PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Armazenamento de Dados Seguro com Zero-Knowledge para Serviços de Nuvem Pública

Wallace Silva de Freitas

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO – CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Sistemas de Informação

Orientador: Sérgio Colcher
Coorientador: Anderson Oliveira da Silva

Rio de Janeiro, Junho de 2024

Wallace Silva de Freitas

Armazenamento de Dados Seguro com Zero-Knowledge para Serviços de Nuvem Pública

Relatório de Projeto Final, apresentado ao programa de Graduação em Sistemas de Informação da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação

Orientador: Sérgio Colcher
Coorientador: Anderson Oliveira da Silva

Rio de Janeiro
Junho de 2024

Agradecimentos

Em primeiro lugar, a família e amigos, por me sustentar nas horas mais difíceis e me permitir ultrapassar todos os obstáculos encontrados.
Às pessoas que estiveram me apoiando e incentivando ao longo da minha graduação, e a todos aqueles que contribuíram, de alguma forma, para a realização deste trabalho. A minha mais profunda gratidão aos meus orientadores, pela extrema paciência, pelas valiosas sugestões e pelo conhecimento compartilhado.

Resumo

FREITAS, Wallace Silva de. COLCHER, Sérgio. SILVA, Anderson Oliveira da. Armazenamento de Dados Seguro com Zero-Knowledge para Serviços de Nuvem Pública. Rio de Janeiro, 2021. 43 p. Relatório de Projeto Final de Graduação em Sistemas de Informação – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Desenvolver aplicações pode exigir que os desenvolvedores de software tenham conhecimento razoável sobre infraestrutura e práticas de segurança. Neste trabalho, exploramos o uso da Infraestrutura como Código (IaC) para simplificar o gerenciamento de infraestrutura em nuvem. Propomos analisar os benefícios, dificuldades e limitações dessa abordagem, sobretudo do ponto de vista do desenvolvedor. Ao adotar a IaC, o processo de desenvolvimento e manutenção de aplicações torna-se mais ágil e eficiente, permitindo que os desenvolvedores concentrem-se na lógica da aplicação em vez de se preocuparem com detalhes de infraestrutura.

Palavras-chave

Docker, containers, criptografia simétrica e assimétrica, prova de conhecimento zero, computação em nuvem, discos virtuais

Abstract

FREITAS, Wallace Silva de. COLCHER, Sérgio. SILVA, Anderson Oliveira da. Zero-Knowledge Secure Data Storage for Public Cloud Services. Rio de Janeiro, 2021. 43 p. Relatório de Projeto Final de Graduação em Sistemas de Informação – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Developing applications may require software developers to have a reasonable understanding of infrastructure and security practices. In this paper, we explore the use of Infrastructure as Code (IaC) to simplify cloud infrastructure management. We propose to analyze the benefits, difficulties, and limitations of this approach, especially from the developer's point of view. By adopting IaC, the application development and maintenance process becomes more agile and efficient, allowing developers to focus on application logic rather than worrying about infrastructure details.

Keywords

Docker, containers, symmetric and asymmetric encryption, zero-knowledge proof, cloud computing, virtual disks

Sumário

1. Introdução

1.1. Contexto e Motivação

Em um universo cada vez mais cibernético e inserido da contextualização do “Modus Viventi” , da era digital, a privacidade do usuário e a segurança dos dados se tornaram questões de suma importância.

Os sistemas de “prova de conhecimento nulo” (zero-knowledge proof systems), figuram com relevância entre as aplicações criptográficas modernas, estando presentes na base de diversos protocolos mais complexos, como esquemas de identificação, protocolos de eleição eletrônica e protocolos de pagamento bancários por intermédio de moedas eletrônicas. (GOLDREICH ET AL, P.690,1991).

Com o advento das aplicações da condição e o pré-requisito do uso de armazenamento em nuvem, a necessidade de proteger os dados dos usuários clientes armazenados tornou-se ainda mais crítica. Deve-se esclarecer que as aplicações que armazenam arquivos de forma segura devem ser cifradas com chaves assimétricas [...] cada vez mais reservam-se rotinas e cuidados a fim de se ater ao Quesito : Segurança. (ANDERSON ET AL,2015)

Ainda segundo Anderson et al(2015), evidencia-se, que a chave pública usada para cifrar os arquivos garante que apenas o proprietário da chave possa acessar os dados, proporcionando um alto nível de segurança.

No entanto, muitos desenvolvedores encontram dificuldades ao tentar criar essas aplicações, além disso, a utilização de linguagens de programação modernas e eficientes, como Golang e Python, permitem um desempenho otimizado.

De acordo com Anderson et al. (2015), esclarecem que o Docker é uma tecnologia de virtualização de contêineres. Em poucas palavras, é como uma máquina virtual [VM] muito leve. É uma plataforma de código aberto que automatiza a implantação, o dimensionamento e a execução de aplicativos em contêineres.

> Ainda, segundo Anderson et al. (2015), que os containers Docker permitem que os desenvolvedores empacotem um aplicativo com todas as suas dependências em uma unidade padronizada para desenvolvimento de software. Essa unidade, conhecida como contêiner, é leve e pode ser facilmente transferida entre diferentes ambientes de computação. Isso significa que os desenvolvedores podem ter certeza de que o software funcionará exatamente da mesma maneira, independentemente do ambiente em que for implantado.

Por serem portadores de uma natureza leve e portátil, os containers Docker os tornam ideais para arquiteturas baseadas em microsserviços, onde os aplicativos são divididos em componentes menores e independentes.

Sua principal função é simplificar o processo de entrega de software permitindo que os desenvolvedores trabalhem em ambientes padronizados usando contêineres locais que fornecem aplicações e serviços. Isso elimina a típica frase “funciona na minha máquina”. Portanto com o uso de contêineres, se funciona em um lugar, funciona em todos os lugares.

É oportuno observar a versatilidade do Docker com outras tecnologias de virtualização de contêineres, disponíveis, contudo em relação a sua complexidade tem a tendência em acrescer quando esses arquivos são armazenados em nuvens públicas.

1.2. Objetivos

O objetivo principal do projeto é apresentar uma solução para o armazenamento seguro de arquivos em nuvens públicas.

O objetivo específico é demonstrar para a comunidade de desenvolvedores e entusiastas da segurança de dados, as possibilidades de se criar soluções de armazenamento seguro em nuvens públicas de forma eficiente e fácil manutenção, sendo sua interface amigável com outras aplicações.

A solução proposta é a criação de um plugin para o projeto do aluno Bernardo Rezende, intitulado Cofre de Senhas, que permite a criação de pastas seguras nas plataformas de armazenamento em nuvem e local.

A aplicação escrita em Golang para o Backend, utiliza chaves assimétricas para assegurar a integridade, autenticidade e sigilo dos arquivos armazenados na pasta segura.

1.3. Organização da Monografia

Esta monografia se subdivide em 4 capítulos. O Capítulo 2 trata da fundamentação teórica do projeto e apresenta conceitos básicos. O Capítulo 3 aborda o desenvolvimento do projeto e analisa seus aspectos em geral. Finalmente, o Capítulo 4 apresenta a conclusão, recapitulando o que foi apresentado ao longo dos capítulos anteriores e comentando as limitações do projeto e trabalhos futuros.

2. Fundamentação teórica

Ao longo deste capítulo, serão abordados os conceitos de Infraestrutura como código (IaC), DevOps, Criptografia, Hash, Prova de Conhecimento Zero, OAuth 2.0 e a relação desses itens e o que já existe nesse contexto hoje em dia. Com isso, o objetivo será prover a fundamentação teórica necessária para demonstrar a relevância desse trabalho e desenvolvê-lo de forma adequada.

2.1. Infraestrutura como código (IaC) e DevOps

A transformação digital tem revolucionado a maneira como desenvolvemos, implantamos e gerenciamos sistemas de software. Nesse contexto, dois conceitos fundamentais ganham destaque: Infraestrutura como Código (IaC) e DevOps. Vamos explorar esses conceitos e entender como eles estão moldando o cenário tecnológico atual.

Infraestrutura como Código (IaC)

A Infraestrutura como Código é uma abordagem que permite tratar a configuração e provisionamento de recursos de infraestrutura (como servidores, redes e bancos de dados) da mesma forma que tratamos o código-fonte de nossas aplicações. Aqui estão os principais pontos: **(ARTAC ET AL,2017)**

Automação e Consistência: Com IaC, descrevemos nossa infraestrutura usando linguagens de programação ou formatos específicos (como YAML ou JSON). Essa descrição é versionada e pode ser compartilhada entre equipes.

A automação resultante garante que a infraestrutura seja criada e mantida de maneira consistente.

Reprodutibilidade: Ao usar IaC, podemos criar ambientes idênticos em diferentes estágios do ciclo de vida de uma aplicação (desenvolvimento, homologação, produção). Isso elimina problemas causados por diferenças manuais na configuração.

Aceitação: Ferramentas populares de IaC incluem o Terraform, Ansible, CloudFormation, Azure Resource Manager e Docker que será usada nesse projeto.

DevOps

DevOps é uma cultura e conjunto de práticas que visa a integração contínua entre desenvolvimento e operações. Aqui estão os principais aspectos do DevOps: **(ARTAC ET AL, 2017)**

Colaboração: DevOps promove a colaboração entre desenvolvedores, administradores de sistemas e equipes de operações. Essa colaboração é essencial para acelerar o ciclo de entrega e melhorar a qualidade do software.

Automação: A automação é um pilar do DevOps. Isso inclui automação de compilação, implantação, testes e monitoramento. Ferramentas como Jenkins, GitLab CI/CD e CircleCI são amplamente usadas.

Entrega Contínua: O objetivo é entregar software de alta qualidade de forma rápida e confiável. A entrega contínua envolve integração contínua, testes automatizados e implantação automatizada.

Usos e Benefícios de IaC e DevOps

Alguns dos usos e benefícios de integrar IaC e DevOps são: **(ARTAC ET AL, 2017)**

Agilidade : IaC e DevOps permitem que as equipes respondam rapidamente às mudanças de requisitos e às demandas do mercado.

Escalabilidade: Com IaC, dimensionar recursos é tão simples quanto alterar um arquivo de configuração.

Redução de Erros: A automação reduz erros humanos e aumenta a confiabilidade.

Eficiência: DevOps otimiza o fluxo de trabalho, reduzindo o tempo de desenvolvimento e implantação.

A combinação de IaC e DevOps está transformando a maneira como construímos e mantemos sistemas, tornando-os mais ágeis, escaláveis e confiáveis. À medida que avançamos, esses conceitos continuarão a moldar o futuro da tecnologia.

2.1.1. IaC Declarativa (.DockerFile)

A Infraestrutura como Código (IaC) é uma abordagem que permite automatizar a criação, configuração e gerenciamento de recursos de infraestrutura por meio de código. Nesse contexto, o uso de uma abordagem declarativa é fundamental para descrever o estado desejado da infraestrutura, em vez de especificar passos detalhados para alcançá-lo. Um dos elementos-chave da IaC é o Dockerfile, que desempenha um papel crucial na criação e implantação de contêineres. **(ARTAC ET AL, 2017)**

Conceitos da Infraestrutura como Código Declarativa

Declaratividade : A abordagem declarativa se concentra no "o quê" em vez do "como". Em vez de instruções detalhadas, descrevemos o estado desejado da infraestrutura. Isso permite que as ferramentas de IaC determinem automaticamente como alcançar esse estado.

Reprodutibilidade: Com a IaC declarativa, podemos criar e recriar ambientes consistentes. O código descreve exatamente o que deve ser provisionado, evitando discrepâncias entre ambientes de desenvolvimento, teste e produção.

Versionamento: O código da infraestrutura pode ser versionado e controlado usando sistemas como Git. Isso facilita o rastreamento de alterações, colaboração e reversão.

Dockerfile e Contêineres

O Dockerfile é um arquivo de texto que define como construir uma imagem de contêiner. Aqui estão os principais conceitos relacionados: **(ANDERSON ET AL,2015)**

Imagem Docker: Uma imagem é uma representação estática de um ambiente de execução. O Dockerfile descreve os passos para criar essa imagem, incluindo a instalação de dependências, configurações e aplicativos.

Contêiner: Um contêiner é uma instância em execução de uma imagem. Ele é isolado, leve e portátil. Os contêineres permitem empacotar aplicativos e suas dependências, garantindo consistência em diferentes ambientes.

Uso Prático

Dockerfile: No Dockerfile, descrevemos os comandos para construir uma imagem. Por exemplo, podemos especificar a base da imagem, copiar arquivos, instalar pacotes e definir variáveis de ambiente.

Comandos Docker: Usamos comandos como `docker build` para criar uma imagem a partir do Dockerfile e `docker run` para iniciar um contêiner com base nessa imagem.

Orquestração: Ferramentas como o Docker Compose e o Kubernetes permitem gerenciar vários contêineres como uma aplicação completa. A IaC pode descrever a configuração dessas ferramentas.

Benefícios

Consistência: O uso de IaC e Dockerfile garante que todos os ambientes sejam idênticos.

Agilidade: A automação acelera a implantação e reduz erros manuais.
Escalabilidade: A replicação de contêineres é simples e escalável.

IaC declarativa, combinada com o Dockerfile, oferece uma maneira poderosa de criar, gerenciar e escalar infraestruturas de forma eficiente e confiável. Ao adotar essas práticas, os desenvolvedores podem focar mais na lógica de aplicação e menos na configuração manual de servidores e ambientes.

2.1.2. IaC Imperativa (.Yaml)

A Infraestrutura como Código (IaC) revolucionou a forma como provisionamos e gerenciamos ambientes de TI, substituindo processos manuais por scripts automatizados. Entre as abordagens de IaC, a Infraestrutura como Código Imperativa destaca-se por oferecer controle granular sobre a criação e modificação de recursos, utilizando linguagens expressivas como YAML no contexto do Docker. **(BEN-KIKI ET AL, 2009)**

Conceitos Fundamentais:

A IaC Imperativa se diferencia pela descrição sequencial das etapas necessárias para alcançar o estado desejado da infraestrutura. Da mesma forma, a IaC Imperativa define cada comando a ser executado para provisionar recursos, como criar servidores, configurar redes e instalar softwares.

Vantagens da IaC Imperativa:

Controle Detalhado: Permite manipular recursos com alta granularidade, ideal para ambientes complexos com requisitos específicos.

Flexibilidade: Facilita adaptações e personalizações de acordo com as necessidades do projeto.

Visibilidade Aprimorada: Documenta o processo de provisionamento de forma clara e rastreável, facilitando a auditoria e o troubleshooting.

Repetibilidade:Garante a consistência na criação de ambientes, eliminando erros manuais e padronizando configurações.

YAML no Docker: A Linguagem Ideal para IaC Imperativa

O YAML (YAML Ain't Markup Language) se tornou a linguagem preferida para definir arquivos de configuração do Docker, devido à sua simplicidade, legibilidade e expressividade. Sua estrutura em hierarquia e uso de chaves e valores facilitam a descrição de recursos complexos de forma organizada e intuitiva. **(BEN-KIKI ET AL, 2009)**

Exemplos de Uso da IaC Imperativa com YAML no Docker

Criação de Containers: Defina as configurações de imagem, volumes, portas, variáveis de ambiente e outros parâmetros para criar containers personalizados.

Gerenciamento de Redes: Configure redes virtuais, pontes, firewalls e regras de roteamento para interconectar containers e serviços.

Provisionamento de Serviços: Crie e configure serviços como bancos de dados, sistemas de mensagens e ferramentas de monitoramento.

Orquestração de Containers: Utilize ferramentas como Docker Compose para orquestrar a criação, inicialização e interconexão de containers em um ambiente complexo.

A IaC Imperativa com YAML no Docker oferece uma poderosa ferramenta para gerenciar infraestrutura de forma eficiente e granular. Sua flexibilidade, controle e visibilidade a tornam ideal para projetos que exigem personalização e adaptabilidade. Ao dominar essa abordagem, os profissionais de TI podem automatizar tarefas complexas, reduzir erros e provisionar ambientes de forma rápida e confiável.

2.1.3. Docker, containers e suas vantagens

A virtualização de software se tornou uma ferramenta essencial para desenvolvedores e operadores de TI, possibilitando a execução de múltiplos ambientes isolados em um único sistema físico. Entre as diversas tecnologias de virtualização, o Docker e seus containers se destacam por sua leveza, eficiência e portabilidade, revolucionando a maneira como os aplicativos são criados, implantados e gerenciados. **(ANDERSON ET AL,2015)**

Conceitos Fundamentais

Containers: São unidades de software autossuficientes que encapsulam um aplicativo, seu código, bibliotecas, dependências e configurações, isolando-o do ambiente externo. Imagine um container como um contêiner físico que transporta tudo o que um aplicativo precisa para funcionar, sem a necessidade de modificar o sistema operacional da máquina host.

Imagens Docker: São modelos reutilizáveis que servem como base para a criação de containers. Elas contêm as instruções e os recursos necessários para executar um aplicativo específico, incluindo o sistema operacional, bibliotecas e configurações. Pense em uma imagem Docker como um plano detalhado para a construção de um container.

Docker Engine: É a plataforma central do Docker, responsável pela criação, execução e gerenciamento de containers. Ele funciona como um orquestrador, traduzindo as instruções das imagens Docker em containers ativos e gerenciando seus recursos.

A grande vantagem deste projeto é a sua portabilidade e facilidade de configuração, por intermédio do Docker. Essa plataforma de instituir o acesso e uma permissão a outros desenvolvedores que tenham a capacidade de dar prosseguimento ao projeto ou configurá-lo em um ambiente de desenvolvimento com facilidade. Também é considerado uma chave facilitadora quando se trata de configuração, manutenção e disponibilidade do sistema, bem como a colaboração de outros desenvolvedores.

Para isso, é implementado um exemplo de aplicação que demonstra o uso do plugin para a criação de discos cifrados. Esta aplicação serve como um exemplo prático da funcionalidade e eficiência do plugin proposto.

Espera-se que seja de grande utilidade para a comunidade de desenvolvedores e entusiastas da segurança de dados, fornecendo uma ferramenta valiosa para o armazenamento seguro de arquivos em nuvem., sendo as características inúmeras vantagens são, quando na ocasião de sua implementação uma característica itemizada é a

Portabilidade entre diferentes plataformas -Isolamento de recursos. -Eficiência e leveza comparado a máquinas virtuais tradicionais. –Facilidade de integração e entrega contínua.

Agilidade: O Docker permite que os desenvolvedores trabalhem com agilidade, pois facilita a criação, teste e remoção de aplicações de forma isolada.
Escalabilidade: Com o Docker, é possível escalar facilmente suas aplicações, pois ele permite que você crie e gerencie múltiplos contêineres de forma eficiente.

Portabilidade: Os containers Docker podem ser movidos facilmente entre diferentes ambientes, garantindo que a aplicação funcione da mesma maneira em todos eles.

Isolamento: Cada container Docker funciona de forma isolada, o que significa que eles não interferem uns com os outros. Isso é especialmente útil quando você está trabalhando com micro serviços.

Controle do processo de DevOps: O Docker permite um controle mais efetivo de todo o processo de DevOps, facilitando a integração contínua e a entrega contínua.

Redução de custos: O Docker pode ajudar a reduzir os custos de licenças de software, pois utiliza menos instâncias do sistema operacional para suas cargas de trabalho.

Alta disponibilidade: O Docker permite que você crie aplicações altamente disponíveis, pois facilita a replicação e o balanceamento de carga entre os contêineres.

Deployment automatizado: O Docker facilita o deployment automatizado de aplicações, tornando o processo mais eficiente e menos propenso a erros.
Ambientes Semelhantes: Uma vez que sua aplicação seja transformada em uma imagem Docker, ela pode ser instanciada como contêiner em qualquer ambiente que desejar. Isso significa que poderá utilizar sua aplicação no notebook do desenvolvedor da mesma forma que seria executada no servidor de produção.

Aplicação como Pacote Completo: Utilizando as imagens Docker é possível empacotar toda sua aplicação e dependências, facilitando a distribuição, pois não será mais necessário enviar uma extensa documentação explicando como configurar a infraestrutura necessária para permitir a execução.

Atualização Simplificada: A estrutura de camadas do Docker viabiliza que, em caso de mudança, apenas a parte modificada seja transferida e assim o ambiente pode ser alterado de forma mais rápida e simples.
Leveza e Manutenção Simplificada: Com o Docker, o desenvolvedor ganha em portabilidade e leveza, além de uma manutenção mais simplificada, pois fica sobre o SO do servidor.

Desenvolvimento Ágil: Ele facilita e simplifica a aplicação da metodologia DevOps e facilita o desenvolvimento ágil.

Portanto indica-se por estas razões citadas anteriormente, a utilização e a implementação que o uso do Docker facilita todos os passos e rotinas de execução.

O primeiro ponto a considerar quando se pensa em projetos, quer sejam individuais ou coletivos em uma aplicação baseada em classes é, sem dúvida, o caráter a qualquer prática e intervenção.
A solução criada, Docker opera em uma arquitetura cliente-servidor.

Segundo Chung et al (2016,p.52), o cliente Docker se comunica com o daemon Docker, que é responsável por construir, executar e distribuir os containers Docker. O cliente e o daemon Docker podem operar no mesmo sistema, ou você pode conectar um cliente Docker a um daemon Docker remoto. O daemon Docker (dockerd) ou servidor é responsável por todas as ações relacionadas aos contêineres. O daemon recebe os comandos do cliente Docker por meio da CLI ou da API REST. O cliente Docker pode estar no mesmo host que um daemon ou presente em qualquer outro host.

Conforme Chung et al(2016,p.53), doravante os containers Docker são executados em cima do sistema operacional host e compartilham o mesmo kernel do sistema operacional.

Cada container Docker funciona como um processo isolado no sistema operacional host.(CHUNG, ET AL, 2016,P.53)

Usabilidade-Arquitetura-Estrutura

Ainda nas definições de Chung et al(2016, p.53), sua estrutura se divide nos principais componentes:

Imagens Docker: São a base dos containers Docker. Uma imagem Docker é um modelo imutável que contém um instantâneo de um aplicativo e todas as dependências necessárias para executar esse aplicativo. Quando um contêiner Docker é iniciado, ele é criado a partir de uma imagem Docker.Os containers Docker podem ser iniciados, parados, movidos e excluídos. Cada contêiner é isolado e seguro.

Dockerfiles: São scripts que contém uma série de instruções para construir uma imagem Docker. Cada instrução em um Dockerfile cria uma nova camada na imagem. Isso inclui instruções para instalar dependências, copiar arquivos de origem, configurar variáveis de ambiente e assim por diante.

Docker Compose: É uma ferramenta que permite definir e gerenciar aplicativos Docker de vários contêineres. Com o Docker Compose, você pode usar um arquivo YAML para definir os serviços do seu aplicativo e as dependências entre eles. Em seguida, com um único comando, você pode criar e iniciar todos os serviços a partir de suas configurações.

Este tópico em assunto contempla as especificidades

2.2. Criptografia e Hash

Em um mundo cada vez mais digital, a segurança da informação se torna um tema crucial. Com o crescente volume de dados armazenados e transmitidos eletronicamente, proteger esses dados contra acessos não autorizados, modificações ou interceptações é fundamental para garantir a privacidade, confidencialidade e integridade. A criptografia e as funções hash são ferramentas essenciais para essa tarefa, oferecendo diferentes níveis de proteção para diversos tipos de dados.

Criptografia: Transformando Dados em Segredos

A criptografia é a técnica de transformar dados legíveis em um formato indecifrável, conhecido como texto cifrado, utilizando algoritmos matemáticos complexos. Essa transformação torna os dados ininteligíveis para qualquer pessoa que não possua a chave secreta para decifrá-los. A criptografia garante a confidencialidade dos dados, protegendo-os contra espionagem e roubo, e é utilizada em diversas áreas, como:

Segurança da Comunicação: Protege a comunicação em redes de dados contra interceptação e escuta.

Armazenamento Seguro: Criptografa dados armazenados em discos rígidos, pendrives e serviços de armazenamento em nuvem, protegendo-os contra acessos não autorizados.

Transações Eletrônicas: Garante a segurança de transações financeiras online, como compras com cartão de crédito e transferências bancárias.

Funções Hash: Verificando a Integridade dos Dado

As funções hash, também conhecidas como funções de resumo de mensagem, são algoritmos matemáticos que transformam qualquer tipo de dado em um valor fixo de tamanho, chamado de hash ou digestão (digest). Esse valor único representa o conteúdo original dos dados e possui a propriedade crucial de ser extremamente sensível a qualquer alteração nos dados. Ou seja, se os dados forem modificados, por menor que seja a alteração, o hash será completamente diferente. As funções hash são utilizadas para:

Verificação de Integridade: Garantem que os dados não foram corrompidos durante a transmissão ou armazenamento, detectando qualquer modificação não autorizada.

Detecção de Duplicatas: Permitem identificar arquivos duplicados, mesmo que tenham sido renomeados ou modificados de forma leve.
Validação de Senhas: Armazenam hashes de senhas em vez das senhas em si, protegendo-as contra roubo e revelação.

Diferenças Essenciais entre Criptografia e Hash

A criptografia visa ocultar o conteúdo dos dados, enquanto as funções hash visam verificar a integridade dos dados. Suas características são:

Reversibilidade: A criptografia é um processo reversível, com a chave secreta permitindo a recuperação dos dados originais. As funções hash são irreversíveis, não sendo possível gerar os dados originais a partir do hash.

Tamanho do Resultado: A criptografia gera um texto cifrado com tamanho variável, enquanto as funções hash geram um valor fixo de tamanho, independente do tamanho dos dados originais.

Aplicações Conjuntas de Criptografia e Hash

A criptografia e as funções hash podem ser utilizadas em conjunto para oferecer um nível ainda maior de segurança. Por exemplo, um arquivo pode ser criptografado e, em seguida, ter seu hash calculado. Ao receber o arquivo criptografado, o destinatário pode decifrá-lo e calcular o hash novamente. Se o hash calculado coincidir com o hash original, isso garante que o arquivo não foi corrompido durante a transmissão ou armazenamento.

A criptografia e as funções hash são ferramentas essenciais para garantir a segurança da informação na era digital. A criptografia protege a confidencialidade dos dados, enquanto as funções hash garantem sua integridade. Ao compreender e utilizar essas ferramentas de forma adequada, é possível proteger dados sensíveis contra diversos tipos de ameaças, assegurando a privacidade, confidencialidade e confiabilidade da informação.

Pilares da Segurança da Informação

Confidencialidade: Protegendo Segredos

A confidencialidade garante que a informação seja acessível apenas a indivíduos, entidades ou processos autorizados. Ela visa impedir que dados confidenciais, como informações pessoais, financeiras ou estratégicas, sejam divulgados ou acessados por pessoas não autorizadas. Isso é alcançado através de medidas como controle de acesso, criptografia e segregação de dados.

Integridade: Assegurando a Veracidade da Informação

A Integridade garante que a informação esteja completa, precisa e consistente, e que não tenha sido corrompida ou modificada de forma não autorizada.

Isso significa que os dados devem refletir a realidade de forma autêntica e confiável. A proteção da integridade é alcançada através de medidas como controle de alterações, autenticação de dados e backups.

Disponibilidade: Garantindo o Acesso à Informação

A Disponibilidade garante que a informação esteja acessível aos usuários autorizados no momento em que precisam. Isso significa que os sistemas e dados devem estar funcionando corretamente e sem interrupções. A proteção da disponibilidade é alcançada através de medidas como redundância de infraestrutura, planos de recuperação de desastres e proteção contra ataques cibernéticos.

Autenticidade: Validando Identidades e Origens

A Autenticidade garante que a origem da informação seja verificável e que os usuários ou sistemas que a acessam sejam quem dizem ser. Isso significa que a informação deve ter uma fonte confiável e que os usuários sejam autenticados antes de acessar os dados. A proteção da autenticidade é alcançada através de medidas como autenticação de usuários, assinatura digital e verificação de origem da informação.

Ampliando a Segurança da Informação

Embora os Quatro Pilares sejam a base fundamental da segurança da informação, outros elementos também desempenham um papel crucial na proteção de dados e sistemas. Entre eles, podemos destacar:

Não Repúdio: Garante que um indivíduo ou sistema não possa negar ter realizado uma ação ou enviando uma mensagem.

Privacidade: Protege a privacidade dos indivíduos, garantindo que seus dados pessoais sejam coletados, utilizados e armazenados de forma ética e responsável.

Utilidade: Garante que a informação seja útil e relevante para o propósito a que se destina.

Confiabilidade: Garante que a informação seja confiável e de boa qualidade.

2.2.1. Criptografia Simétrica

A criptografia surge como ferramenta essencial para proteger dados confidenciais, como senhas, informações financeiras e comunicações privadas. Entre as diversas técnicas de criptografia, a criptografia simétrica se destaca por sua simplicidade, eficiência e ampla aplicabilidade.

Funcionamento da Criptografia Simétrica

A criptografia simétrica, também conhecida como criptografia de chave secreta, utiliza uma única chave secreta para criptografar e descriptografar dados.
Essa chave deve ser compartilhada entre o remetente e o destinatário da mensagem, e precisa ser mantida em sigilo absoluto para garantir a segurança da comunicação.

No processo de criptografia, o remetente utiliza a chave secreta para transformar o texto original (chamado de plaintext) em um texto cifrado (ciphertext), que se torna ilegível para qualquer pessoa que não possua a chave. Ao receber o ciphertext, o destinatário utiliza a mesma chave secreta para reverter o processo, transformando o ciphertext de volta no plaintext original.

Tamanho do Texto Cifrado

A segurança da criptografia simétrica reside na confidencialidade da chave secreta. Se um invasor conseguir obter a chave, ele poderá ler todas as mensagens criptografadas com ela. Por isso, é fundamental que a chave seja mantida em sigilo e protegida contra acessos não autorizados.

O tamanho do texto cifrado geralmente é maior que o tamanho do plaintext original. Isso ocorre porque o processo de criptografia adiciona informações redundantes ao texto para torná-lo ilegível. No entanto, o aumento do tamanho é geralmente pequeno e imperceptível para a maioria das aplicações.

Custo Computacional e Eficiência

A criptografia simétrica é considerada uma técnica eficiente do ponto de vista computacional. Os algoritmos utilizados para criptografar e descriptografar dados geralmente são rápidos e podem ser facilmente implementados em hardware e software.

A criptografia simétrica se diferencia da criptografia assimétrica, que utiliza um par de chaves: uma chave pública para criptografar e uma chave privada para descriptografar. A principal vantagem da criptografia simétrica é sua simplicidade e eficiência, enquanto a criptografia assimétrica oferece maior segurança para a distribuição da chave pública, que pode ser compartilhada livremente.

2.2.2. Criptografia Assimétrica

Ao contrário da criptografia simétrica, que utiliza uma única chave secreta para criptografar e descriptografar dados, a criptografia assimétrica utiliza um par de chaves matematicamente relacionadas: uma chave pública e uma chave privada. A chave pública pode ser compartilhada livremente com qualquer pessoa, enquanto a chave privada deve ser mantida em sigilo absoluto pelo proprietário.

No processo de criptografia, o remetente utiliza a chave pública do destinatário para transformar o texto original (plaintext) em um texto cifrado (ciphertext). Como a chave pública é de conhecimento público, qualquer pessoa pode criptografar uma mensagem para o destinatário. No entanto, apenas o destinatário, que possui a chave privada correspondente, pode descriptografar a mensagem e recuperar o plaintext original.

Segurança e Vantagens

A segurança da criptografia assimétrica reside na dificuldade computacional de obter a chave privada a partir da chave pública. Mesmo que a chave pública seja de conhecimento público, é praticamente impossível para um invasor, utilizando métodos matemáticos conhecidos, derivar a chave privada correspondente. Isso garante que apenas o proprietário legítimo da chave privada possa descriptografar as mensagens.

A criptografia assimétrica oferece diversas vantagens em relação à criptografia simétrica:

Distribuição de Chaves Segura: A chave pública pode ser compartilhada livremente, sem a necessidade de um canal de comunicação seguro para a troca de chaves. Isso facilita a comunicação segura entre pessoas que nunca se conheceram antes.

Autenticação de Identidade: A chave pública pode ser utilizada para assinar digitalmente mensagens e documentos, garantindo a autenticidade da origem e a integridade da informação.

Não Repúdio: A assinatura digital garante que o signatário não possa negar ter assinado uma mensagem ou documento.

Custo Computacional

A criptografia assimétrica é geralmente mais lenta do que a criptografia simétrica, devido à complexidade dos algoritmos matemáticos utilizados. No entanto, essa diferença de desempenho geralmente não é um problema para a maioria das aplicações, pois a criptografia é realizada apenas algumas vezes, como no início de uma comunicação segura ou na assinatura de um documento.

A criptografia assimétrica é uma ferramenta poderosa para garantir a segurança da informação em um mundo digital cada vez mais conectado. Sua capacidade de distribuir chaves de forma segura, autenticar identidades e proteger a confidencialidade da comunicação a torna essencial para diversas aplicações críticas. Apesar do custo computacional ser um pouco mais alto do que na criptografia simétrica, essa desvantagem é compensada pelos seus benefícios em termos de segurança e flexibilidade.

2.2.3. Hash Criptográfico

O SHA-256 (Secure Hash Algorithm 256) é uma função hash criptográfica desenvolvida pela Agência Nacional de Padrões e Tecnologia dos Estados Unidos (NIST). Ela faz parte da família SHA-2 de funções hash, reconhecida por sua robustez e segurança contra ataques criptográficos.

Funcionamento do SHA-256

O SHA-256 funciona como uma "impressão digital" para dados digitais. Ele recebe um conjunto de dados de entrada, de qualquer tamanho, e gera um valor de hash fixo de 256 bits, também conhecido como digestão ou resumo do hash. Esse valor de hash é único para cada conjunto de dados de entrada, mesmo que sejam modificados apenas ligeiramente.

Propriedades Essenciais do SHA-256

Colisão: É extremamente improvável que dois conjuntos de dados diferentes gerem o mesmo valor de hash SHA-256.

Pré Imagem: É computacionalmente inviável encontrar um conjunto de dados que gere um valor de hash SHA-256 específico.

Segurança contra segunda pré-imagem: É computacionalmente inviável encontrar um segundo conjunto de dados que gere o mesmo valor de hash SHA-256 de um conjunto de dados original, mesmo que o conjunto original seja conhecido.

O SHA-256 é uma função hash criptográfica robusta e confiável que desempenha um papel crucial na segurança da informação na era digital. Sua capacidade de garantir a integridade, autenticidade e confiabilidade dos dados o torna uma ferramenta essencial para proteger informações confidenciais, prevenir fraudes e garantir a confiabilidade de transações online. À medida que a quantidade de dados digitais cresce exponencialmente, o SHA-256 continuará a ser uma ferramenta fundamental para proteger a segurança da informação em um mundo cada vez mais conectado.

2.2.4. Assinatura Digital

A assinatura digital com chave pública e privada utiliza os princípios da criptografia assimétrica para criar uma assinatura eletrônica que vincula um documento digital ao seu signatário. O processo envolve as seguintes etapas:

Cálculo do Hash: O documento digital é transformado em um valor de hash único utilizando uma função hash criptográfica, como SHA-256.

Criptografia do Hash: O valor de hash é criptografado com a chave privada do signatário.

Assinatura Digital: A assinatura digital é composta pelo valor de hash criptografado e pela chave pública do signatário.

Verificação da Assinatura: O destinatário do documento utiliza a chave pública do signatário para descriptografar o valor de hash e recalcular o hash do documento original. Se os hashes coincidirem, a assinatura digital é válida e o documento não foi alterado.

Vantagens da Assinatura Digital

A assinatura digital com chave pública e privada oferece diversas vantagens em relação aos métodos tradicionais de assinatura em papel:

Autenticidade: Garante que o documento foi assinado por quem alega ser o signatário, impedindo falsificações e repúdio.

Integridade: Garante que o documento não foi alterado após a assinatura, protegendo contra fraudes e manipulações.

Não Repúdio: O signatário não pode negar ter assinado o documento, pois a assinatura digital é uma prova criptográfica incontestável.

Eficiência: Elimina a necessidade de papel físico, assinaturas manuscritas e processos manuais de autenticação, otimizando o tempo e os custos das transações.

Segurança: Utiliza criptografia robusta para proteger a assinatura digital contra falsificações e interceptações.

Segurança e Gerenciamento das Chaves

A segurança da assinatura digital depende da confidencialidade da chave privada do signatário. Se a chave privada for perdida ou roubada, um invasor poderá criar assinaturas digitais falsas em nome do signatário. Por isso, é fundamental que a chave privada seja armazenada de forma segura em um dispositivo criptografado.

A assinatura digital com chave pública e privada é uma ferramenta essencial para garantir a autenticidade e a integridade dos dados digitais na era digital. Com a crescente adoção de transações online e a necessidade de garantir a segurança da informação, a assinatura digital continuará a desempenhar um papel fundamental na construção de um ambiente digital mais confiável e seguro.

2.3. Prova de Conhecimento Zero

As provas de conhecimento zero (ZKPs) são uma família de protocolos criptográficos que permitem que uma parte (denominada o provador) produza provas de conhecimento para um conjunto de valores chamados valores testemunhais. Estas provas permitem que o provador prove o conhecimento dos valores testemunha a outra parte (denominada o verificador) sem revelar informações adicionais sobre os próprios valores. Esse mecanismo se estende naturalmente à integridade computacional, ou seja, a garantia de que uma determinada computação foi realizada corretamente e honestamente.

Prova de conhecimento zero não interativa - zkSNARKs

ZK significa "zero knowledge" (conhecimento zero). Permite que um verificador verifique que uma prova feita por um provador é válida, sem aprender nada sobre as informações específicas na prova. Essa característica garante um alto nível de privacidade, permitindo que informações sensíveis permaneçam ocultas, ao mesmo tempo em que fornecem prova suficiente de sua legitimidade.

S significa "sucinto". Isso significa que o tamanho da prova deve ser relativamente pequeno. Geralmente, essas provas têm apenas algumas centenas de bytes de comprimento, permitindo uma verificação extremamente rápida em apenas alguns milissegundos.

N significa "não interativo". Essa propriedade é a mais importante dos zkSNARKs. Uma prova não interativa significa que ela pode ser gerada sem qualquer interação do verificador. Uma vez criada, a prova pode ser enviada a um verificador, que pode confirmar ou refutar a prova. A falta de interação permite que a prova seja publicamente verificada por qualquer pessoa, sem a necessidade de comunicação com o provador original.

AR significa "argumento (de)" e acompanha a próxima letra.

Vantagens das ZKPs

Privacidade : As ZKPs permitem que as partes provem fatos sem revelar informações confidenciais. Isso é crucial para transações em criptomoedas e outras aplicações onde a privacidade é essencial.

Eficiência : As provas são pequenas e a verificação é rápida, tornando-as adequadas para sistemas em tempo real.

Segurança : Aumentam a segurança sem comprometer a integridade das transações.

Desvantagens das ZKPs:

Computação Intensiva: A geração de provas pode ser computacionalmente custosa.

Confiança no Setup Inicial: A confiança no setup inicial (geração dos parâmetros) é fundamental.

Aplicações em Cofres Digitais

Em um cofre digital, as ZKPs podem ser usadas para garantir a privacidade dos arquivos armazenados. O usuário pode provar que possui acesso a determinados arquivos sem revelar quais são esses arquivos. Isso permite que os dados permaneçam seguros e confidenciais, mesmo quando armazenados em um ambiente compartilhado.

As provas de conhecimento zero são uma ferramenta poderosa para melhorar a privacidade, segurança e eficiência em sistemas criptográficos. Embora apresentem desafios, seu potencial é promissor para proteger informações sensíveis e garantir a confidencialidade em ambientes digitais.

2.4. OAuth - Autorização Simplificada para Aplicações Modernas

O OAuth 2.0 é um padrão de autorização online que permite que um aplicativo acesse recursos de outros aplicativos em nome de um usuário. Vamos explorar o que é o OAuth 2.0, como funciona e por que é essencial para a segurança e integração de sistemas. **(HARDT ET AL,2012)**

O protocolo define como os aplicativos podem obter acesso autorizado a recursos protegidos em nome de um usuário. Ele foi projetado para ser simples, seguro e flexível, permitindo que aplicativos se autentiquem e obtenham permissões para acessar dados ou executar ações em serviços externos.

Funcionamento

1 - Usuário Autentica o Aplicativo : Quando um usuário deseja usar um aplicativo que requer acesso a um serviço externo (como fazer login com o Google ou usar recursos do Facebook), o aplicativo redireciona o usuário para o provedor de autenticação (por exemplo, o Google).

2 - Autorização do Usuário : O provedor de autenticação solicita ao usuário que conceda permissão ao aplicativo para acessar seus dados ou executar ações em seu nome. Isso é feito por meio de telas de autorização.

3 - Obtenção do Token de Acesso : Se o usuário conceder permissão, o provedor de autenticação gera um token de acesso e o envia de volta ao aplicativo. Esse token é usado pelo aplicativo para acessar os recursos protegidos.

4 - Acesso aos Recursos : O aplicativo usa o token de acesso para fazer chamadas aos serviços externos em nome do usuário. O provedor de recursos (por exemplo, o Google ou o Facebook) verifica o token e permite ou nega o acesso.

Características Importantes do OAuth2

Segurança: O OAuth 2.0 permite que aplicativos acessem recursos sem expor as credenciais do usuário. Os tokens de acesso são temporários e podem ser revogados a qualquer momento.

Integração de Serviços: Com o OAuth 2.0, aplicativos podem se integrar facilmente a serviços externos, como redes sociais, armazenamento em nuvem e APIs de terceiros.

Escopo Granular: O usuário pode conceder permissões específicas (escopos, como somente leitura e/ou escrita) para cada aplicativo, controlando quais dados o aplicativo pode acessar.

O OAuth 2.0 é uma ferramenta poderosa para a segurança e integração de aplicativos modernos. Ele permite que os desenvolvedores criem experiências de usuário mais fluidas e seguras, enquanto mantêm o controle sobre o acesso aos dados.

2.5. Considerações Finais

Dada a visão geral dos tópicos que serão usados no trabalho vistos nesse capítulo podemos compreender o fluxo completo da aplicação. Onde o usuário irá subir a aplicação com facilidade, ao submeter aos seus arquivos ao cofre digital, seu binários serão convertidos para base64, podendo ser manipulados como um texto plano qualquer, esse conteúdo será cifrado com uma chave simétrica pela sua eficiência computacional e a chave simétrica cifrada com uma chave assimétrica, garantindo que apenas o dono possa ver o conteúdo cifrado. O servidor manipula os dados, mas não os conhece, se por qualquer motivo o tráfego for interceptado ou o servidor comprometido, nenhum dado será exposto. O usuário que for dono do arquivo poderá recuperá-lo sempre que quiser.

Este trabalho tem como objetivo implementar uma Prova de Conceito de uma aplicação segura utilizando criptografia de forma simplificada e otimizada. A aplicação visa ser leve e versátil, tanto em tecnologia quanto em interface, com protocolos de segurança robustos e criptografia de última geração.

3. Armazenamento de Dados Seguro com Zero-Knowledge para Serviços de Nuvem Pública

3.1. Preparando o ambiente para rodar a aplicação

Como descrito no capítulo anterior, esse projeto irá demonstrar o uso do Armazenamento de Dados Seguro, com o objetivo de tentar facilitar o trabalho do desenvolvedor de software, avaliando os pontos negativos
e positivos desse uso.

Este capítulo tem como objetivo apresentar e analisar as funcionalidades da aplicação, com o usuário subindo a aplicação, adicionando arquivos no cofre digital e posteriormente resgatando eles, usando tanto o plugin para armazenamento local quanto em nuvem pública (Google Drive).

Dessa forma, o Capítulo 3 está organizado de forma a mostrar primeiro os passos que o desenvolvedor deve seguir para a correta utilização do projeto e, ao final, discutir o que foi observado durante o desenvolvimento, uso e resultados do projeto.

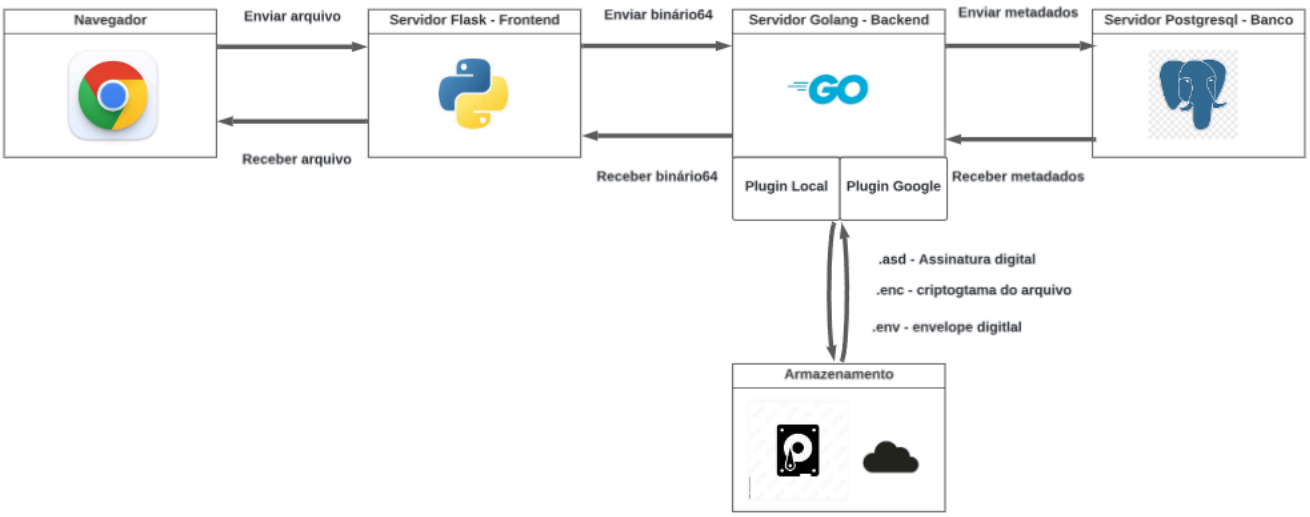


Figura 3.0 - Arquitetura da Aplicação do Cofre Digital

3.1.1. Preparando o ambiente Google

3.1.1.1. Criação do serviço

Caso o desenvolvedor não possua uma conta na Google, é necessária a criação. Para criar uma conta na Google, é necessário acessar o endereço https://accounts.google.com/signup, vide Figura 3.1. Nessa tela, ele deverá seguir as instruções que serão dadas para realizar o cadastro.

Figura 3.1: Página Inicial Google (https://accounts.google.com/signup)

Após finalizar o cadastro e a conta estar devidamente criada, o desenvolvedor deve seguir os passos dessa página https://developers.google.com/drive/activity/v2/quickstart/go?hl=pt-br

para ativar a API e configurar a tela de permissão, que estarei resumindo abaixo. Caso o desenvolvedor não tenha uma API e precise gerá-la, basta que, logado na sua conta Google, clique em “Ativar API”, vide figura 3.2 , na nova página selecione “Criar Projeto”, vide figura 3.3, siga a instruções da criação de um novo projeto, que nesse caso terá o nome “INF1951”, vide figura 3.4, após a criação do projeto só precisaremos ativar a API, vide figura 3.5 e 3.6. Retornando a figura 3.2 selecione “Acessar a tela de permissão OAuth” que te levará a tela de permissão, note que o projeto já foi selecionado automaticamente no topo esquerda da tela, devemos dar permissão para qualquer usuário fora da organização, após selecionar a opção correta devemos apertar “Criar”, vide figura 3.7, nessa tela de configuração de permissões pedirá o Nome da aplicação que pedirá permissão, devemos preencher e clicar em “Salvar e Continuar”, vide figura 3.8, Na tela de seleção de escopo é a parte mais delicada, aqui definimos quais permissões nossa API terá, é imprescindível que tenhamos permissão de CRUD no google drive, então nessa tela clique em “Adicionar ou Remover escopo” e na lista apresentada selecione o escopo “.../auth/drive.file” que permite “Ver, editar, criar e excluir apenas os arquivos do Google Drive que você usa com este app” e a final da página clicar em “Salvar e Continuar”, vide figura 3.9 e 3.10, sendo necessário apenas seguir clicando em “Salvar e Continuar” pois as tela seguintes são sobre testes o que não abordado neste projeto. Feito isso, temos nossa API ativada e sua tela de permissão para qualquer usuário que

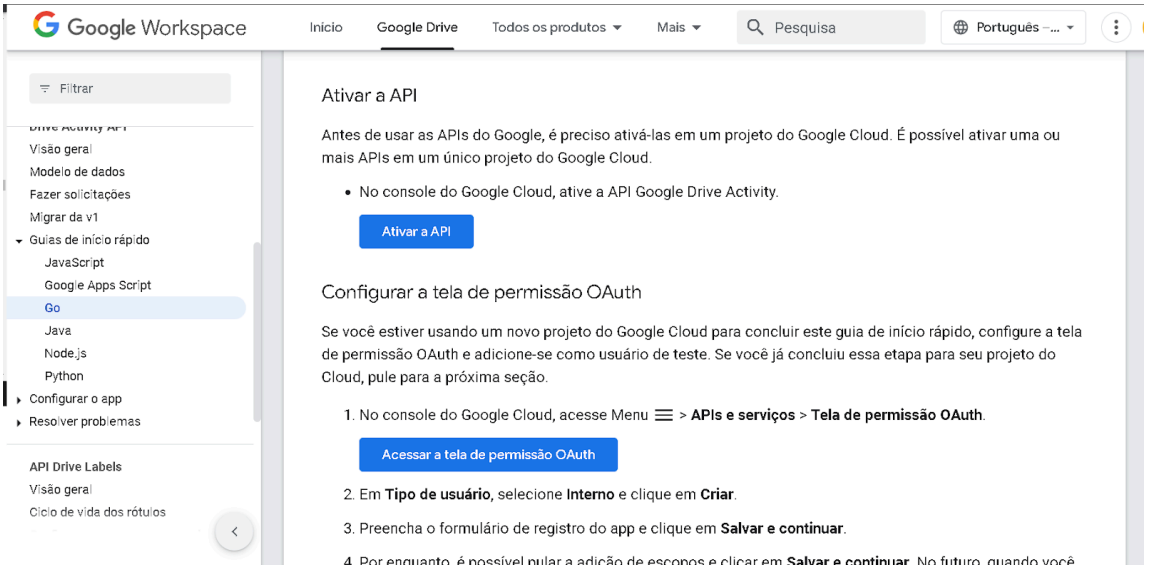


Figura 3.2: Página Inicial do Google WorkSpace (https://developers.google.com/drive/activity/v2/quickstart/go?hl=pt-br)

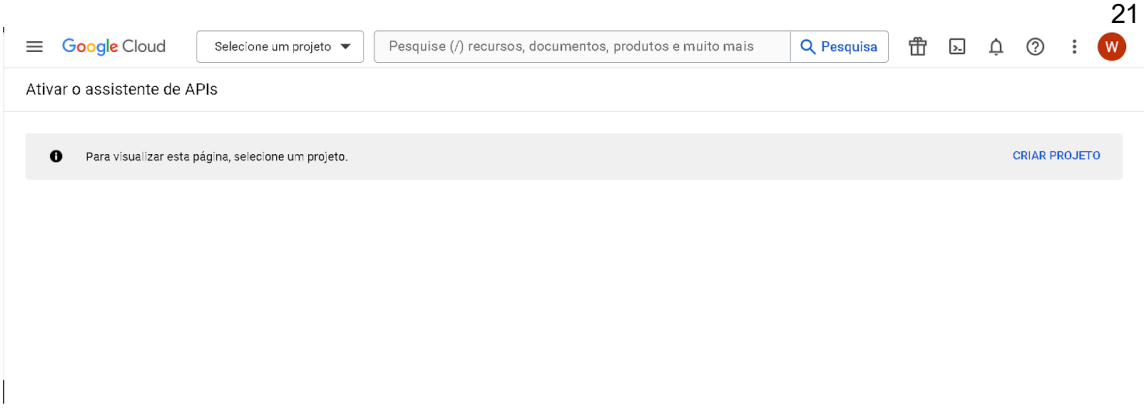


Figura 3.3: Página de projetos do Google Cloud (https://console.cloud.google.com/projectselector2/apis/enableflow?apiid=driveactivity.googleapis.com&hl=pt-br&supportedpurview=project)

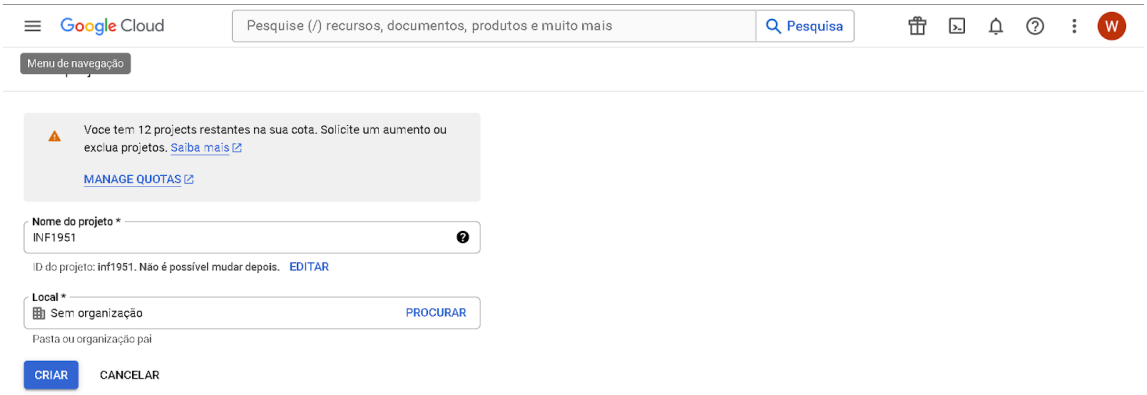


Figura 3.4: Página criação de projetos do Google Cloud

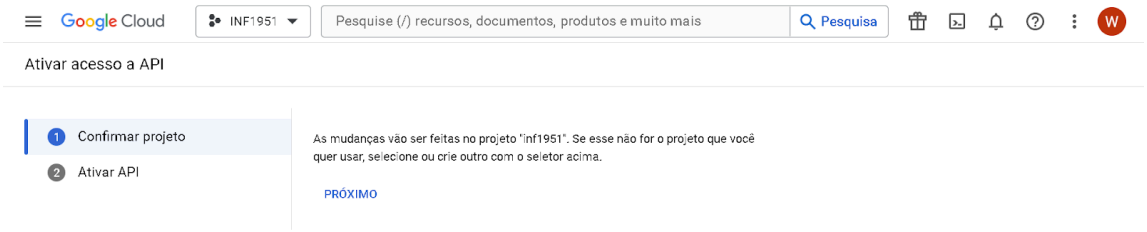


Figura 3.5: Página criação da API

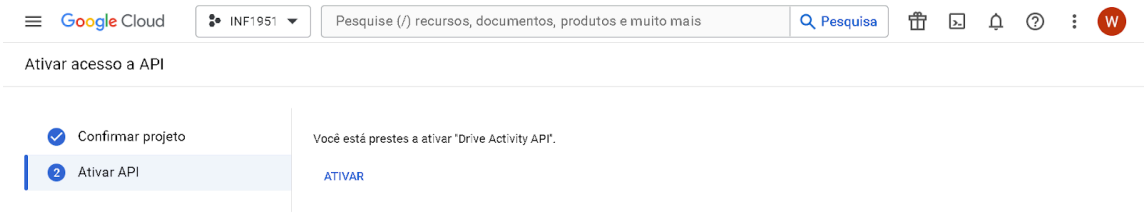


Figura 3.6: Página Ativação da API

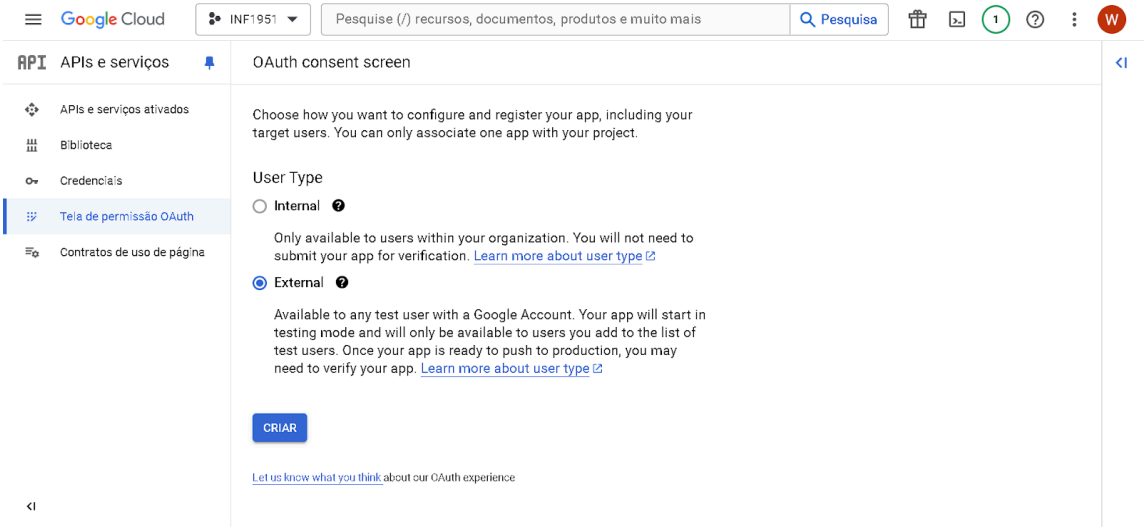


Figura 3.7: Página criação da Tela de Permissão

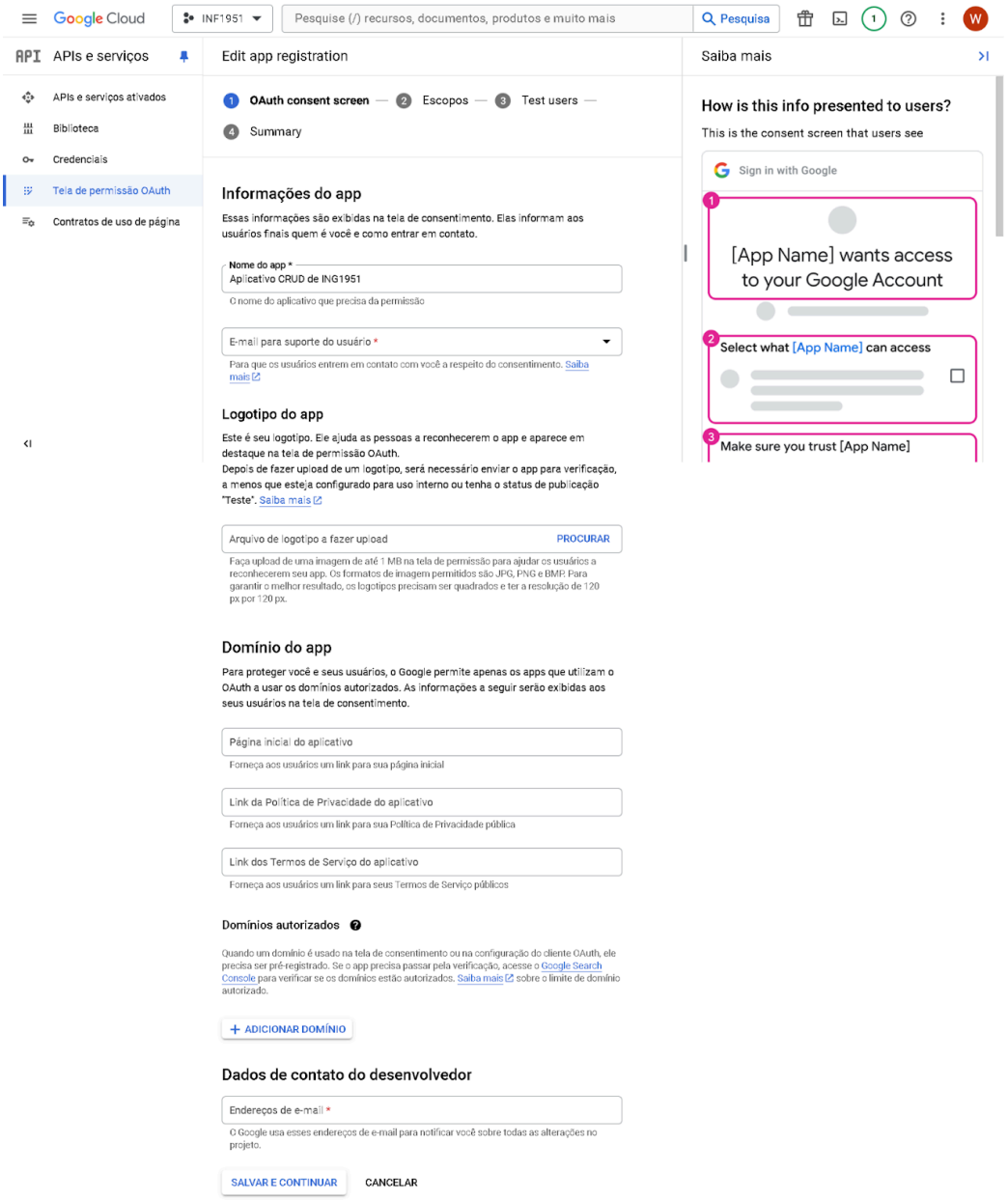


Figura 3.8: Página configuração de informação da Tela de Permissão

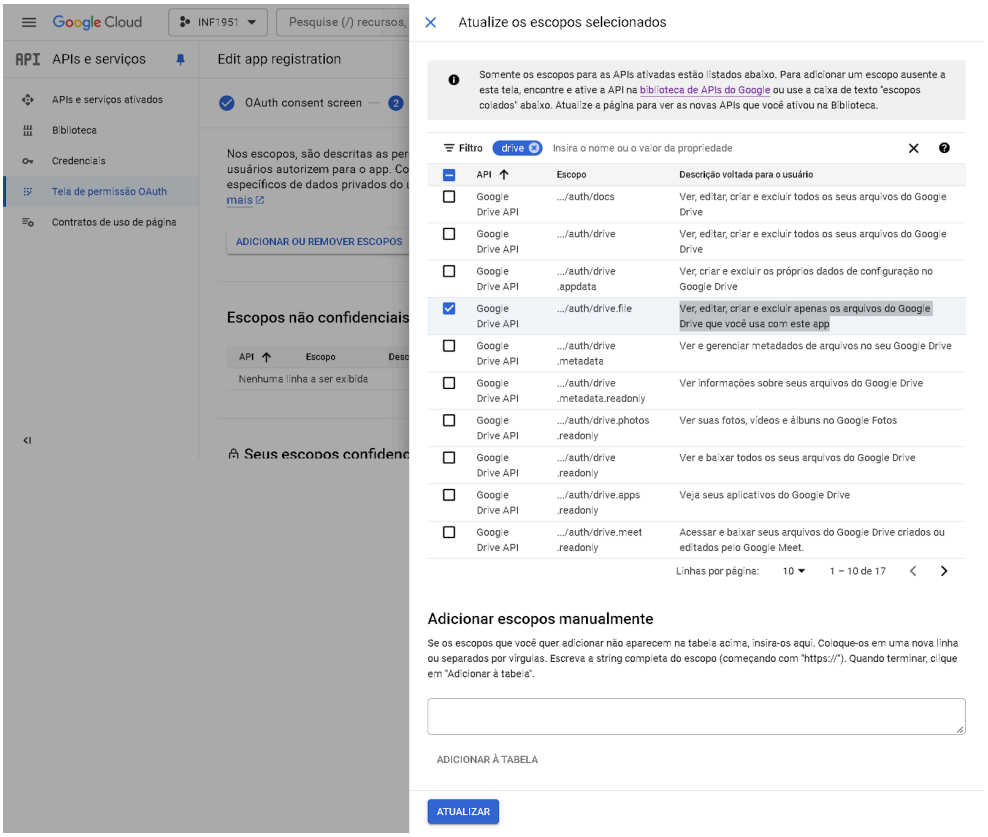


Figura 3.9: Página configuração de Escopo

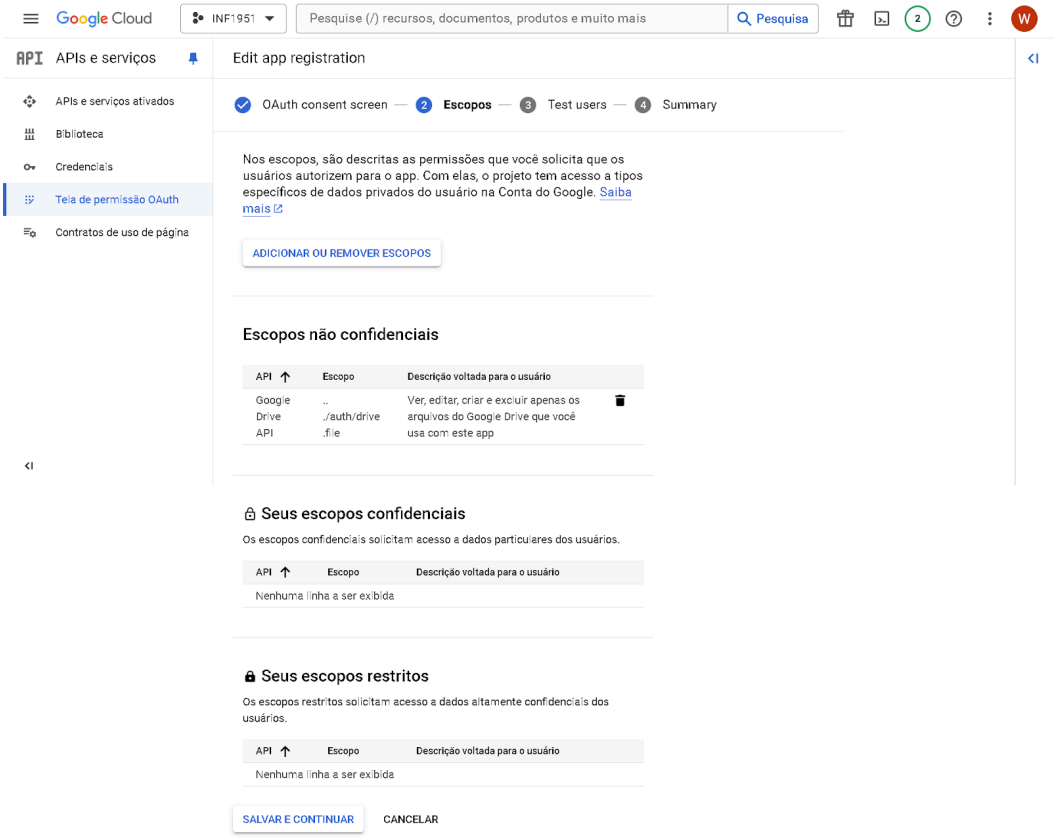


Figura 3.10: Página configuração de Escopo com escopo selecionado

3.1.1.2. Gerar a credencial de acesso

Uma vez preparado a API, o desenvolvedor deve gerar um token de acesso para a nossa aplicação ter permissão de uso de recursos necessários para o projeto.

O primeiro passo é voltar tela da Figura 3.2 e selecionar a opção “Ir para Credenciais”, nesta tela iremos criar as credenciais do OAuth2 selecionando em “Criar Credenciais” e clicar em “ID do cliente OAuth”, vide figura 3.11, selecione o tipo de aplicativo para “app de computador” e escolha um nome qualquer e clique em “criar” , vide Figura 3.12, após a criação da credencial, já é disponibilizado o download que deve ser nomeado para “credentials.json” e posto na pasta de certificados do projeto, vide Figura 3.13

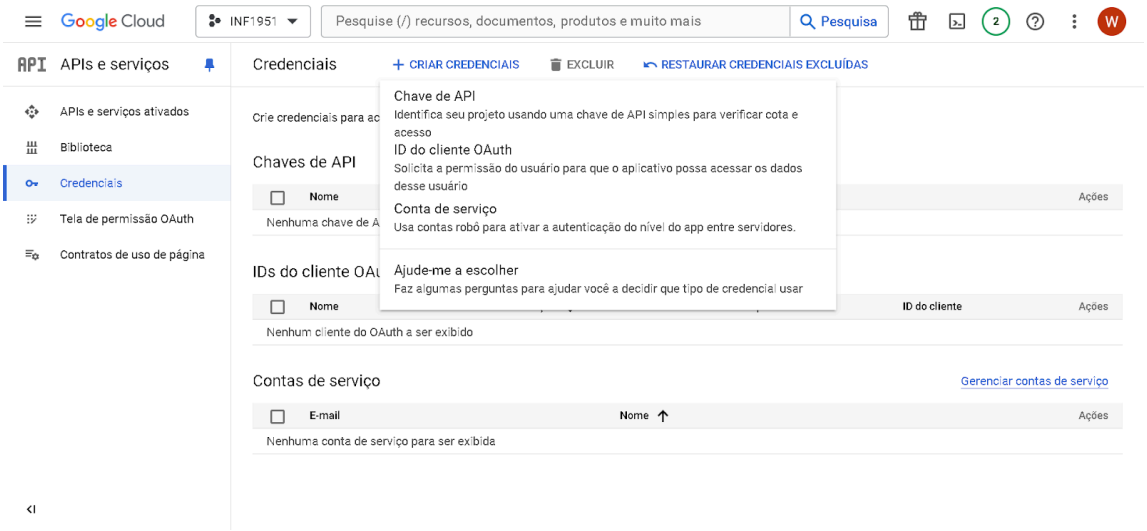


Figura 3.11: Página criação do ID do cliente Oauth2

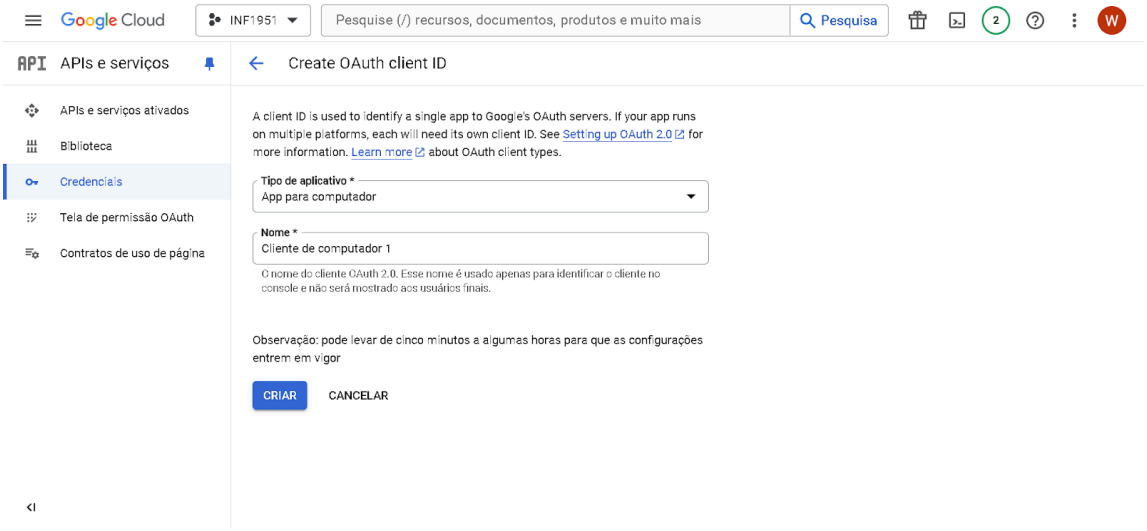


Figura 3.12: Página configuração do ID do cliente Oauth2

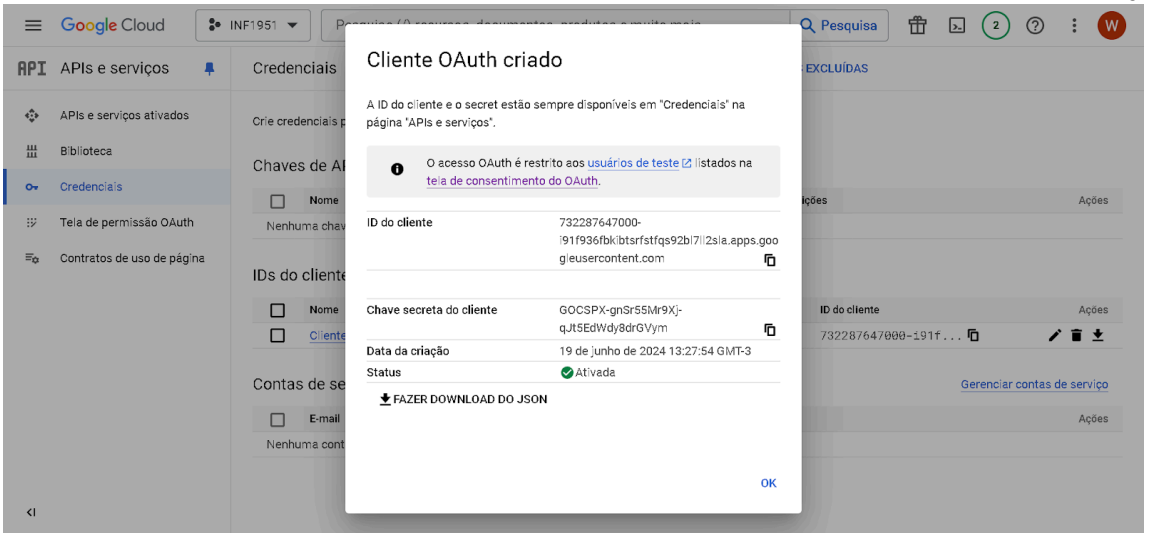


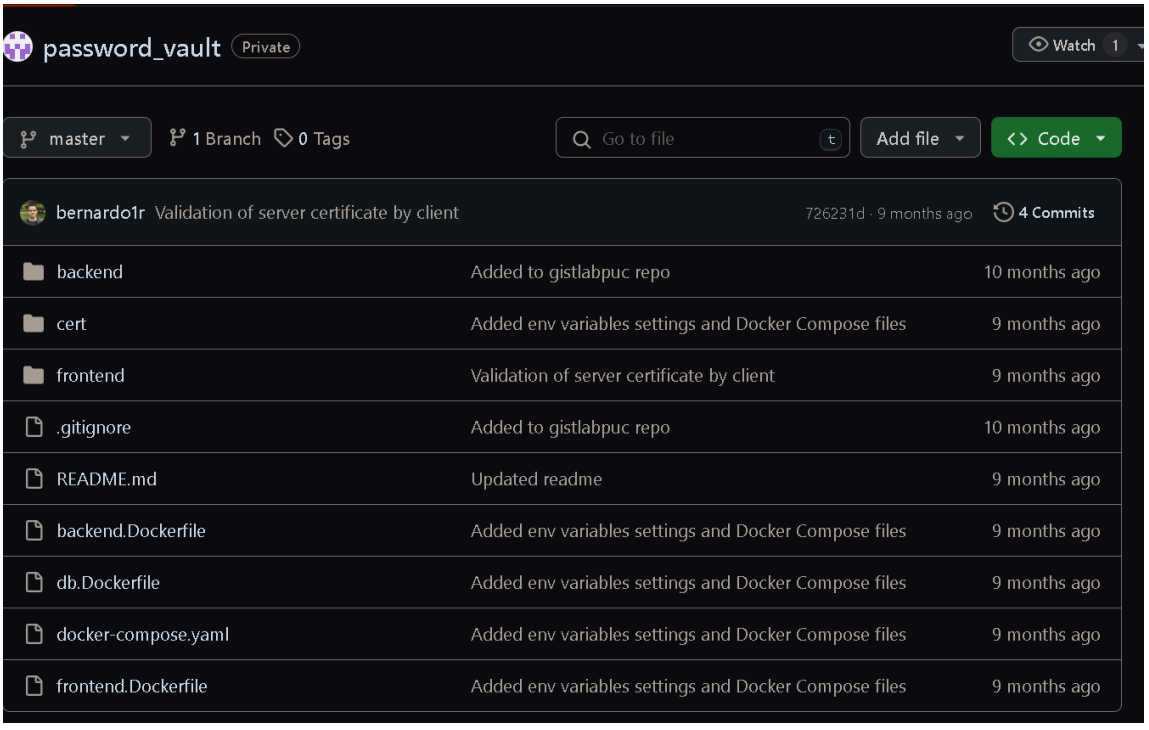
Figura 3.13: Página de download .json do cliente Oauth2

3.1.2. Preparando o ambiente local

Uma vez preparado o ambiente Google, o desenvolvedor deve instalar e configurar em sua máquina local alguns recursos necessários para o projeto. Um deles é ter o Docker (https://docs.docker.com/get-docker/) instalado com os passos referentes ao seu sistema e a biblioteca e Openssl (https://wiki.openssl.org/index.php/Binaries) para criação dos certificados e o Git (https://www.git-scm.com/downloads) para a clonagem do repositório.

3.1.2.1. Clonar o repositório

A primeira coisa a se fazer é clonar o repositório do projeto (https://github.com/gistlabpuc/password_vault), o repositório no momento é privado.



3.1.2.2. Criação dos certificados dos servidores

Com o repositório clonado, entre na pasta do projeto e use os comandos no openssl no terminal.

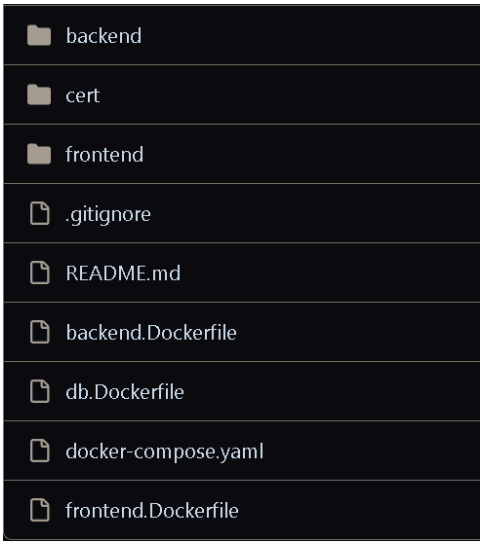
Para gerar o certificado do Backend

```
openssl req -newkey rsa:4096 -x509 -sha256 -nodes -out cert/backend/cert.pem -keyout cert/backend/key.pem
```

Para gerar o certificado do Frontend

```
openssl req -newkey rsa:4096 -x509 -sha256 -nodes -out cert/frontend/cert.pem -keyout cert/frontend/key.pem
```

3.1.3. Estrutura do projeto



O projeto contém diversos arquivos e está estruturado da seguinte maneira:

● README.md: descreve as instruções gerais do projeto;

● .gitignore: arquivo de configuração de itens a serem ignorados pelo Git;

● backend: diretório contendo os arquivos na linguagem Golang que serão copiados para o container que irá servir o Backend;

● frontend: diretório contendo os arquivos na linguagem Python que serão copiados para o container que irá servir o frontend;

● cert: diretório contendo os certificados do Backend, Frontend e da API do Google, esses certificados serão copiados para os containers do do backend e do frontend;

● .Dockerfile: arquivo que descreve como a imagem Docker do projeto deve ser instanciada para seu correto funcionamento;

● .Yaml: arquivo de configuração usado para definir e gerenciar os recursos alocados para seu correto funcionamento;

3.1.4. Detalhes da Implementação

Este projeto tem 2 principais arquivos de código-fonte**.** O primeiro deles é o “storage.go” que é encarregado de receber a chave privada, do texto plano e faz as operações criando os arquivos: .asd - Assinatura digital, .enc - Criptograma do arquivo, .env - Envelope digital e decidindo onde os arquivos serão salvos, sendo local, dentro do container ou no Google Drive. Além de executar todas as tarefas relacionadas ao salvamento em local.

```go
func CreateFile(filename string, file_content string, location string, base64_privatekey string) {

    // Decodificar a chave privada
    privateKeyBytes, _ := base64.StdEncoding.DecodeString(base64_privatekey)
    block, _ := pem.Decode(privateKeyBytes)
    privateKey, _ := x509.ParsePKCS1PrivateKey(block.Bytes)

    // Gerar a chave pública a partir da chave privada
    publicKey := &privateKey.PublicKey

    // Converter o conteúdo em bytes
    contentBytes := []byte(file_content)

    // Criar um hash SHA-256 do conteúdo
    hash := sha256.New()
    hash.Write(contentBytes)
    hashedContent := hash.Sum(nil)

    // Assinar o hash do conteúdo com a chave privada , criação do .asd
    signature, _ := rsa.SignPKCS1v15(rand.Reader, privateKey, crypto.SHA256, hashedContent)

    // Criar uma chave simétrica
    key := make([]byte, 32)
    if _, err := io.ReadFull(rand.Reader, key); err != nil {
        panic(err.Error())
    }

    // Criar um novo bloco cifrado usando a chave simétrica
    cipherBlock, err := aes.NewCipher(key)
    if err != nil {
        panic(err.Error())
    }

    // Criar um novo GCM - Galois/Counter Mode
    aesgcm, err := cipher.NewGCM(cipherBlock)
    if err != nil {
        panic(err.Error())
    }

    nonce := make([]byte, aesgcm.NonceSize())
    if _, err = io.ReadFull(rand.Reader, nonce); err != nil {
        panic(err.Error())
    }

    // Cifrar o conteúdo usando a chave simétrica , criação do .enc
    ciphertext := aesgcm.Seal(nonce, nonce, contentBytes, nil)

    // Criptografar a chave simétrica com a chave pública do usuário, criação do .env
    encryptedKey, err := rsa.EncryptPKCS1v15(rand.Reader, publicKey, key)
    if err != nil {
        panic(err.Error())
    }

    if location == "Local" {
        WriteLocal(filename+".asd", signature)
        WriteLocal(filename+".enc", ciphertext)
        WriteLocal(filename+".env", encryptedKey)
    }

    if location == "GDrive" {
        saveFileGdrive(filename+".asd", signature)
        saveFileGdrive(filename+".enc", ciphertext)
        saveFileGdrive(filename+".env", encryptedKey)
    }
}
```

Implementação “storage.go” - Parte 1

```go
func WriteLocal(filename string, content []byte) {
    // Escrever a assinatura no arquivo
    f, _ := os.Create(filename)
    defer f.Close()
    f.Write(content)

    // Mover o arquivo para a pasta "safe_vault"
    os.Rename(filename, "safe_vault/"+filename)
}

func ObtainFile(filename string, location string, base64_privatekey string) (string, error) {
    // Decodificar a chave privada
    privateKeyBytes, _ := base64.StdEncoding.DecodeString(base64_privatekey)
    block, _ := pem.Decode(privateKeyBytes)
    privateKey, _ := x509.ParsePKCS1PrivateKey(block.Bytes)

    // Ler a assinatura do arquivo
    signature := ReadFile(filename+".asd", location)

    // Ler o conteúdo cifrado do arquivo
    ciphertext := ReadFile(filename+".enc", location)

    // Ler a chave simétrica criptografada do arquivo
    encryptedKey := ReadFile(filename+".env", location)

    // Descriptografar a chave simétrica com a chave privada do usuário
    key, err := rsa.DecryptPKCS1v15(rand.Reader, privateKey, encryptedKey)
    if err != nil {
        return "", err
    }

    // Criar um novo bloco cifrado usando a chave simétrica
    cipherBlock, err := aes.NewCipher(key)
    if err != nil {
        return "", err
    }

    // Criar um novo GCM - Galois/Counter Mode
    aesgcm, err := cipher.NewGCM(cipherBlock)
    if err != nil {
        return "", err
    }

    // Verificar o tamanho do nonce
    if len(ciphertext) < aesgcm.NonceSize() {
        return "", fmt.Errorf("ciphertext too short")
    }

    // Separar o nonce e o ciphertext
    nonce, ciphertext := ciphertext[:aesgcm.NonceSize()], ciphertext[aesgcm.NonceSize():]

    // Descriptografar o conteúdo usando a chave simétrica
    plaintext, err := aesgcm.Open(nil, nonce, ciphertext, nil)
    if err != nil {
        return "", err
    }

    // Verificar a assinatura
    hash := sha256.New()
    hash.Write(plaintext)
    hashedContent := hash.Sum(nil)

    err = rsa.VerifyPKCS1v15(&privateKey.PublicKey, crypto.SHA256, hashedContent, signature)
    if err != nil {
        return "", err
    }

    return string(plaintext), nil
}
```

Implementação “storage.go” - Parte 2

```go
func ReadFile(filename string, location string) []byte {
    switch location {
    case "Local":
        return ReadLocal(filename)
    case "GDrive":
        return ReadFileGdrive(filename)
    default:
        fmt.Println("Unknown location. Returning dummy.")
        return []byte("dummy")
    }
}

func ReadLocal(filename string) []byte {
    // Ler o conteúdo do arquivo
    f, _ := os.Open("safe_vault/" + filename)
    defer f.Close()

    content, _ := io.ReadAll(f)

    return content
}

func DeleteFile(filename string, location string) {
    switch location {
    case "Local":
        DeleteLocal(filename + ".asd")
        DeleteLocal(filename + ".enc")
        DeleteLocal(filename + ".env")

    case "GDrive":
        DeleteFileGdrive(filename + ".asd")
        DeleteFileGdrive(filename + ".enc")
        DeleteFileGdrive(filename + ".env")
    default:
        fmt.Println("File not found. Nothing was deleted.")
    }
}

func DeleteLocal(filename string) error {
    // Apagar o arquivo
    err := os.Remove("safe_vault/" + filename)

    // Se houver um erro ao apagar o arquivo, retorne o erro
    if err != nil {
        return err
    }

    // Se o arquivo foi apagado com sucesso, retorne nil
    return nil
}
```

Implementação “storage.go” - Parte 3

O segundo é o “google.go”, que recebe os binários criados em “storage.go” e executa a conexão com a API do Google executando todas as tarefas relacionadas ao salvamento em nuvem.

```go
func saveFileGdrive(filename string, content []byte) {
    ctx := context.Background()
    b, err := ioutil.ReadFile("credentials.json")
    if err != nil {
        log.Fatalf("Unable to read client secret file: %v", err)
    }

    config, err := google.ConfigFromJSON(b, drive.DriveFileScope)
    if err != nil {
        log.Fatalf("Unable to parse client secret file to config: %v", err)
    }
    client := getClient(config)

    srv, err := drive.NewService(ctx, option.WithHTTPClient(client))
    if err != nil {
        log.Fatalf("Unable to retrieve Drive client: %v", err)
    }

    // Save the file to the root of Google Drive
    file := ioutil.NopCloser(bytes.NewReader(content))
    defer file.Close()

    driveFile, err := srv.Files.Create(&drive.File{Name: filename}).Media(file).Do()
    if err != nil {
        log.Fatalf("Unable to create file on drive: %v", err)
    }

    fmt.Printf("Saved file %s at %s\n", driveFile.Name, driveFile.Id)
}
```

Implementação “google.go” - Parte 1

```go
func ReadFileGdrive(filename string) []byte {
    ctx := context.Background()
    b, err := ioutil.ReadFile("credentials.json")
    if err != nil {
        log.Fatalf("Unable to read client secret file: %v", err)
    }

    config, err := google.ConfigFromJSON(b, drive.DriveFileScope)
    if err != nil {
        log.Fatalf("Unable to parse client secret file to config: %v", err)
    }
    client := getClient(config)

    srv, err := drive.NewService(ctx, option.WithHTTPClient(client))
    if err != nil {
        log.Fatalf("Unable to retrieve Drive client: %v", err)
    }

    // Find the file in the root of Google Drive
    fileList, err := srv.Files.List().Q(fmt.Sprintf("name='%s'", filename)).Do()
    if err != nil {
        log.Fatalf("Unable to retrieve files: %v", err)
    }
    if len(fileList.Files) == 0 {
        log.Fatalf("File not found")
    }

    // Download the file
    fileID := fileList.Files[0].Id
    response, err := srv.Files.Get(fileID).Download()
    if err != nil {
        log.Fatalf("Unable to download file: %v", err)
    }
    defer response.Body.Close()

    // Read the file into a byte slice
    content, err := ioutil.ReadAll(response.Body)
    if err != nil {
        log.Fatalf("Unable to read file: %v", err)
    }

    return content
}

```

Implementação “google.go” - Parte 2

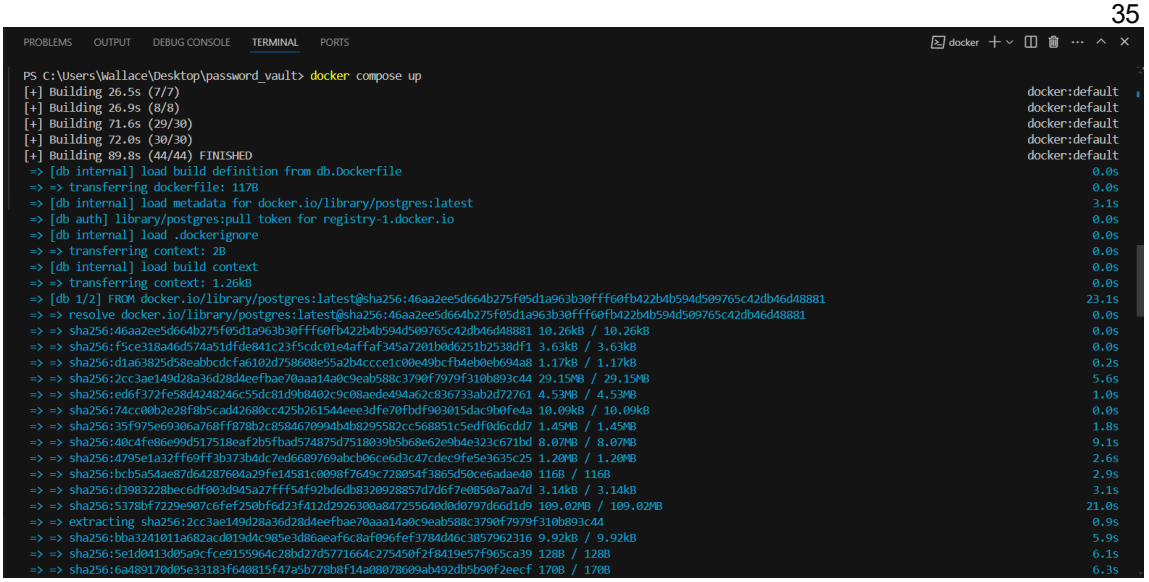
```go
func DeleteFileGdrive(filename string) error {
    ctx := context.Background()
    b, err := ioutil.ReadFile("credentials.json")
    if err != nil {
        log.Fatalf("Unable to read client secret file: %v", err)
    }

    config, err := google.ConfigFromJSON(b, drive.DriveFileScope)
    if err != nil {
        log.Fatalf("Unable to parse client secret file to config: %v", err)
    }
    client := getClient(config)

    srv, err := drive.NewService(ctx, option.WithHTTPClient(client))
    if err != nil {
        log.Fatalf("Unable to retrieve Drive client: %v", err)
    }

    // Find the file in the root of Google Drive
    fileList, err := srv.Files.List().Q(fmt.Sprintf("name='%s'", filename)).Do()
    if err != nil {
        log.Fatalf("Unable to retrieve files: %v", err)
    }
    if len(fileList.Files) == 0 {
        return fmt.Errorf("file not found")
    }

    // Delete the file
    fileID := fileList.Files[0].Id
    err = srv.Files.Delete(fileID).Do()
    if err != nil {
        log.Fatalf("Unable to delete file: %v", err)
    }

    fmt.Printf("Deleted file %s\n", filename)
    return nil
}
```

Implementação “google.go” - Parte 3

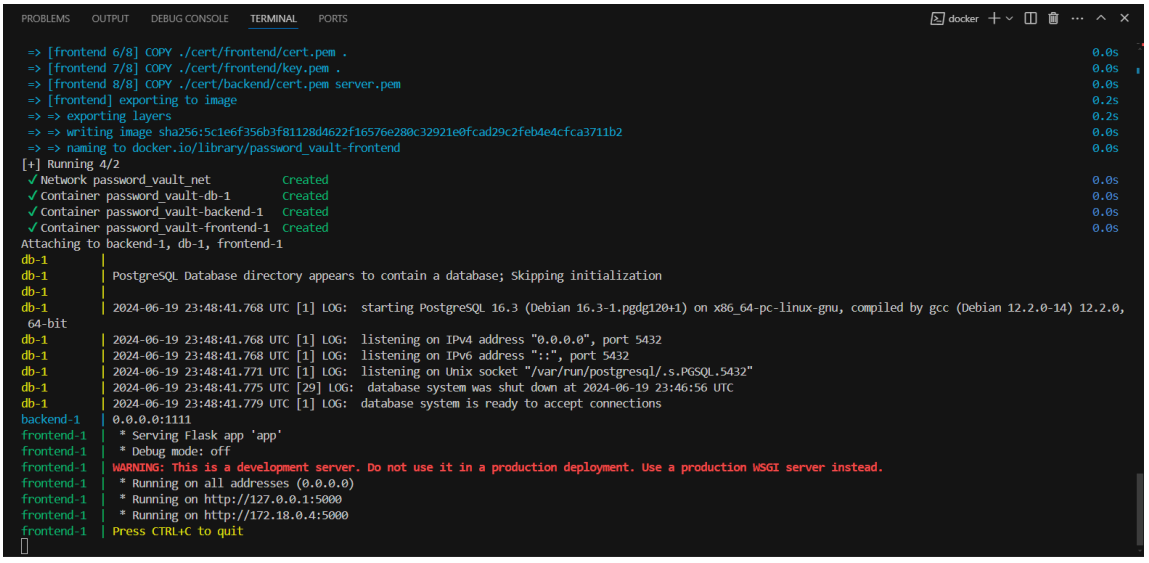
3.1.5. Executando a aplicação

Com todos os passos feitos, só é necessário entrar na pasta a projeto e usar o comando no terminal para rodar a aplicação.

```
docker compose up
```



Iniciando a aplicação - Início



Iniciando a aplicação - Fim

3.2. Discussão

3.2.1. Benefícios e Dificuldades Percebidas

Durante o desenvolvimento desse projeto, foi possível observar alguns pontos relevantes de serem destacados nesta monografia. Primeiramente, um grande benefício de se utilizar o Docker, é impressionante como com apenas um comando é capaz de levantar uma aplicação dessas sem nenhuma dificuldade. Basta usar o comando “docker compose up” e a engine faz todo o trabalho de levantar e configurar o ambiente. Isso simplifica muito o trabalho do desenvolvedor em questão e diminui a chance de erros de implementação neste aspecto.

Outro benefício encontrado é o fato de se utilizar linguagens como Python que é de fácil uso e sintaxe e Golang que é muito eficiente.

Porém, algumas desvantagens também devem ser citadas. Uma delas é que foi necessário grande conhecimento da documentação dos recursos da estrutura do docker e da API de serviços Google que foram utilizados nesse projeto para se conseguir fazer com que a aplicação se tornasse algo de fácil utilização pelo usuário final.

Outra desvantagem é a dificuldade para debugar o código recebido , uma vez que o conteúdo do projeto é carregado para o container, basicamente perdemos acesso a ele via IDE, o que fez a compreensão da estrutura de projeto e a adição de novos recursos uma tarefa bem desafiadora. necessitando destruir o container e recriá lo em toda e qualquer mudança de código.

Porém, é importante ressaltar que este projeto foi construído de forma a buscar otimizar/facilitar o trabalho de seus usuários ao máximo possível.

3.2.2. Avaliação da Facilidade de Uso

Com o objetivo de verificar a facilidade de uso do roteiro de implantação deste projeto, foi realizada uma avaliação por uma pessoa desenvolvedora de software.

Nessa avaliação, o participante foi solicitado a seguir o roteiro descrito na Seção 3.1 para implantar e executar a aplicação. Inicialmente, a pessoa conseguiu fazê-los seguindo o roteiro da forma adequada e, como feedback, solicitou apenas a inclusão de mais algumas informações/telas na criação da API do Google. Essa inclusão foi realizada e a pessoa confirmou que estava satisfeita com a alteração e que, mesmo sendo novata em relação às ferramentas e tecnologias utilizadas no projeto e não tendo participação alguma no desenvolvimento do mesmo, o roteiro de implantação e execução era suficientemente claro.

3.3. Considerações Finais

Neste capítulo, foi descrito como este projeto foi implementado, apresentando de forma resumida a sua estrutura e código-fonte, além de analisar suas principais características, descrevendo seus pontos positivos e negativos. Nele, também foi adicionado um roteiro para instruir a um desenvolvedor como poderia utilizar esse projeto, que foi devidamente validado e teve seus resultados analisados e descritos.

4. Conclusões

4.1. Contribuições

Conforme dito no Capítulo 1, este projeto tem o objetivo de apresentar alternativas para o desenvolvimento de aplicações de Cofres Digitais de forma menos burocrática e mais eficiente para o desenvolvedor responsável, aplicando o uso do Docker como recurso infraestrutural.

Visando alcançar esse objetivo, podemos concluir que essa monografia tem como contribuições os seguintes pontos:

● Fornecer uma fundamentação teórica sobre uma visão geral de Infraestrutura como código (IaC), DevOps, Criptografia, Hash, Prova de Conhecimento Zero, OAuth 2.0, apresentando os recursos utilizados no projeto;

● Armazenamento de Dados Seguro com Zero-Knowledge para Serviços de Nuvem Pública como infraestrutura, roteirizando todo o processo e dando exemplos de apoio para utilização do projeto e/ou reprodução do mesmo;

● Discutir o nível de dificuldade e de utilidade de se criar projetos unindo esses recursos, do roteiro e dos exemplos aqui fornecidos.

4.2. Limitações

Uma limitação do projeto é ser totalmente dependente da Docker Engine. Ao construir um projeto inteiramente em torno do Docker, você o torna altamente dependente dessa tecnologia. Isso pode dificultar a migração para outras plataformas ou ambientes se o Docker Engine não estiver disponível ou não for suportado em um novo ambiente, ou seja, caso em algum momento se queira mudar de plataforma, provavelmente o custo disso (em questão de retrabalho, tempo, aquisição de conhecimento e etc.) será bem alto.

4.3. Trabalhos Futuros

Podemos identificar alguns possíveis trabalhos futuros a serem desenvolvidos. A primeira possibilidade seria a criação de novos plugins para outras nuvens públicas, seja ela quais forem, para necessidade do desenvolvedor. Foi desenvolvido o plugin para cofre seguro Local e no Google Drive, mas muitos outros podem ser adicionados posteriormente.

A segunda possibilidade seria mudar o escopo do projeto para um modelo serverless como o Kubernetes, da forma que o projeto está estruturado, fazer essa mudança para tornar a aplicação disponível a qualquer pessoa é bem fácil de ser alcançada.

Finalmente, uma última possibilidade seria a melhoria da estrutura de algumas partes do código usando padrões de projetos, como por exemplo na seleção de qual disco virtual será usado, onde atualmente só temos 2 plugins, mas no futuro, teremos mais plugins desenvolvidos e isso se poderá se tornar um problema.

Referências bibliográficas

ANDERSON, Charles. **Docker [software engineering]**. Ieee Software, v. 32, n. 3, p. 102-c3, 2015.

HARDT, Dick. The OAuth 2.0 authorization framework. 2012.

BEN-KIKI, Oren; EVANS, Clark; INGERSON, Brian. Yaml ain't markup language (yaml™) version 1.1. Working Draft 2008, v. 5, n. 11, 2009.

ARTAC, Matej et al. DevOps: introducing infrastructure-as-code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, 2017. p. 497-498.

GOLDREICH, O;S. MICALI, and A. WIGDERSNn. **Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems**. Journal of the ACM, 38(3):p.690–728, 1991.

CANETTI ,R. **Security and composition of multi-party cryptographic protocols**. Journal of Cryptology, 13(1): p.143–202, 2000.

CHUNG, Minh Thanh et al. **Using docker in high performance computing applications**. In: 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE). IEEE,. p. 52-57. 2016.