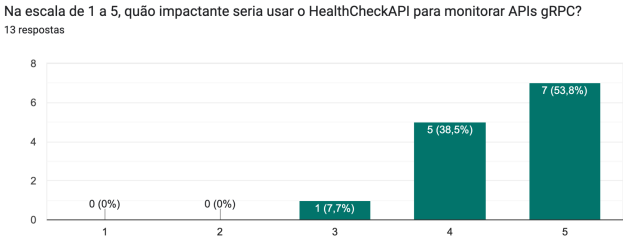

Figura 13: Gráfico em colunas das respostas para a pergunta "Na escala de 1 a 5, quão impactante seria usar o HealthCheckAPI para monitorar APIs gRPC?".

Também foi perguntado aos usuários se eles usariam ou recomendariam o HealthCheckAPI para fazer o monitoramento de APIs gRPC, das respostas 84,6% afirmam que usariam ou recomendariam, 15,4% não têm certeza e nenhum usuário respondeu que não usariam ou não recomendariam. A Figura 14 mostra o gráfico com essas respostas.

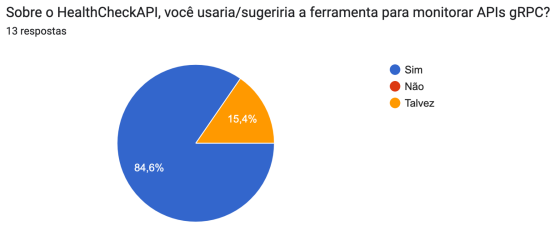

Figura 14: Gráfico de pizza das respostas para a pergunta "Sobre o HealthCheckAPI, você usaria/sugeriria a ferramenta para monitorar APIs gRPC?".

Por fim, foi disponibilizado um campo de texto livre para que os usuários pudessem adicionar sugestões e feedbacks relacionados a ferramenta. Houve quatro respostas neste campo com sugestões de melhorias para a ferramenta. Entre eles sugestão para adicionar configuração para alerta os usuários para quando algum check falhar ou a latência foi acima de um limite definido pelo usuário, esse ponto já é algo que está na lista de funcionalidades a serem adicionadas na ferramenta. Outra sugestão foi para adicionar a estrutura das mensagens de requisição e resposta no momento de criação do check e o usuário precisaria apenas informar quais devem ser os valores de cada campo. Atualmente, os campos de requisição e resposta dos

checks são campos livres, o que realmente pode impactar a experiência do usuário, pois ele precisa procurar a definição dessas mensagens para conseguir montar a request e o response.

5. Conclusão

O uso de gRPC para construção de APIs vem se tornando cada vez mais popular o que torna necessário também ferramentas que deem suporte a esse tipo de tecnologia. Dito isso, neste trabalho foi apresentado o desenvolvimento de uma ferramenta de monitoramento de APIs gRPC, o HealthCheckAPI. Essa ferramenta tem como foco principal além de fazer monitoramento e observabilidade das API, fornecer suporte e uma boa experiência para os seus usuários a fim de fomentar e tornar cada vez mais fácil criar e manter APIs usando esse tipo de tecnologia. Como trabalhos futuros, pretende-se implementar funcionalidades como as que foram sugeridas no questionário disponibilizado para os usuários de teste: adicionar configuração de alertas e tornar mais fácil a configuração das mensagens de requisição e resposta dos checks.

6. Agradecimentos

Agradeço primeiramente a Deus por todas as conquistas e bênçãos que concedidas durante a minha vida, entre elas a realização deste trabalho. Agradeço imensamente a minha tia Heliane por ter me criado desde pequena mesmo diante de todas as dificuldades. Agradeço ao meu noivo por todo apoio incondicional durante a realização deste trabalho e em vários outros momentos da minha vida. Agradeço ao meu filho João, que ainda no meu ventre, já é um grande motivador na minha vida para que eu me torne uma pessoa e uma profissional melhor. Agradeço a minha mãe por todo carinho e incentivo. Agradeço ao meu orientador João Arthur por ter me ajudado a tornar esse sonho realidade. Agradeço a minha grande amiga Aylla por ter sido meu pilar durante toda a graduação, juntas vivemos muitas alegrias e tristezas, mas sempre juntas apoiando uma à outra. Agradeço também a todos os professores que tive no IFPB, em especial o professor Ruan e Erick que sempre confiaram em mim e que me inspiram até hoje. Por fim, agradeço também a todos os meus demais familiares, professores e amigos que também fizeram parte da minha trajetória. Muito obrigada!

7. REFERÊNCIAS

[1] GRPC AUTHORS. **The state of gRPC in the browser**. [S. l.], 8 jan. 2019. Disponível em: https://grpc.io/blog/state-of-grpc-web/. Acesso em: 25 mar. 2022.

[2] INDRASIRI, Kasun; KURUPPU, Danesh. **gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes**. " O'Reilly Media, Inc.", 2020.

[3] GRPC AUTHORS. **About gRPC**: Who is using gRPC and why. [S. l.], 8 jan. 2019. Disponível em: https://grpc.io/about/#the-story-behind-grpc. Acesso em: 24 mar. 2022.

[4] HEGER, Christoph et al. Application performance management: State of the art and challenges for the future. In: **Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering**. 2017. p. 429-432.

[5] APPLICATION Performance Monitoring Reviews and Ratings. [S. l.]: Gartner Peer Insights, 2022. Disponível em: https://www.gartner.com/reviews/market/application-performance-monitoring. Acesso em: 24 mar. 2022.

[6] CALCOTE, Lee; BUTCHER, Zack. Istio: **Up and running: Using a service mesh to connect, secure, control, and observe**. O'Reilly Media, 2019.

[7] HOLOPAINEN, Matti. Monitoring Container Environment with Prometheus and Grafana. 2021.

[8] COMO MONITORAR sua API: Cloud Endpoints com gRPC. [S. l.], 22 mar. 2022. Disponível em: https://cloud.google.com/endpoints/docs/grpc/monitoring-your-api. Acesso em: 29 mar. 2022.

[9] CHECKLY documentation. [S. l.], 9 mar. 2022. Disponível em: https://www.checklyhq.com/docs. Acesso em: 29 mar. 2022.

[10] WHY Assertible?. [S. l.], 13 jan. 2016. Disponível em: https://assertible.com/about. Acesso em: 29 mar. 2022.