UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Gabriel José Prá Gonçalves

Desenvolvimento de uma plataforma de investimentos em criptoativos baseada em Ethereum

Florianópolis
2020

Gabriel José Prá Gonçalves

Desenvolvimento de uma plataforma de investimentos em criptoativos baseada em Ethereum

Trabalho Conclusão do Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Bacharel em Engenharia de Controle e Automação.

Orientador: Professor Carlos Barros Montez, Dr.

Florianópolis
2020

Gabriel José Prá Gonçalves

Desenvolvimento de uma plataforma de investimentos em criptoativos baseada em Ethereum

Esta monografia foi julgada no contexto da disciplina DAS5511: Projeto de Fim de Curso e aprovada na sua forma final pelo Curso de Engenharia de Controle e Automação.

Florianópolis, 27 de novembro de 2019

Banca Examinadora:

Rafael Jung
Orientador na Empresa
Jungsoft

Carlos Barros Montez
Orientador no Curso
Universidade Federal de Santa Catarina

Rodrigo Tacla Saad
Avaliador
Universidade Federal de Santa Catarina

Vanderlei Munhoz Pereira Filho
Debatedor
Universidade Federal de Santa Catarina

Thuany Karoline Stuart
Debatedora
Universidade Federal de Santa Catarina

Resumo

No cenário global de crescimento acelerado do mercado de criptomoedas, diversas empresas e *startups* estão surgindo, propondo novas e diversas soluções baseadas na tecnologia *blockchain*. A *fintech* Sppyns faz parte desse movimento, desenvolvendo um *marketplace* global de fundos de criptoativos, conectando especialistas de gestão dessa classe de ativos a investidores dos mais variados perfis. O trabalho apresentado neste documento tem como foco o desenvolvimento do *backend* dessa plataforma, desde o levantamento de requisitos e planejamento até o bem sucedido lançamento da aplicação funcional em ambiente de produção, que já conta com mais de cem usuários registrados.

Palavras-chave: Sistemas web. Elixir. Phoenix. Blockchain. Ethereum. Aplicações Financeiras.

Abstract

In the global scenario of rapid growth of the cryptocurrency market, various companies and startups are emerging, proposing new and different solutions based on the blockchain technology. The fintech Sppyns is a part of that movement, developing a global marketplace of crypto assets, connecting experts in managing this class of assets to investors with the most diverse profiles. The work presented in this document has as its primary focus the development of this platform's backend, since the requirements discovery and planning until the successful release of the fully-functioning application in production, which already has more than one hundred registered users.

Keywords: Web systems. Elixir. Phoenix. Blockchain. Ethereum. Financial Applicaitons.

Lista de ilustrações

Figura 1 – *Exchange* de criptomoedas *Binance*, uma das maiores do mundo. . . . 10
Figura 2 – Exemplo de tela do *layout* produzida pela equipe de *UI/UX*. 13
Figura 3 – Componentes principais da arquitetura *Docker*. 18
Figura 4 – Visão geral da arquitetura do sistema. 19
Figura 5 – Diagrama do Banco de Dados do sistema. 21
Figura 6 – Tela de gerenciamento de contêineres no Portainer. 22
Figura 7 – Ilustração da arquitetura hexagonal. 25
Figura 8 – Fluxo de login. 26
Figura 9 – Fluxo de criação de conta. 27
Figura 10 – Fluxo de envio de passaporte e posterior verificação de identidade. . . . 28
Figura 11 – Fluxo de investir e resgatar. 30
Figura 12 – Fluxo de retirar dinheiro da plataforma. 30
Figura 13 – Fluxo de notificação de depósito. 31
Figura 14 – Gráfico de composição de um fundo. 32
Figura 15 – Interface para criação manual de avaliações. 35
Figura 16 – Gráfico de performance de um fundo. 37
Figura 17 – Gráfico de performance de um usuário. 37
Figura 18 – Exemplo de informações na tela de Portfolio. 38
Figura 19 – Exemplo de informações na tela de Portfolio. 38
Figura 20 – Exemplo básico de um programa em Solidity. 41
Figura 21 – Exemplo de um contrato representando um Fundo. 42
Figura 22 – Exemplo de uma transação de investimento. 43
Figura 23 – Exemplo de uma consulta GraphQL em um grafo cíclico. 46
Figura 24 – Exemplo de uma consulta bloqueada pelo sistema de autorização à nível de requisição. 47
Figura 25 – Exemplo de uma consulta bloqueada pelo sistema de autorização à nível de objeto. 47
Figura 26 – Resultado textual da ferramenta Sobelow. 48

Lista de abreviaturas e siglas

	Lista de Siglas
API	*Application Programming Interface*
DDD	*Domain-Driven Design*
TDD	*Test Driven Development*
CAS	*Crypto Asset Stack*
TAS	*Tokenized Asset Stack*
SOA	*Service Oriented Architecture*
REST	*Representational State Transfer*
UI	*User Interface*
UX	*User Experience*
CI	*Continuous Integration*
CD	*Continuous Development*
SGBD	Sistema de Gerenciamento de Banco de Dados
DSL	*Domain Specific Language*
SQL	*Structured Query Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
SaaS	*Software as a Service*
SMS	*Short Message Service*
IP	*Internet Protocol*
ERC	*Ethereum Request for Comments*
JWT	*JSON Web Token*
JSON	*JavaScript Object Notation*
DoS	*Denial of Service*
BEAM	*Erlang virtual machine*

Lista de símbolos

BTC	*Bitcoin*
ETH	*Ether*
$USDT$	*Tether*
USD	*United States Dollar*

Sumário

1	**INTRODUÇÃO**	**9**
1.1	**Motivação**	**9**
1.2	**Objetivos**	**9**
1.2.1	Objetivos Gerais	9
1.3	**Estrutura do documento**	**10**
2	**EMPRESAS ENVOLVIDAS**	**12**
2.1	**Jungsoft**	**12**
2.2	**Sppyns**	**12**
3	**ANÁLISE DE REQUISITOS**	**13**
3.1	**Requisitos Funcionais**	**13**
3.2	**Requisitos Não-Funcionais**	**14**
4	**ARQUITETURA E MODELAGEM**	**16**
4.1	**Aspectos Conceituais e Tecnologias**	**16**
4.1.1	Domain-Driven Design	16
4.1.2	Arquitetura Orientada a Serviços	17
4.1.3	Docker	17
4.1.4	GraphQL	17
4.2	**Arquitetura do Sistema**	**18**
4.2.1	Requisitos	18
4.3	**Modelagem Conceitual**	**20**
4.4	**Implementação e Resultados**	**20**
4.4.1	Infraestrutura	20
4.4.2	Banco de Dados	21
5	**INTEGRAÇÕES**	**23**
5.1	**Aspectos Conceituais e Tecnologias**	**23**
5.1.1	API	23
5.1.2	Desenvolvimento guiado por testes	23
5.1.3	Arquitetura Hexagonal	24
5.1.4	Azure	24
5.2	**Implementação e Resultados**	**25**
5.2.1	Azure Active Directory	25
5.2.2	Azure Blob Storage	26

5.2.3 Sppyns Wallet . . . 28
5.2.3.1 Arquitetura Hexagonal . . . 28
5.2.3.2 Segurança . . . 29
5.2.3.3 Operações . . . 29
5.2.4 Exchanges . . . 30

6 MEDIÇÃO DE DESEMPENHO . . . 32
6.1 Aspectos Conceituais e Tecnologias . . . 33
6.1.1 Cron Jobs . . . 33
6.1.2 Máquina Virtual Erlang . . . 33
6.1.3 GenServer . . . 33
6.1.4 Bitcoin . . . 33
6.1.5 Tether . . . 33
6.1.6 Fundo na Sppyns . . . 34
6.1.7 CAS . . . 34
6.1.8 TAS . . . 34
6.2 Implementação e Resultados . . . 35
6.2.1 Avaliação automática . . . 35
6.2.2 Consistência nas Avaliações . . . 35
6.2.3 Desempenho de um Fundo . . . 36
6.2.4 Desempenho de um Usuário . . . 36

7 ETHEREUM . . . 39
7.1 Aspectos Conceituais e Tecnologias . . . 39
7.1.1 Blockchain . . . 39
7.1.2 Ethereum . . . 39
7.1.3 Geth . . . 40
7.1.4 Solidity . . . 40
7.2 Implementação e Resultados . . . 41
7.2.1 Carteiras multi-assinatura . . . 42

8 SEGURANÇA . . . 44
8.1 Aspectos Conceituais e Tecnologias . . . 44
8.1.1 JSON Web Token . . . 44
8.1.2 Teste de Intrusão . . . 44
8.1.3 Ataque de negação de serviço . . . 44
8.2 Implementação e Resultado . . . 45
8.2.1 Segurança extra para movimentação de dinheiro . . . 45
8.2.2 Resistência a DoS . . . 45
8.2.3 Teste de Intrusão Interno . . . 46

8.2.4 Teste de Intrusão Externo 48

9 RESULTADOS E PERSPECTIVAS 49
9.1 Próximos Passos 49
9.1.1 Portal de Parceiros 49
9.1.2 Suporte a múltiplas moedas por fundo 49
9.1.3 Migração do Servidor 50

REFERÊNCIAS 51

1 Introdução

Presencia-se atualmente um acelerado crescimento da relevância das criptomoedas, habilitado pelo forte desenvolvimento da tecnologia *blockchain*, que permite a descentralização de diversos processos. Um desses processos é o de investimentos financeiros, classicamente ambientado nas bolsas de valores, controladas por uma entidade central em cada país, na qual se precisa confiar tanto na honestidade quanto na competência para que o mercado possa funcionar corretamente. O investimento em criptoativos, por outro lado, é geralmente reservado a especialistas e entusiastas da área, principalmente por envolver uma tecnologia muito nova e pouco conhecida pela população geral.

O projeto descrito neste documento tem como objetivo solucionar os problemas mencionados, desmitificando e democratizando as oportunidades de investimentos em criptoativos através de uma plataforma web amigável e de simples uso, possibilitando que qualquer pessoa, independente de seu conhecimento técnico, possa investir em criptomoedas.

O sistema desenvolvido utiliza a *blockchain Ethereum* para registrar as transações na plataforma, de forma descentralizada, segura e transparente.

1.1 Motivação

O mundo das criptomoedas encontra-se ainda muito distante da população geral, que o vê como muito complexo e mistificado. Percebe-se esse problema nas chamadas *exchanges*, casas de câmbio de criptomoedas virtuais, que focam em usuários experientes, utilizando termos especializados e gráficos complexos, como exemplifica a Figura 1.

A falta de inclusão é um grande problema, pois uma tecnologia descentralizada como a *blockchain* depende da adoção em massa, já que se torna mais robusta e confiável a medida que cresce o número de pessoas participando da rede. Tal fato é derivado das propriedades dos Algoritmos de Consenso, base da tecnologia *blockchain* que será explicado de forma mais detalhada no Capítulo 7.

1.2 Objetivos

1.2.1 Objetivos Gerais

O escopo do trabalho é o planejamento e desenvolvimento do *backend* de uma plataforma de investimentos em criptoativos. Para tanto, as seguintes atividades são contempladas:

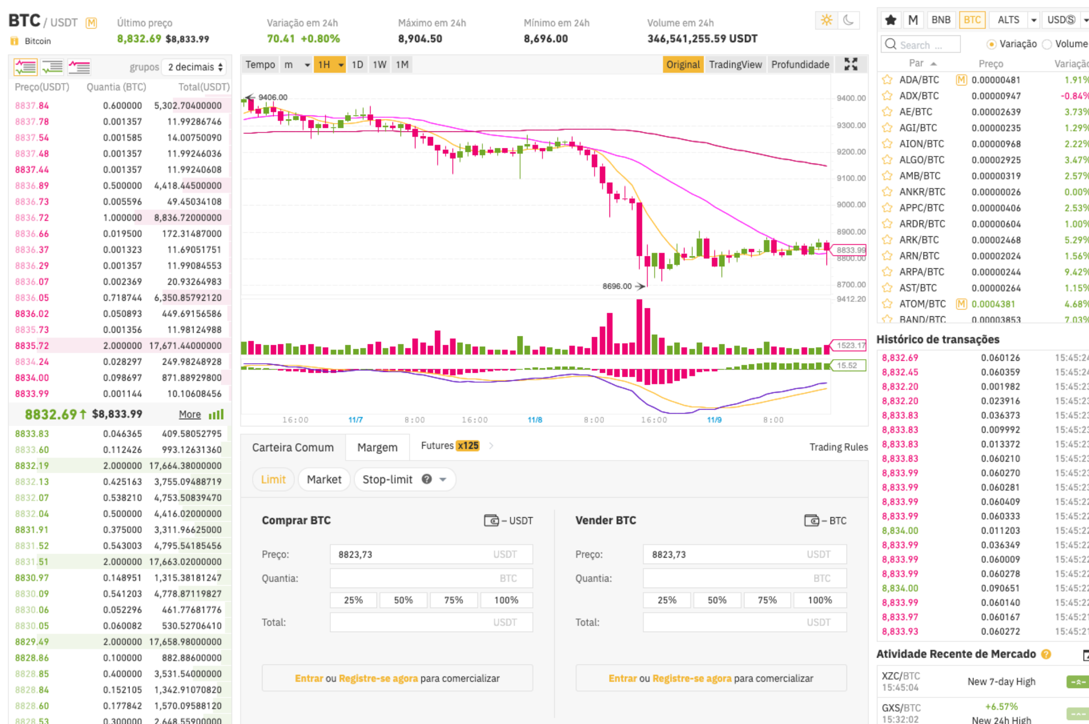

Figura 1 – *Exchange* de criptomoedas *Binance*, uma das maiores do mundo.

1. Planejamento da arquitetura do sistema, considerando suas integrações com serviços externos;
2. Planejamento do banco de dados;
3. Planejamento da API para comunicação com o *frontend*;
4. Desenvolvimento do *backend*, considerando todos os aspectos previamente planejados;
5. Integração com *Smart Contracts* na *blockchain Ethereum*;
6. Avaliação e aprimoramento da segurança do sistema;
7. Implantação e gerenciamento da infraestrutura no servidor;

1.3 Estrutura do documento

Este documento está dividido em 9 capítulos. Os capítulos 1 e 2 introduzem o problema a ser resolvido, assim como o escopo e o contexto do projeto realizado. O capítulo 3 tem como objetivo apresentar os requisitos gerais e específicos, de onde se baseia o raciocínio para as decisões tomadas nos capítulos subsequentes.

Os capítulos de 4 a 8 descrevem tarefas específicas, que em conjunto resultam na aplicação final. Esses capítulos iniciam apresentando o problema a ser resolvido e os

aspectos conceituais e tecnológicos necessários para sua resolução. Apresenta-se também eventuais requisitos específicos a serem cumpridos, a descrição da implementação da solução escolhida e os resultados finais.

O capítulo final apresenta uma visão geral dos resultados obtidos e perspectivas de desenvolvimento futuro para ambas empresas envolvidas no projeto.

2 Empresas Envolvidas

O projeto foi desenvolvido na empresa de desenvolvimento de *softwares web* **Jungsoft**. O cliente da Jungsoft para o qual o projeto foi desenvolvido é a **Sppyns**.

2.1 Jungsoft

A Jungsoft[1] é uma empresa de desenvolvimento de *softwares web* modernos, com escritórios em Florianópolis, Brasil e Berlim, Alemanha. A empresa foi estabelecida em 2018 e hoje conta com aproximadamente 15 funcionários, divididos entre desenvolvedores e designers.

Um dos modelos de negócio da empresa é ser responsável por todo o setor de tecnologia de *startups*, e é nesse modelo que se encaixa a relação com a Sppyns, sendo a Jungsoft responsável pelo desenvolvimento e manutenção da plataforma de investimentos, site comercial e toda a infraestrutura tecnólogica.

2.2 Sppyns

A Sppyns[2] é uma *startup fintech* criada em 2017 e atualmente sediada em Zug, Suíça, fazendo parte da *Crypto Valley Association*[3]. Trata-se de um *hub* que conecta globalmente negócios de cripto investimentos, consultoria e educação. A base deste *hub* é o *marketplace online* de fundos de investimentos em ativos digitais, descritos neste documento.

A proposta da Sppyns é ligar especialistas de gestão de ativos a pessoas querendo investir, mesmo que estas não possuam muito conhecimento prévio sobre o ecossistema dos criptoativos. Tal proposta é bem resumida em seu slogan: *"No geek, no problem."*.

[1] https://jungsoft.io
[2] https://sppyns.co
[3] https://cryptovalley.swiss

3 Análise de Requisitos

Este capítulo descreve as primeiras tarefas desenvolvidas no projeto, que visam levantar os requisitos do sistema a ser desenvolvido. Tais requisitos baseiam todo o planejamento, modelagem e desenvolvimento futuro.

3.1 Requisitos Funcionais

A metodologia utilizada para o levantamento de requisitos funcionais da plataforma foi analisar os casos de uso do sistema juntamente com o cliente e a equipe de *User Interface and User Experience, ou UI/UX,* gerando fluxogramas e *mockups* interativos, além de uma lista de requisitos mais técnicos, principalmente voltado ao *backend* da aplicação. Essa abordagem permite criar um sistema focado na experiência do usuário, identificando as funcionalidades que o sistema deve implementar e expor em sua API.

Outra vantagem desse método é facilitar a comunicação com todas as partes envolvidas no projeto, já que os *mockups* interativos expressam claramente as funcionalidades planejadas, sem requerer conhecimentos técnicos prévios.

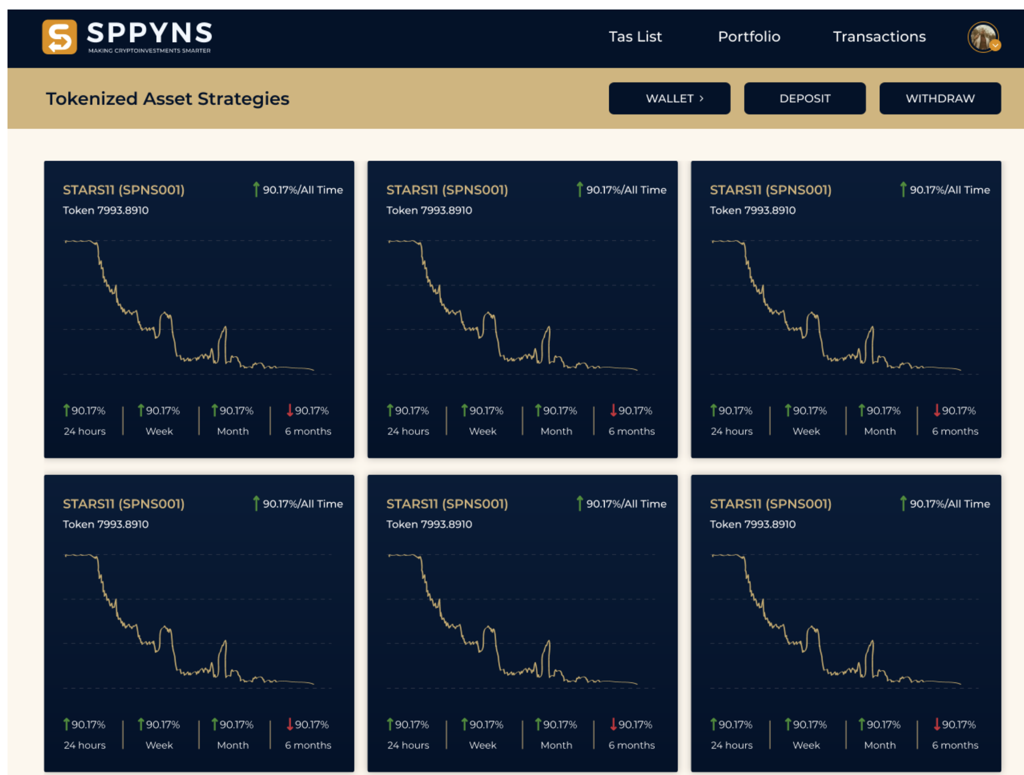

Figura 2 – Exemplo de tela do *layout* produzida pela equipe de *UI/UX*.

Através da análise das telas e fluxos planejados, foi possível listar as funcionalidades principais do sistema:

- Integração com a *blockchain*;
- Investimento e regaste em fundos, respeitando variadas regras e taxas configuráveis por fundo;
- Retirada de dinheiro da plataforma;
- Apresentação clara do histórico de investimentos dos usuários;
- Validação de identidade através do envio de documentos;
- Geração de relatórios para administradores da plataforma e gerentes de fundos;
- Suporte a diferentes tipos de fundos;
- Suporte a diferentes tipos de criptomoeadas;
- Funcionalidade de administrador para gerenciamento da plataforma;
- Análise de desempenho dos investimentos;
- Suporte a duas *Exchanges* iniciais, com flexibilidade para adição de novas.

3.2 Requisitos Não-Funcionais

Os requisitos não-funcionais são importantes para definir exigências gerais do projeto e devem ser estabelecidos antes das etapas de arquitetura e modelagem do sistema. Os requisitos não-funcionais vão além das funcionalidades a serem implementadas, determinando a infraestrutura a ser planejada e a metodologia de desenvolvimento a ser seguida.

O primeiro requisito trata da **segurança**, obviamente importante em aplicações financeiras. Não há uma determinada funcionalidade que atenda esse requisito, mas é necessário levá-lo em consideração durante todo o planejamento e desenvolvimento do sistema.

Outro requisito diz respeito a **robustez**, igualmente importante à segurança, pois erros podem ser tão problemáticos quanto ataques externos. Além de buscar essa característica em todas as funcionalidades implementadas, esse requisito teve papel fundamental na escolha da linguagem de programação a ser utilizada.

Tem-se, também, o requisito de **flexibilidade**, exemplificado pelos dois tipos distintos de Fundos presentes na aplicação, que lidam com diferentes moedas e *blockchains*.

Além disso, é esperado que uma *startup* apresente mudanças durante sua trajetória, que devem ser facilmente acomodadas no sistema.

Outo ponto importante é a **eficiência**, significando a menor necessidade de interferência humana possível, por meio de processos automatizados na plataforma. Outro fator que impacta a eficiência é a fácil manutenção do sistema, alcançado por um código limpo, bem estruturado e coberto por testes.

4 Arquitetura e Modelagem

Este capítulo descreve o planejamento básico do projeto, que consiste da Arquitetura do Sistema, onde se define quais são as interações da aplicação e como essas acontecem, e da Modelagem Conceitual, que define os diferentes contextos da aplicação e as entidades que os compõem.

4.1 Aspectos Conceituais e Tecnologias

Esta seção descreve os conceitos utilizados nas seções posteriores, fundamentais para a compreensão dos assuntos apresentados neste capítulo.

4.1.1 Domain-Driven Design

Domain-Driven Design (**DDD**) é uma abordagem para desenvolvimento de *software* complexos que busca decompor o sistema em pedaços menores e independentes, de forma a serem mais facilmente entendidos e gerenciados. Nessa abordagem, o foco principal é o **domínio** do problema, que consiste no conjunto de requisitos, linguagem, modelos, contextos e funcionalidades que o envolve [1].

Há um grande foco na linguagem do domínio, chamada de Linguagem Onipresente (*Ubiquitous Language*) que se refere a como especialistas da área se comunicam sobre determinados conceitos. Portanto, é essencial conversar com especialistas do domínio para melhor entender o problema e consequentemente modelá-lo de forma mais próxima a realidade.

Outro conceito fundamental é o de Contexto Limitado (*Bounded Context*), que se refere a contextos isolados dentro de uma aplicação, cada qual contendo seus próprios modelos, linguagem e responsabilidades. Muitas vezes, esses contextos são derivados diretamente da organização das equipes dentro de uma empresa.

O objetivo final dessa abordagem é lidar com *softwares* intrinsecamente complexos, aceitando o conhecimento de especialistas da área em que o problema se encontra e reconhecendo as diferentes facetas do domínio em questão.

Em alguns casos, os Contextos Limitados podem ser aplicações completamente separadas, desenvolvidas por times de desenvolvedores diferentes e usando tecnologias diferentes, o que indica uma grande separação de responsabilidades, fator muito importante na escalabilidade, flexibilidade e manutenibilidade de um sistema.

4.1.2 Arquitetura Orientada a Serviços

A Arquitetura Orientada a Serviços (*Service-Oriented Architecture (SOA)*) é uma abordagem para *design* de sistemas de *software* em que diferentes responsabilidades são distribuídas entre diferentes serviços, que são aplicações completamente independentes que se comunicam por meio de uma rede.

Percebe-se que essa arquitetura se encaixa bem com a filosofia do DDD, apresentado anteriormente, já que separa Contextos Limitados em serviços autônomos.

Uma variante bastante considerada atualmente é a Arquitetura baseada em Microserviços, em que cada serviço deve ser pequeno e leve, com apenas uma responsabilidade muito bem definida.

4.1.3 Docker

Docker é uma plataforma aberta para desenvolvimento, expedição e execução de aplicações, funcionando como uma camada de abstração sobre a virtualização de sistemas operacionais. Tal abstração permite separar a aplicação da infraestrutura, facilitando e acelerando a entrega de *software* [2].

O **contêiner** é um dos conceitos fundamentais dessa tecnologia. Trata-se de um ambiente isolado e leve, que executa diretamente no *kernel* da máquina que o hospeda. Assemelha-se a uma Máquina Virtual, porém muito mais leve e facilmente gerenciado, permitindo que centenas de contêineres executem em uma mesma máquina, ainda trazendo os benefícios de segurança e isolamento de uma máquina virtual tradicional.

A Figura 3 apresenta as principais peças que compõem o sistema *Docker*, mostrando que o mecanismo é bastante complexo e completo. As redes *Docker*, por exemplo, são utilizadas para comunicação entre diferentes contêineres e trazem diversas vantagens que serão exploradas também no capítulo sobre segurança.

4.1.4 GraphQL

GraphQL é uma linguagem de consulta (*Data Query Language*) criada internamente pelo *Facebook* em 2012. A utilização de *GraphQL* traz diversos benefícios em relação a APIs REST tradicionais [3]:

1. Evita *overfetching*, já que é possível solicitar apenas os dados necessários;
2. Possui um único *endpoint*, expondo uma estrutura de grafo. Dessa forma, não é necessário construir *endpoints* repetidos para telas específicas;
3. Apresenta boa integração com as ferramentas do ambiente *React*, melhorando a experiência de desenvolvimento do *frontend*;

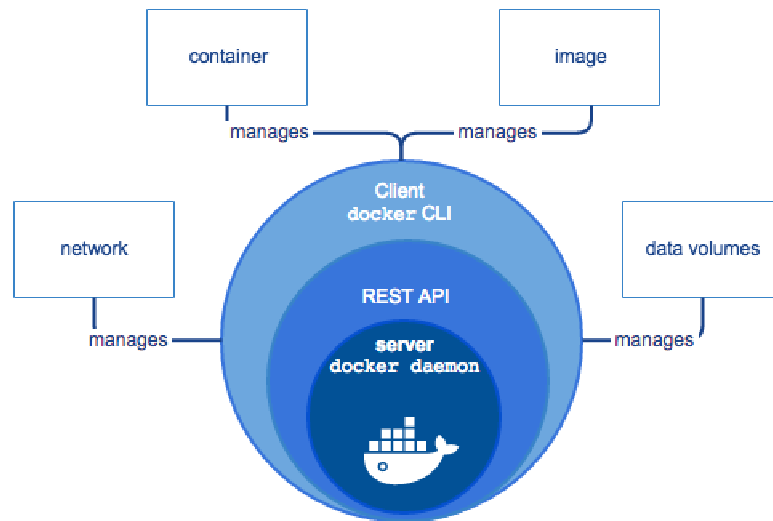

Figura 3 – Componentes principais da arquitetura *Docker*.

4.2 Arquitetura do Sistema

4.2.1 Requisitos

A arquitetura do sistema deve levar em consideração os requisitos não-funcionais do sistema. No contexto da arquitetura, destacam-se:

1. **Alta disponibilidade:** Em uma aplicação financeira, onde os usuários têm seu dinheiro investido, é altamente importante que o sistema esteja sempre disponível, evitando quaisquer sustos e desconfortos para os investidores;

2. **Segurança e robustez:** Quando se trata de movimentação de dinheiro, principalmente de dinheiro completamente virtual, como é o caso das criptomoedas, é fundamental que o sistema seja seguro, levando em conta que a motivação de atacantes maliciosos é alta;

3. **Agilidade:** Em uma *startup*, é fundamental que as iterações de desenvolvimento e *deployment* sejam as mais rápidas possíveis;

4. **Flexibilidade:** Por se tratar de uma *startup* em seus primeiros passos, é comum haver mudanças de requisitos e objetivos. Portanto, é importante que o sistema seja flexível e facilmente modificável;

Levando em conta esses requisitos, planejou-se a arquitetura descrita de forma geral pelo diagrama da Figura 4:

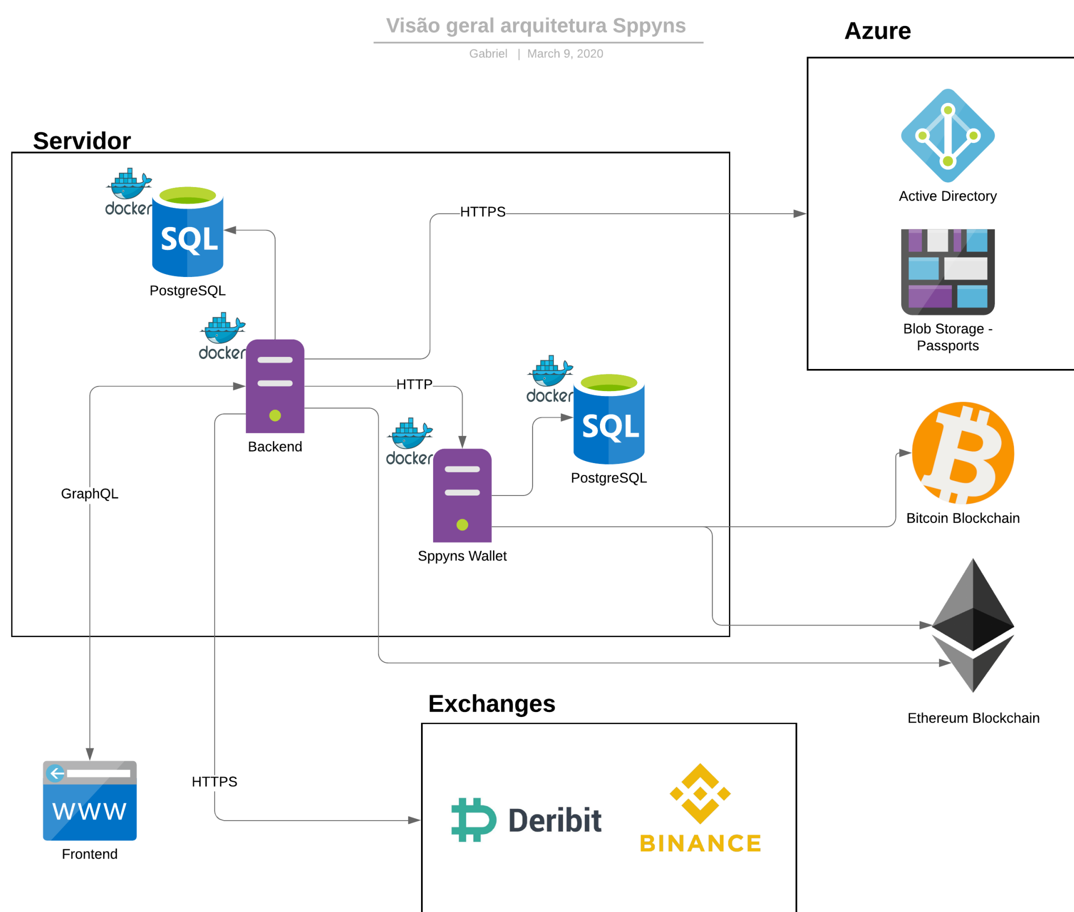

Figura 4 – Visão geral da arquitetura do sistema.

Conforme apresentado na Figura 4, a aplicação foi separada em 3 serviços principais: Backend, Sppyns Wallet e Frontend, além dos sistemas externos, também apresentados na visão geral.

O **Sppyns Wallet** é o serviço responsável por interagir com a *blockchain* e gerenciar as carteiras dos usuários. É um sistema baseado em uma plataforma *open-source* de desenvolvimento de *exchanges* de criptomoeadas Peatio. Desenvolvido em *Ruby*, o sistema é mantido por outro desenvolvedor da equipe e não será tratado em profundidade neste documento.

O **Frontend**, componente da aplicação que interage diretamente com os usuários através de uma interface gráfica, desenvolvido em *React*, também é responsabilidade de outras pessoas da equipe e não é o foco do documento. Entretanto, há uma forte interação entre o *Backend* e o *Frontend*, que se comunicam por meio de uma API *GraphQL*.

Já o **Backend**, foco deste documento, foi desenvolvido usando a linguagem *Elixir*[1]

[1] https://elixir-lang.org/

e a *framework web Phoenix*[2]. A escolha dessas tecnologias está diretamente ligada aos requisitos de disponibilidade, robustez e flexibilidade, já que se trata de um ecossistema distribuído e tolerante a falhas. A tolerância a falhas se deve ao fato de que erros em um componente do sistema não leva à falha do sistema como um todo, pois existem mecanismos como os processos supervisores que possibilitam a restauração desses componentes em caso de falha. [4]

4.3 Modelagem Conceitual

Na etapa de Modelagem Conceitual do sistema descrito anteriormente como *Backend*, busca-se entender e definir o domínio da aplicação. Para isso, define-se os contextos que compõem a aplicação, além das entidades que por sua vez compõem esses contextos.

Inicia-se com o planejamento do Banco de Dados do sistema, utilizando um Diagrama Relacional, o que permite uma visão geral do sistema, para a subsequente identificação dos contextos.

Semelhantemente à metodologia para levantamento de Requisitos Funcionais, a modelagem do banco de dados baseia-se nos casos de uso do sistema, assim como as informações apresentadas nas diferentes páginas planejadas, ambos resultantes do trabalho em conjunto com a equipe de *UI/UX*.

O diagrama relacional do Banco de Dados final é apresentado na Figura 5, onde alguns campos foram omitidos, principalmente da tabela *Funds*, para facilitar a visualização.

4.4 Implementação e Resultados

4.4.1 Infraestrutura

Cada um dos serviços, assim como os bancos de dados a eles associados, são executados em um contêiner *Docker*, o que traz como principal vantagem a Agilidade, permitindo um fácil gerenciamento da infraestrutura e *deploys* rápidos, principalmente ao se considerar que as Imagens *Docker* são automaticamente geradas no sistema de Integração Contínua (*CI*) e Entrega Contínua (*CD*), prontas para serem executadas no servidor [5].

Para configurar todos os contêineres que compõem o sistema, utiliza-se a ferramenta *Docker Compose*, que auxilia no gerenciamento de aplicações compostas de múltiplos contêineres, centralizando a configuração em um único arquivo *YAML*[3], o qual descreve

[2] https://phoenixframework.org/
[3] https://yaml.org/

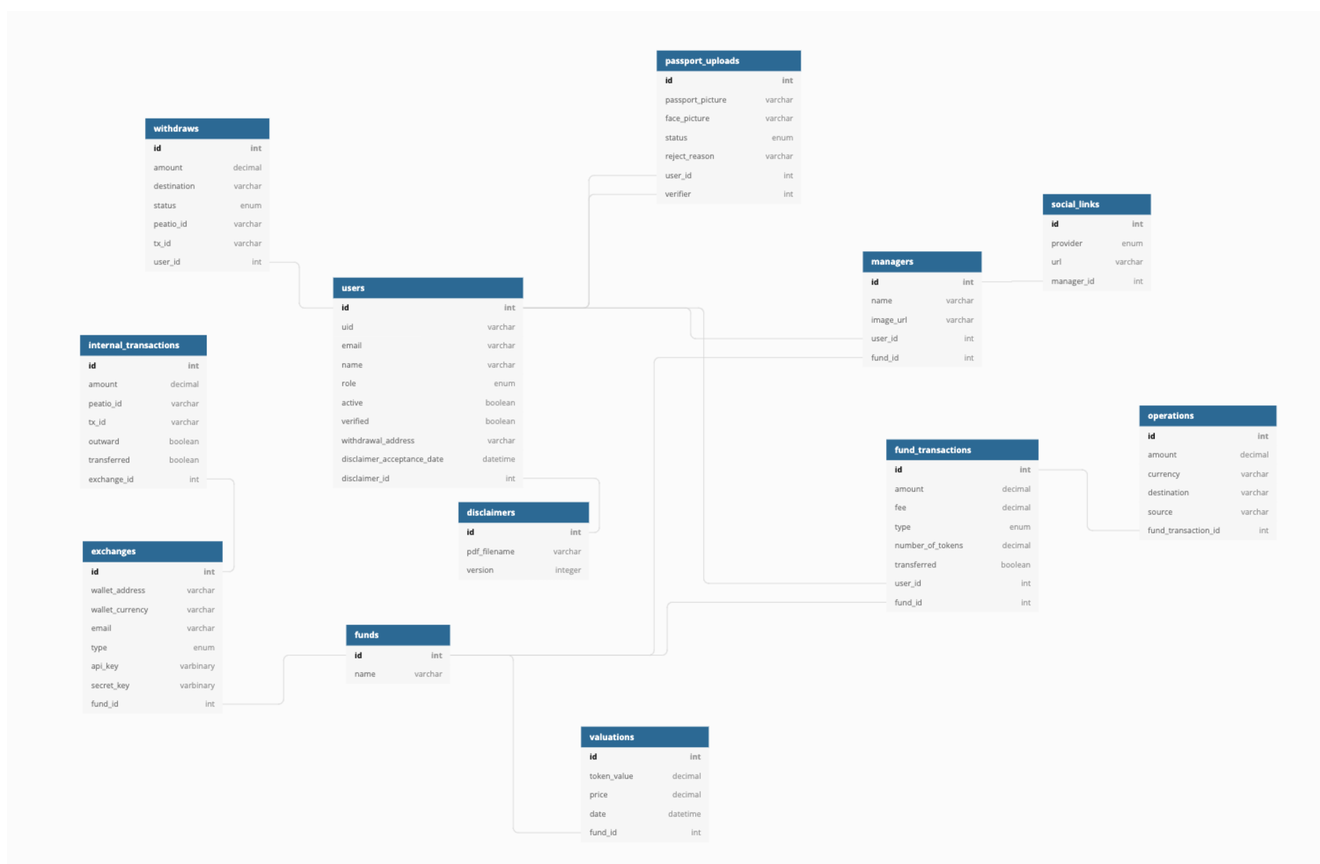

Figura 5 – Diagrama do Banco de Dados do sistema.

não só as configurações individuais de cada contêiner, como as relações de dependência existentes entre estes [6].

Utiliza-se também a ferramenta *Portainer*[4] para gerenciar os ambientes *Docker*, provendo uma interface gráfica amigável que facilita a atualização e supervisionamento dos contêineres. A Figura 6 apresenta um exemplo de uma das telas da ferramenta.

Para alcançar a arquitetura planejada, desenvolveu-se um *Dockerfile* (arquivo de configuração *Docker* que descreve uma imagem, cuja instância em execução é um contêiner, já previamente esclarecido) para o *Frontend* e um para o *Backend*, além de um arquivo de configuração do *Docker Compose*. A parte do *Sppyns Wallet* foi desenvolvida separadamente, já que diz respeito a outro repositório [7].

4.4.2 Banco de Dados

Optou-se por utilizar um banco de dados relacional, por ser o tipo de banco de dados mais amplamente utilizado e conhecido, além de não haver nenhum requisito do projeto que demande algo diferente a nível de banco de dados.

[4] https://www.portainer.io/

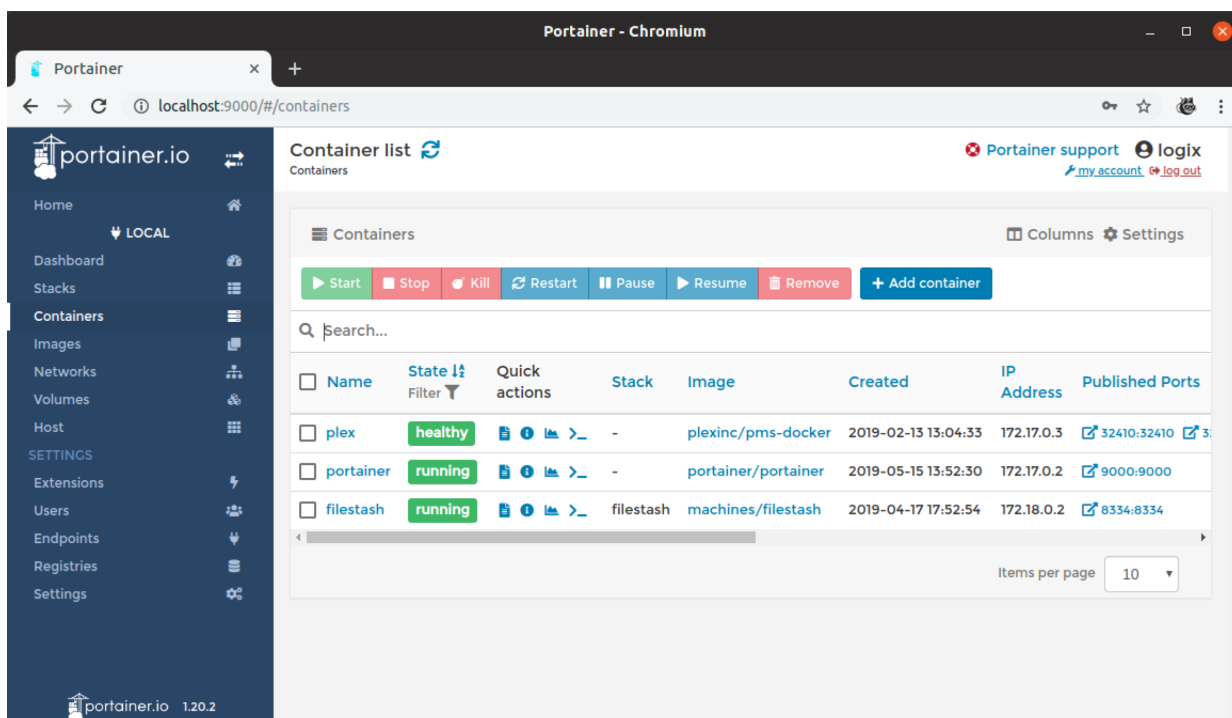

Figura 6 – Tela de gerenciamento de contêineres no Portainer.

Entre os sistemas de banco de dados relacionais, o Sistema Gerenciador de Banco de Dados Objeto Relacional (SGBD) escolhido foi o **PostgreSQL**, por ser o maior banco de dados relacional de código aberto do mundo e amplamente testado, contendo todas as características necessárias, como transações atômicas, consistência e chaves estrangeiras. Além disso, este é o Banco de Dados padrão ao se utilizar a *framework Phoenix*, o que faz com que existam mais e melhores ferramentas [8].

Tratando-se da aplicação em *Elixir*, utiliza-se a biblioteca *Ecto*[5], que fornece ferramentas como mapeamento dos dados no Banco de Dados para estruturas em *Elixir*; validação de dados; e uma *DSL (Domain Specific Language)*, ou Linguagem de domínio específico, que permite a construção de consultas elaboradas utilizando a linguagem *Elixir*, que é transformada automaticamente em consultas *SQL*.

Além disso, existe também uma ferramenta para criar migrações no banco de dados, o que permite alterar a estrutura deste de forma segura e controlada. Isso permite, por exemplo, deletar tabelas sem perder dados, evitando que o banco de dados atinja pontos de inconsistência.

[5] https://github.com/elixir-ecto/ecto

5 Integrações

Como apresentado no capítulo anterior, foram planejadas algumas integrações com sistemas externos, são eles:

1. **Azure Active Directory**, responsável pela autenticação dos usuários;
2. **Azure Blob Storage**, responsável por armazenar os passaportes dos usuários;
3. **Exchanges Deribit e Binance**, onde os gerentes dos fundos de investimentos operarão;
4. **Bitcoin blockchain**, através do serviço *Sppyns Wallet*;
5. **Ethereum blockchain**;

Este capítulo discutirá todas essas integrações, exceto a com a *blockchain Ethereum*, que será abordada em mais profundidade no Capítulo 7.

5.1 Aspectos Conceituais e Tecnologias

5.1.1 API

Uma *Application Programming Interface*, ou simplesmente *API*, é uma interface definida por um conjunto de regras - escritas na forma de código - e especificações que programas de *software* seguem para se comunicar com um determinado serviço. Uma *API* pode ser vista como um contrato, ou seja, uma promessa de fornecer os serviços descritos quando chamada de formas específicas.

Um subconjunto de *APIs* bastante conhecido, principalmente na *web*, é o de *APIs RESTful*, que tem com característica ser *stateless* e usar as quatro ações do protocolo *HTTP* para definir as requisições: *GET, POST, PUT e DELETE.*

A característica *stateless* significa que as ações não possuem estado dinâmico, de forma que requisições passadas não influenciam as futuras e portanto toda a informação necessária para processar uma requisição sempre está contida na própria requisição [9].

5.1.2 Desenvolvimento guiado por testes

Também conhecido como *Test Driven Development*, ou **TDD**, o desenvolvimento guiado por testes é uma técnica de desenvolvimento de *software* em que primeiro se

desenvolve o teste para a funcionalidade desejada. Ao notar-se que este falha, desenvolve-se o código de forma a fazer o teste passar. Então, há uma oportunidade de refatoração, onde se pensa se há uma melhor forma de resolver o problema. Após essa etapa, o ciclo se repete escrevendo o próximo teste.

O objetivo desta técnica é pensar nos requisitos antes de iniciar a implementação de fato, guiando o desenvolvimento naturalmente à mais simples solução, já que não se deve implementar nada mais que não tenha como objetivo satisfazer o teste. O TDD é bastante utilizado em metodologias ágeis como forma de alcançar um código limpo, robusto e de fácil entendimento.

5.1.3 Arquitetura Hexagonal

A Arquitetura Hexagonal, também conhecida como Arquitetura de Interfaces e Serviços (*Ports and Adapters Architecture*), é um padrão de arquitetura para organização e design de *software*. O objetivo dessa arquitetura é produzir *software* altamente flexível, evitando acoplamento entre diferentes componentes, que só devem interagir através de interfaces (*ports*), sem se importar qual é a implementação (*adapter*). Essa estratégia permite a fácil troca de implementações diferentes para um mesmo serviço, o que facilita a criação de testes automatizados [10].

Além disso, esse padrão incentiva a separação da aplicação em diversas camadas, sendo a camada mais interna a lógica de negócio da aplicação, que depende apenas de *ports*, nunca de detalhes de implementação. Dessa forma, evita-se misturar a lógica de negócio com detalhes de implementação ou lógica relacionada a interface gráfica da aplicação.

É importante que componentes de uma camada dependam apenas de interfaces da camada imediatamente superior, aumentando ainda mais o desacoplamento entre diferentes partes do sistema, que possuem responsabilidades distintas.

5.1.4 Azure

O *Microsoft Azure* é uma plataforma de computação em nuvem, oferecendo diversos serviços no modelo de *software* como serviço (*SaaS*). Entre os serviços oferecidos, encontram-se Inteligência Artificial, Internet das Coisas, Web, Realidade Aumentada, entre outros. No contexto da aplicação desenvolvida, três serviços são relevantes [12].

O primeiro é o *Azure Active Directory*, uma plataforma universal para gerenciar e proteger identidades. Este serviço foi escolhido devido à restrição de segurança do sistema, já que é amplamente testado e investe-se muito em sua segurança. Além disso, oferece a opção de autenticação multifator, também muito importante no contexto desse sistema [13].

O segundo serviço utilizado é o Armazenamento de Blob (ou *Blob Storage*), que permite o armazenamento seguro e escalonável para dados não estruturados. Esse serviço

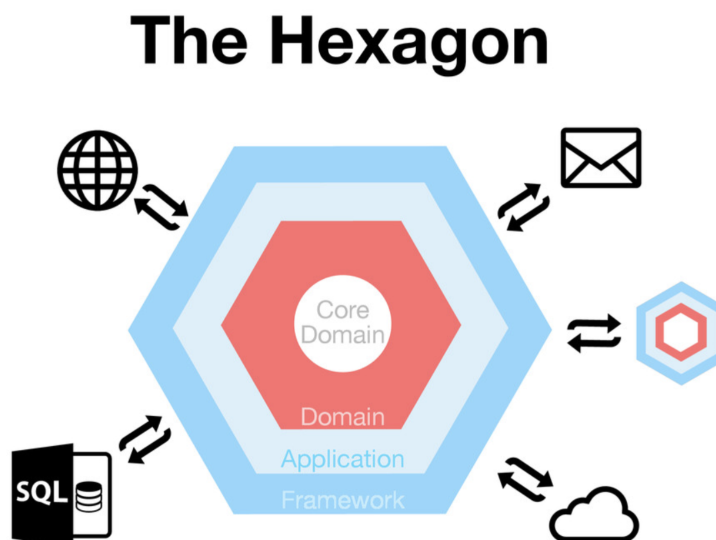

Figura 7 – Ilustração da arquitetura hexagonal.

Fonte: [11]

é utilizado para o armazenamento de fotos de passaportes dos usuários, necessário para comprovar suas identidades. Como o passaporte é um documento altamente confidencial, optou-se por utilizar um serviço bem consolidado como esse a fim de evitar vazamentos ou problemas de segurança [14].

O terceiro serviço é o *Azure Virtual Machines*, serviço de Máquinas Virtuais para computação na nuvem, onde é hospedada a aplicação. Como todo o projeto foi desenvolvido de maneira agnóstica ao ambiente em que é executado, esse serviço não influenciou decisões durante o desenvolvimento e portanto não será detalhado mais profundamente.

5.2 Implementação e Resultados

5.2.1 Azure Active Directory

Para integrar o sistema da *Azure Active Directory* à aplicação *web*, utilizou-se a biblioteca Ueberauth[1], que auxilia no processamento e validação dos *callbacks*, de forma que a aplicação pode confiar que as requisições recebidas têm realmente origem da *Azure* e possui as credenciais corretas.

O fluxo tem inicio com a aplicação redirecionando o usuário a uma página controlada pelo serviço de autenticação, que irá requisitar as credenciais do usuário juntamente com uma validação de segundo fator via SMS. Assim que o usuário termina o processo de autenticação, a *Azure* envia uma requisição ao servidor contendo um *token* que identifica

[1] https://github.com/ueberauth/ueberauth

a sessão iniciada pelo usuário.

O *backend* então redireciona o usuário à página inicial da aplicação, fornecendo a ele esse *token*, que em nenhum momento é armazenado no servidor. A partir desse momento, o usuário deve enviar esse *token* em toda requisição ao servidor, que valida sua autenticidade e identifica o usuário.

O fluxo descrito é apresentado na Figura 8, onde se pode também perceber que as informações pessoais não ficam armazenadas no banco de dados da aplicação, o que significa que apenas o usuário consegue acessar suas próprias informações pessoais.

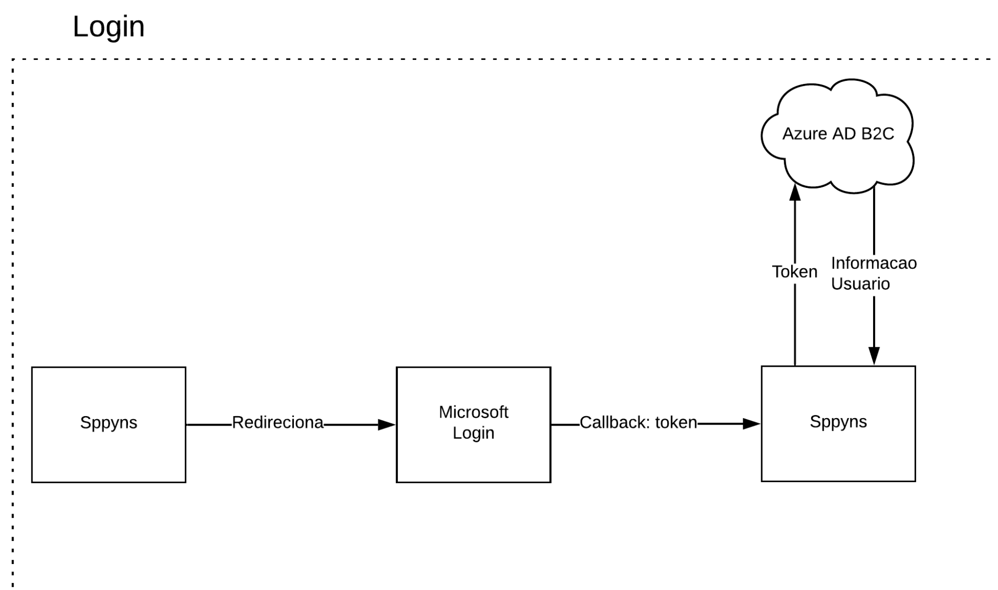

Figura 8 – Fluxo de login.

A criação de conta acontece de forma muito semelhante, já que assim que um novo *token* é recebido, verifica-se a existência de um usuário cadastrado no banco de dados que esteja associado a este *token* e, caso não esteja, um novo usuário é criado, armazenando apenas o *email* e o nome do usuário para envio de notificações da plataforma, além do identificador único proveniente da Microsoft. Esta sequência é apresentada na Figura 9, onde nota-se que ao criar um usuário na plataforma há também uma interação com o *Sppyns Wallet*, que criará uma carteira associada a esse novo usuário, permitindo o recebimento de criptomoedas.

5.2.2 Azure Blob Storage

A interação com esse serviço requer uma chave secreta, de forma que apenas a aplicação, na qual a chave está configurada, consegue interagir com o serviço de armazena-

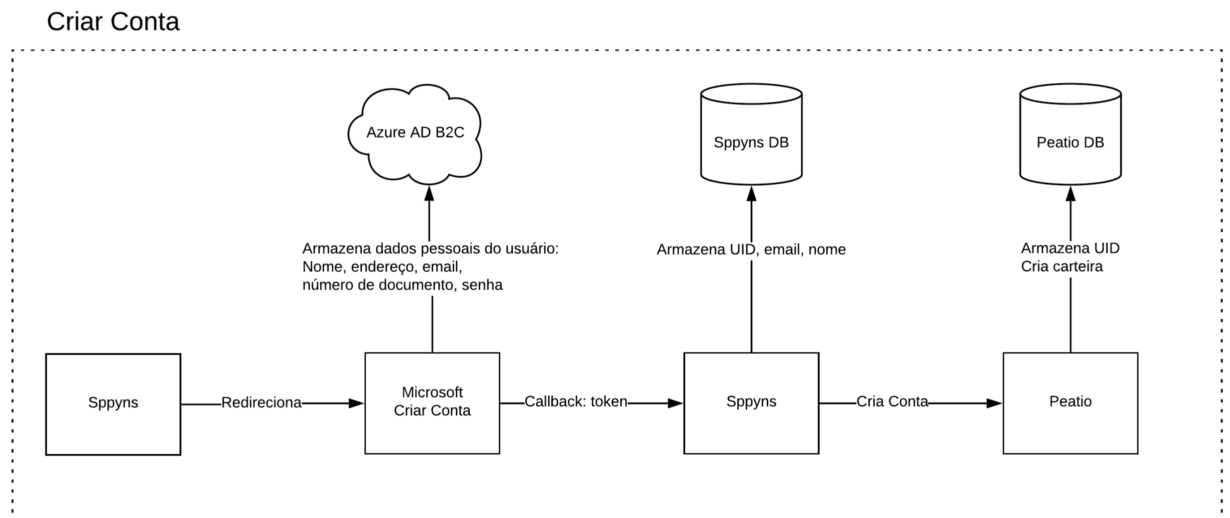

Figura 9 – Fluxo de criação de conta.

mento. Isso significa que apenas entregar um *link* para o usuário não seria suficiente, já que este não tem acesso direto.

Para resolver esse problema sem deteriorar a segurança do sistema, executa-se os seguintes passos:

1. Quando o usuário envia a foto de seu passaporte, o sistema a envia ao *Blob Storage* utilizando sua API HTTPS;

2. O resultado é um *link*, que é então armazenado no banco de dados;

3. Assim que o usuário envia uma requisição para receber o passaporte, o servidor utiliza o *link* armazenado para baixar o documento, armazenando-o localmente com um nome gerado de forma aleatória, usando um gerador de número pseudo-aleatório criptograficamente seguro;

4. A *URL* para esse arquivo é então enviado ao usuário, que tem 20 segundos para abrir o documento, que após esse período é deletado;

Dessa forma, a única forma de um terceiro acessar o documento seria interceptando o *link* enviado ao usuário, o que não é possível por se utilizar o protocolo HTTPS, ou adivinhar o nome do arquivo gerado, o que é computacionalmente impossível em um tempo tão curto quanto 20 segundos.

O fluxo de envio e posterior requisição de passaporte visando a verificação de identidade é apresentado na Figura 10 a seguir.

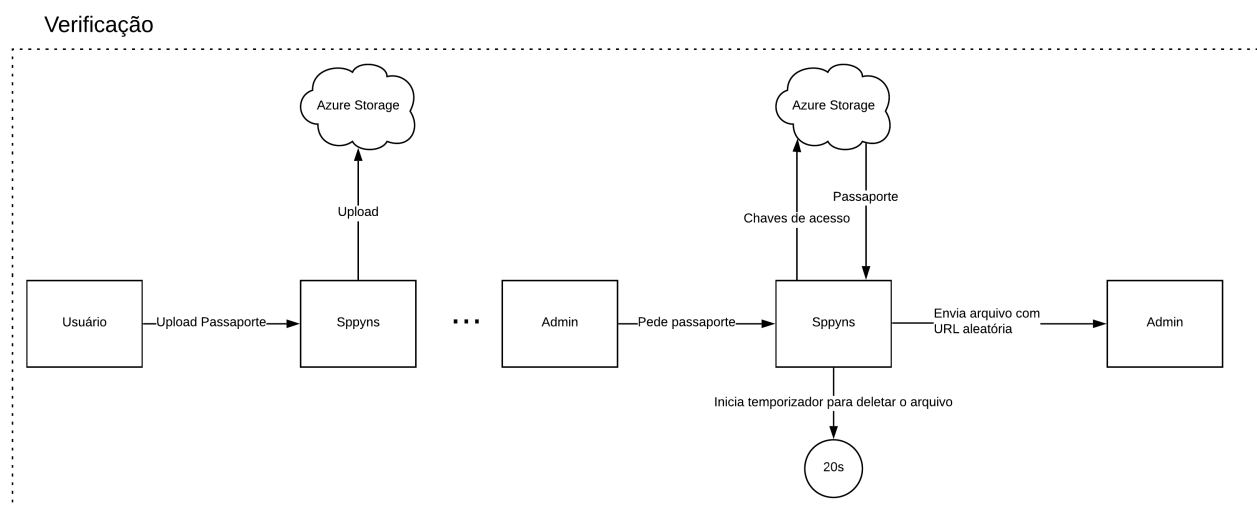

Figura 10 – Fluxo de envio de passaporte e posterior verificação de identidade.

5.2.3 Sppyns Wallet

5.2.3.1 Arquitetura Hexagonal

Como mencionado anteriormente, o *Sppyns Wallet* é o serviço responsável por gerenciar as carteiras dos usuários e interagir com a *blockchain* de forma geral. É um serviço essencial, pois a aplicação depende dele para realizar as transações, saber o saldo dos usuários e seus endereços de depósito.

Por esses motivos, realizar testes locais no ambiente de desenvolvimento ou nos ambientes de testes e *Quality Assurance* da empresa sem a presença desse serviço seria muito limitada. Semelhantemente, a realização de testes automatizados seria impossível sem a presença desse serviço, o que impediria o uso do Desenvolvimento Guiado por testes, utilizado durante o desenvolvimento da aplicação.

Por outro lado, é um serviço bastante complexo que interage diretamente com a *blockchain*, o que é pesado e custoso, além do fato de que todas as transações lá realizadas são eternas, o que não é desejável para um ambiente de testes invariante no tempo.

A Arquitetura Hexagonal resolve esse problema ao fazer com que a aplicação não dependa diretamente deste serviço, mas sim de uma interface que esse serviço implementa. Assim, é possível trocar a implementação verdadeira por um serviço que simplesmente retorna dados falsos e configuráveis, o que empodera o teste de diferentes cenários com grande facilidade. Este serviço falso é conhecido na literatura como *Mock*, um objeto que simula o comportamento de um serviço real, porém de forma controlada. [15]

Além disso, caso no futuro deseje-se utilizar um serviço diferente para interagir com a *blockchain*, a migração seria simples, bastando que o novo serviço esteja em conformidade

com a interface existente.

5.2.3.2 Segurança

O *Sppyns Wallet* é o serviço responsável por realmente fazer transações e movimentar dinheiro, o que significa que é fundamental que seja seguro. Existem dois níveis de segurança necessários na comunicação com este componente do sistema.

Primeiramente, já que o sistema foi desenvolvido de forma agnóstica ao ambiente em que se encontra, utiliza-se de Criptografia Assimétrica para tornar segura a comunicação entre o *Sppyns Wallet* e qualquer outro sistema, de forma que esse serviço apenas se comunicará com requisições cuja assinatura reflete uma das chaves públicas autorizadas.

Essa abordagem traz diversos benefícios, entre eles:

1. Controle de permissões, já que apenas aplicações registradas previamente conseguem se comunicar com o serviço;
2. Confidencialidade, já que torna-se impossível interceptar a comunicação por estar criptografada;
3. Integridade, pois todas as requisições são assinadas pelo serviço que a realiza, o que significa que a alteração de qualquer informação dentro da requisição invalidaria a assinatura.

Além disso, existe uma camada de segurança a nível de infraestrutura, pois o *Sppyns Wallet* encontra-se numa rede *Docker* interna no servidor, protegida por um *Firewall*, de forma que apenas outros componentes com acesso a essa Rede Interna, como é o caso do *Backend*, conseguem se comunicar com esse serviço.

5.2.3.3 Operações

As operações de Investir e Resgatar constituem a base da plataforma, já que é através delas que os usuários investem nos fundos e posteriormente resgatam seus lucros. A sequência de operações executadas para isso acontecer é apresentada na Figura 11, onde há também alguns passos extras de segurança que serão abordados no Capítulo 8.

Outra operação que acontece por meio da interação com o *Sppyns Wallet* é o de retirar dinheiro da plataforma, enviando para outra conta qualquer na *blockchain*. Esse fluxo é retratado na Figura 12.

A última operação é a de Depósito, que acontece de forma um pouco diferente, já que o usuário utiliza algum outro sistema onde suas criptomoedas estão armazenadas para enviá-las para a plataforma Sppyns. O *Sppyns Wallet* é responsável por ativamente

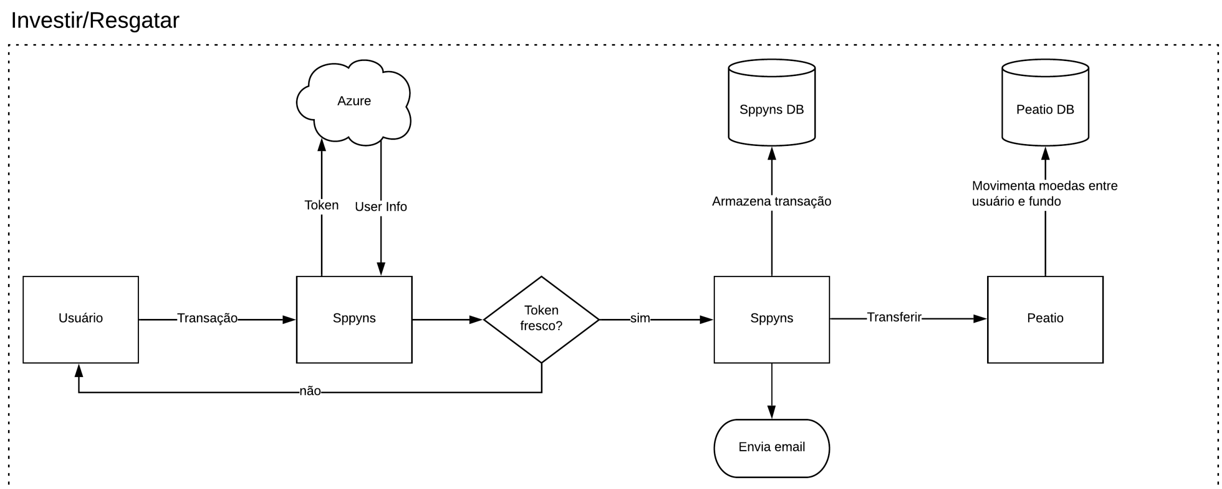

Figura 11 – Fluxo de investir e resgatar.

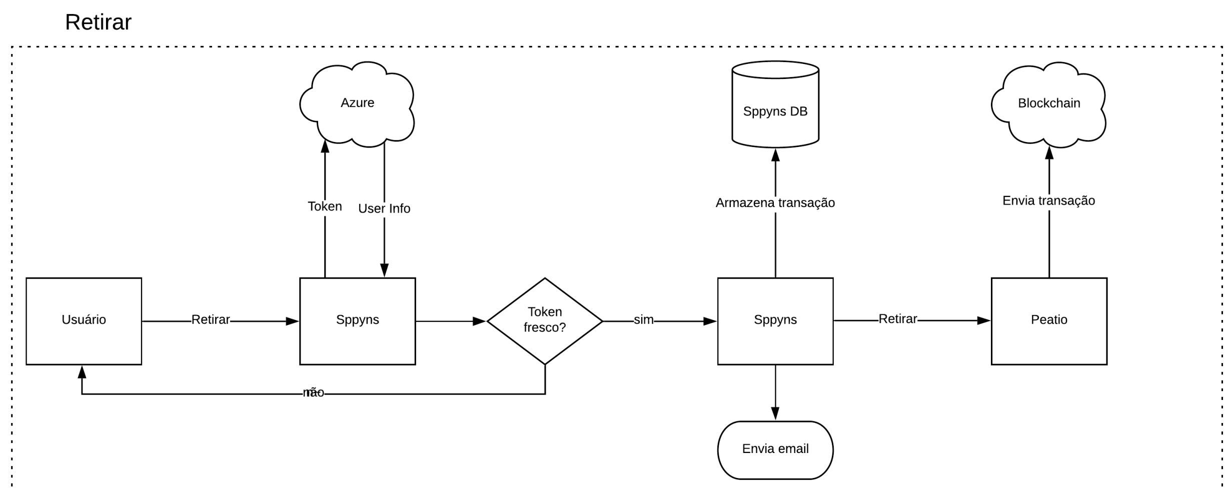

Figura 12 – Fluxo de retirar dinheiro da plataforma.

procurar novos depósitos com destino a alguma carteira por ele controlada e, assim que um novo depósito é identificado, uma requisição é enviada para o Backend notificando-o desse novo depósito, como mostra a Figura 13.

Nesse caso, é responsabilidade do Backend verificar a validade da notificação por meio da assinatura na mensagem e, então, notificar o usuário por meio de um *email* que a transação foi completada com sucesso e já é possível começar a investir na plataforma.

5.2.4 Exchanges

Para a integração com ambas as *Exchanges* suportadas pela plataforma, adotou-se a mesma abordagem de Arquitetura Hexagonal discutida anteriormente, trazendo os mesmos

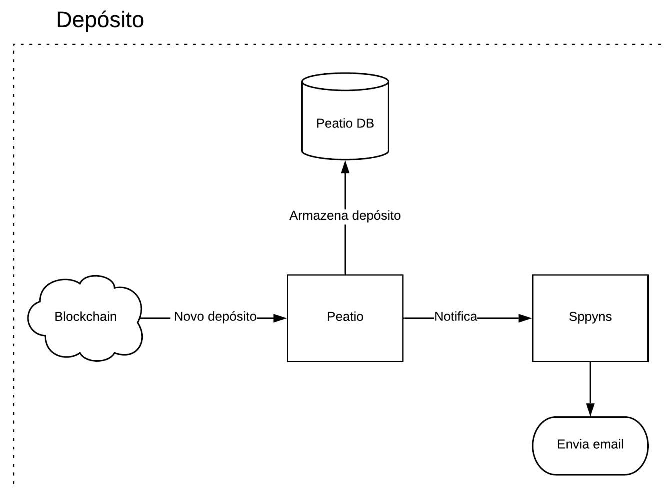

Figura 13 – Fluxo de notificação de depósito.

benefícios de facilitar os testes e aumentar a flexibilidade.

Já existia uma biblioteca para facilitar a interação com a API da *Binance*, porém se mostrou incompleta para o escopo da aplicação, sendo necessário modificá-la para acomodar as funcionalidades requeridas[2]. Já para a *Deribit*, não havia nenhuma biblioteca para a linguagem *Elixir* e portanto foi criada uma biblioteca de código aberto para facilitar a integração[3].

Como cada fundo possui diferentes contas em diferentes sistemas, salva-se no banco de dados as chaves de acesso para a API de cada fundo. Como medida de segurança, essas chaves são armazenadas de forma criptografada, além de haver restrições de endereços IP na configuração das próprias Exchanges, onde apenas os endereços previamente cadastrados têm autorização para interagir com a API.

A principal funcionalidade dessa integração é obter o saldo dos fundos visando avaliá-los, operação que será melhor aprofundada no próximo capítulo.

[2] https://github.com/gabrielpra1/binance.ex
[3] https://github.com/gabrielpra1/deribit-elixir

6 Medição de Desempenho

A Medição de Desempenho, tanto de fundos individuais, quanto da carteira de investimentos de cada usuário, é peça fundamental de uma plataforma de investimentos. O cálculo da performance é realizado utilizando dados provenientes da medição de uma ou mais variáveis.

No caso da plataforma desenvolvida, utiliza-se as APIs das Exchanges em que os fundos operam para obter a quantidade de ativos que os compõem. Todos os ativos são então convertidos para uma moeda base, geralmente *BTC* ou *USDT*, para que possam ser facilmente comparados.

Utiliza-se também as APIs para determinar qual a composição de cada fundo, informação disponibilizada publicamente através de um gráfico como o mostrado na figura 14, buscando facilitar a análise e decisão dos investidores.

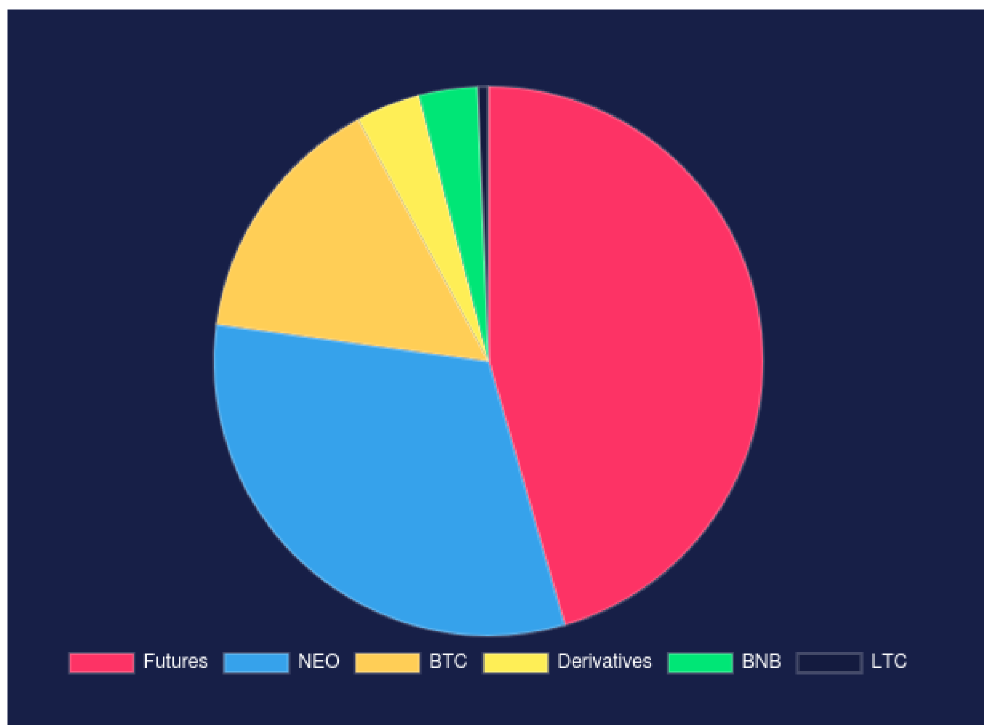

Figura 14 – Gráfico de composição de um fundo.

6.1 Aspectos Conceituais e Tecnologias

6.1.1 Cron Jobs

Cron é uma ferramenta de propósito geral de sistemas operacionais *Unix* para o gerenciamento e agendamento de tarefas, que permite configurar tarefas para ocorrerem com uma certa frequência ou em determinados momentos. Tarefas controladas pelo *cron* são comumente chamadas de *Cron Jobs*.

A maioria das linguagens de programação modernas possuem bibliotecas que imitam o comportamento dessa ferramenta de forma independente do sistema operacional, ou que dependem do *cron* para agendar tarefas.

6.1.2 Máquina Virtual Erlang

A máquina virtual *Erlang* (**BEAM**) é a máquina virtual em que se executa códigos *Elixir* e *Erlang*. Essa máquina virtual emula processos de sistemas operacionais de forma muito eficiente, enquanto mantém todos os benefícios de separação de processos. A comunicação entre processos se dá por meio de troca de mensagens assíncronas [16].

6.1.3 GenServer

Um *GenServer (Generic Server Process)* na linguagem de programação *Elixir* é uma abstração sobre um processo da máquina virtual *Erlang*, usado para manter estado, executar código assincronamente ou executar um *loop*. Devido à natureza dos processos na BEAM, milhares de *GenServers* podem ser executados paralelamente, aprimorando a escalabilidade da aplicação [17].

6.1.4 Bitcoin

Bitcoin (símbolo BTC) é a criptomoeda descentralizada baseada na tecnologia *blockchain* mais conhecida, com o propósito de revolucionar o mercado financeiro através de métodos inovadores de pagamento [18].

6.1.5 Tether

Tether (símbolo USDT) é também uma criptomoeda descentralizada baseada na tecnologia *blockchain*, porém em vez de possuir sua própria *blockchain*, utiliza a *Ethereum*. A diferença dessa moeda é a estabilidade, já que cada USDT é apoiado por um Dólar Americano, de forma que 1 USDT é sempre avaliado em 1 USD.

6.1.6 Fundo na Sppyns

A plataforma Sppyns contém dois tipos distintos de Fundos de Investimentos. Além das diferenças de lógica de negócio, diferem apenas nas criptomoedas que se baseiam e na forma de avaliação.

Todos os fundos tem como aspecto base os *tokens*. Um *token* representa uma parte do fundo, de forma que o ato de investir é na verdade a compra de *tokens* de um fundo. O usuário que possui esses *tokens* é dono de parte dos ativos deste fundo, que varia de valor ao longo do tempo. Portanto, ao avaliar um fundo, a informação mais importante é o **valor do token**.

6.1.7 CAS

Um dos tipos de fundos são chamados de **Crypto Asset Stack (CAS)**, que são constituídos por um conjunto de criptoativos, necessariamente negociados nas duas *exchanges* suportadas pela plataforma.

Devido a essa característica, esses tipos de fundo são avaliados de forma automática, seguindo o processo melhor detalhado na seção de 6.2, onde se apresenta os detalhes de implementação.

Esse tipo de fundo usa como base o BTC, já que é a maior e mais importante criptomoeda da atualidade, servindo de base de comparação para investimentos puramente voltados aos criptoativos.

6.1.8 TAS

O outro tipo de fundo é chamado de **Tokenized Asset Stack (TAS)**. Trata-se de um fundo mais geral, representando qualquer ativo que pode ser tokenizado, como empresas ou fundos imobiliários. Como esses fundos podem ser compostos de ativos de valor mais subjetivo e de difícil mensuração, a avaliação ocorre de forma manual, a uma frequência mínima de uma vez por mês. Para tanto, disponibiliza-se uma interface gráfica para os administradores do sistema, apresentada na Figura 15.

Para esses fundos, a alta flutuação do BTC é indesejável, utilizando-se, então, o USDT como moeda base.

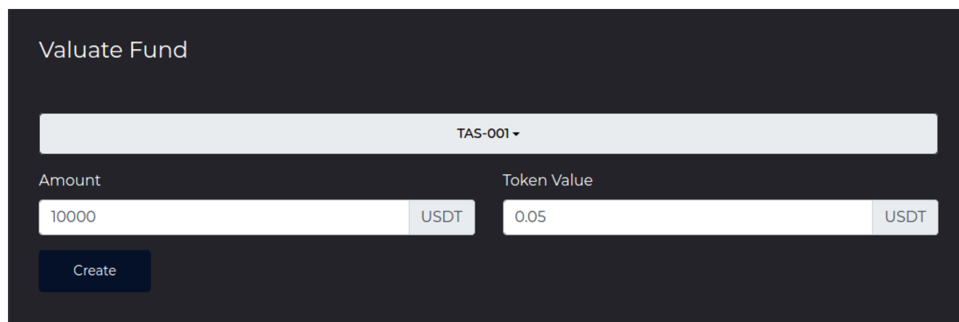

Figura 15 – Interface para criação manual de avaliações.

6.2 Implementação e Resultados

6.2.1 Avaliação automática

Para avaliar os CAS automaticamente, utiliza-se a biblioteca *quantum-core*[1], que oferece funcionalidade semelhante ao *cron* tradicional, inclusive utilizando a mesma sintaxe, para gerenciar tarefas que são executadas a cada minuto.

Para avaliar um fundo, faz-se uma requisição para cada *Exchange* em que tal fundo está registrado, obtendo a quantidade de cada ativo em sua custódia. Então, converte-se todos esses ativos para a moeda base do fundo (BTC no caso dos CAS), utilizando a API pública da *Binance* para obter todas as relações de preço entre as moedas negociadas.

Tendo o valor total atual de um fundo, calcula-se o número de *tokens* indisponíveis do fundo, ou seja, o número de *tokens* que todos os investidores possuem somados. Finalmente, o valor do *token* é a divisão entre o valor total e o número de *tokens*.

Por se tratar de um processo que interage com sistemas externos e que acontece com alta frequência, a possibilidade de falhas é alta. Para lidar de forma mais robusta com eventuais falhas e tornar a frequência de avaliação mais estável, avalia-se cada fundo em um processo assíncrono independente na *BEAM*, de forma que um erro durante a avaliação de um fundo não impacte os demais.

No momento de escrita deste documento, o sistema já acumula mais de **1 milhão** de avaliações de fundos, o que traz também a necessidade de otimização das consultas necessárias para calcular os dados apresentados nas seções posteriores.

6.2.2 Consistência nas Avaliações

Como descrito anteriormente, a avaliação leva em conta o valor dos ativos nas *Exchanges* e o número de *tokens* investidos, o que pode levar a inconsistências durante a movimentação de dinheiro entre a Sppyns e as *Exchanges*. Isso acontece pois assim que

[1] https://github.com/quantum-elixir/quantum-core

um usuário investe, o dinheiro é transferido para a conta interna do fundo na plataforma, quando o seu gerente então decidirá para qual das *Exchanges* enviará essa quantia.

Devido à natureza da *blockchain*, as transações podem levar horas para serem confirmadas, o que faz com que o dinheiro não seja considerado nem como dentro da conta interna do fundo nem como nas *Exchanges*, dando a ilusão de que o fundo desvalorizou. Essa inconsistência é bastante grave, pois uma falsa desvalorização do fundo significa que os investidores estariam pagando menos que o valor real do *token*, trazendo uma perda de dinheiro para plataforma e para o fundo. Como as avaliações acontecem a todo minuto, é importante que não exista nenhum momento de inconsistência nesse fluxo.

Para resolver esse problema, desenvolveu-se um *GenServer* para rastrear essas transações, de forma que cada nova transação gera um novo *GenServer* associado, que vive até completar sua tarefa. Esse processo tem duas fases: primeiro, espera-se a transação entrar na *blockchain*, momento em que recebe um identificador; então, checa-se repetidamente, a uma frequência maior que a frequência em que os fundos são avaliados, se a transação já foi identificada na *Exchange* de destino. Assim que a transação é identificada na *Exchange*, é marcada como completamente transferida, fazendo com que o sistema de avaliação pare de compensar a falta dessa quantia e garantindo a consistência durante todo o processo.

6.2.3 Desempenho de um Fundo

Como descrito anteriormente, o desempenho de um fundo está atrelada diretamente ao valor de seu *token*. Sendo assim, o gráfico apresentado na Figura 16 é calculado obtendo o valor do *token* em diversos pontos no tempo. Devido ao alto número de pontos, o cálculo do gráfico foi otimizado para variar a granularidade de acordo com o período escolhido, de forma a manter o número de pontos aproximadamente constante, independente do período em que se deseja observar a performance.

6.2.4 Desempenho de um Usuário

O cálculo do desempenho de um usuário é mais complexo, pois depende de diversas transações executadas em momentos diferentes e em fundos diferentes. O gráfico da Figura 17 mostra um exemplo onde o usuário possui investimentos em dois fundos diferentes, calculando o desempenho do usuário baseado em suas compras e vendas, assim como na variação do valor dos fundos ao longo do tempo.

Outras informações gerais também são calculadas, como mostra a Figura 18, onde apresenta-se o desempenho do usuário em números para períodos fixos, a distribuição de alocação de investimentos e o valor de sua carteira atual.

O dado conhecido como *Profit and Loss* também é bastante importante ao analisar uma carteira de investimento, representando qual foi o lucro (ou prejuízo) bruto de um

Figura 16 – Gráfico de performance de um fundo.

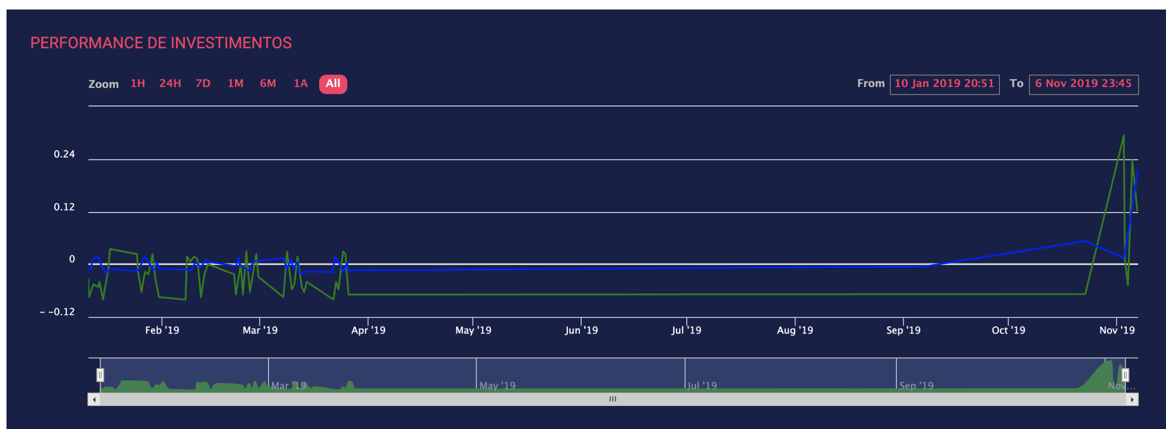

Figura 17 – Gráfico de performance de um usuário.

determinado investimento. A tabela da Figura 19 apresenta essa informação, assim como o conjunto de transações que trouxeram esse resultado.

Figura 18 – Exemplo de informações na tela de Portfolio.

ATIVOS EM CRYPTO

CAS	VALOR INVESTIDO (BTC)	TOTAL DE TOKENS	VALOR DO TOKEN ATUAL	PL (BTC)	PL (%)
CAS-001	9.00000000	5.90150701	1.52000000	2.97029065	33.00%

Id	Quantidade (BTC)	Montante após taxas (BTC)	Tokens	Valor de Token da Ordem (BTC)	Tipo	Data / Hora
7	0.20000000	0.19800000	0.20625000	0.96000000	INVEST	6/Nov/2019 18:51:54
6	2.00000000	2.00000000	2.00000000	1.00000000	REDEEM	2/Nov/2019 20:51:53
5	3.70000000	3.66300000	3.66300000	1.00000000	INVEST	1/Nov/2019 20:51:53
4	3.00000000	2.97000000	2.97000000	1.00000000	INVEST	22/Oct/2019 21:51:53
3	1.00000000	1.00000000	1.00000000	1.00000000	REDEEM	6/Aug/2019 21:51:53
2	1.50000000	1.48500000	1.48500000	1.00000000	INVEST	6/Jul/2019 21:51:53
1	0.60000000	0.59400000	0.57725700	1.02900439	INVEST	6/Nov/2018 20:51:44

CAS	VALOR INVESTIDO (BTC)	TOTAL DE TOKENS	VALOR DO TOKEN ATUAL	PL (BTC)	PL (%)
CAS-002	0.80000000	0.54365385	1.50000000	0.21548077	26.94%

Figura 19 – Exemplo de informações na tela de Portfolio.

7 Ethereum

Um dos grandes problemas de aplicações financeiras tradicionais é centralização do controle, requerendo confiança total nas instituições que fazem parte do processo financeiro. Ao utilizar a plataforma *Ethereum* como base da aplicação desenvolvida, busca-se resolver esse problema ao descentralizar o controle dos investimentos.

7.1 Aspectos Conceituais e Tecnologias

7.1.1 Blockchain

A *blockchain*, de forma simplificada, é uma lista de dados, chamados de blocos, que são conectados através de técnicas criptográficas. A tecnologia foi inventada por Satoshi Nakamoto (um pseudônimo, a verdadeira pessoa por trás desta tecnologia ainda não foi identificada) em 2008 [19] [20].

Trata-se de uma tecnologia de registro distribuído, o que significa que não existe uma entidade central que possui controle sobre esses dados. Essa característica é proveniente dos algoritmos de consenso, parte fundamental da *blockchain*. Os algoritmos de consenso permitem que uma rede de pessoas (ou computadores) tomem decisões conjuntas sem a necessidade de existir confiança entre os integrantes da rede. Nesse modelo, as decisões são tomadas de forma justa, chegando em acordos que satisfaçam a maioria [21].

Entre as características da *blockchain* ainda se encontram a resistência a modificações e o armazenamento de transações entre duas partes de forma pública, eficiente, verificável e permanente.

Devido a propriedades criptográficas, assim que um dado é gravado na *blockchain*, não pode ser alterado sem que todos os blocos subsequentes também o sejam. Como qualquer alteração do registro requer consenso entre a maioria das entidades da rede, torna-se muito difícil fazer esta alteração, tendo em vista que os blocos anteriores já são conhecidos e foram acordados entre todos os nodos da rede. Dessa forma, conquista-se segurança através de consenso descentralizado, assumindo que nenhuma entidade ou grupo controle mais de metade da rede *blockchain*.

7.1.2 Ethereum

A *Ethereum*, fundada por Vitalik Buterin em 2014, é uma *blockchain* de propósito geral, o que significa que é programável para realizar as mais diversas tarefas. Diferentemente da *blockchain* do *Bitcoin*, cujo único propósito é movimentar sua criptomoeda, a

moeda da *Ethereum*, chamada de *Ether* (ETH), é utilizada como dinheiro para pagar a execução de código na sua *blockchain* [22].

A *blockchain Ethereum*, portanto, pode ser vista como uma grande máquina virtual descentralizada, que além de servir como registro tem a capacidade de executar aplicações, que recebem os benefícios tradicionais da tecnologia, de confiança e descentralização.

A plataforma também introduz o conceito de **Contratos Inteligentes**, que é um programa de computador (uma série de comandos) que vive na rede *Ethereum*. Devido às características da *blockchain* descritas previamente, tem-se programas de computador confiáveis, que sempre executarão a mesma ação dadas as mesmas condições, sem a possibilidade de ficarem *offline*, fraude ou interferência externa, possibilitando a programação de contratos sem a necessidade de uma terceira parte intermediadora [23].

Apesar de muito nova, esta tecnologia vem se mostrando muito útil, permitindo criar diversos tipos de aplicações descentralizadas, como carteiras de criptomoedas, aplicações financeiras, mercados descentralizados, sistemas eleitorais, jogos e muito mais.

7.1.3 Geth

Uma rede *blockchain* é composta por um conjunto de nodos (computadores) executando o mesmo protocolo. No lançamento da *Ethereum*, três implementações do protocolo *Ethereum* foram distribuídas, sendo **Geth** a mais popular delas, desenvolvida utilizando a linguagem de programação *Go*[1] [24].

O *Geth* é, então, um programa de código aberto que implementa o protocolo *Ethereum*, permitindo que qualquer computador faça parte da rede. Um computador executando um programa como esse é chamado de *nodo*, sendo que os nodos conseguem executar transações, interagir com Contratos Inteligentes ou simplesmente verificar as transações que acontecem na rede [25].

7.1.4 Solidity

Existem quatro linguagens de programação principais suportadas pela Máquina Virtual *Ethereum*, que podem ser utilizadas para o desenvolvimento de Contratos Inteligentes. A mais popular entre elas é a *Solidity*, que é uma linguagem orientada a objetos influenciada por *JavaScript*, *Python* e *C++* [26].

A linguagem é estaticamente tipada e foi desenvolvida especificamente para a rede *Ethereum*, possuindo uma linguagem de alto nível para a construção de Contratos Inteligentes.

[1] https://golang.org/

```
pragma solidity ^0.5.0;

contract HelloWorld {
  function helloWorld() external pure returns (string memory) {
    return "Hello, World!";
  }
}
```

Figura 20 – Exemplo básico de um programa em Solidity.

Fonte: https://github.com/ethereum/solidity

7.2 Implementação e Resultados

O objetivo da integração da plataforma com a *blockchain Ethereum* é dar segurança aos investidores, já que suas transações na plataforma são registradas de forma segura, imutável e rastreável. Dessa forma, não se requer confiança total do usuário na plataforma, pois um erro ou até mesmo má fé por parte da empresa não pode fazer com que seus *tokens* desapareçam.

Para alcançar esse objetivo, desenvolveu-se, de forma terceirizada, um Contrato Inteligente em *Solidity* que representa um fundo de investimento na plataforma. Esse contrato implementa o padrão *ERC-20*, um protocolo para criação de contratos que lidam com *tokens* [27].

É responsabilidade da aplicação *Backend* então fazer o envio de um novo contrato para a *blockchain* assim que um novo fundo é criado, associando-o a este contrato. Assim que uma operação de investimento é realizada, cria-se também uma transação na *blockchain*, registrando a transferência de *tokens* entre usuário e fundo.

Dessa forma, a fonte de verdade da aplicação é de fato a *blockchain*, sendo as operações armazenadas no banco de dados interno apenas para permitir os cálculos de performance discutidos no capítulo anterior.

Essa integração segue também a Arquitetura Hexagonal, permitindo testes controlados da aplicação em diferentes cenários. Nesse caso, as operações implementadas por esse serviço são:

1. O *deploy* de novos contratos representando um novo fundo;
2. As operações de transferência de *tokens*;
3. A alteração de informações do fundo (como por exemplo risco, objetivos e taxas) em contratos já existentes na *blockchain*;
4. A leitura e processamento de transações da *blockchain* para o contexto da plataforma.

A Figura 21 mostra um exemplo de um contrato representando um fundo, de nome *Jungsoft GmbH* e que possui **1,000,000** tokens, dos quais **999,987,009866** ainda estão no fundo, enquanto os outros são controlados por investidores.

Figura 21 – Exemplo de um contrato representando um Fundo.

Fonte: Etherscan

A Figura 22 mostra um exemplo de uma transação de investimento, onde **0,9333** tokens são transferidos para uma conta de um usuário da plataforma.

Essas figuras demonstram os pontos anteriormente destacados, já que cada transação ficará registrada para sempre na *blockchain*, sendo publicamente acessível. Ao se visitar uma conta de um usuário, é possível observar a quantidade de *tokens* que este possui, fato que não pode ser manipulado. Essa propriedade é extremamente importante, pois no caso de um Fundo do tipo TAS, esses *tokens* podem representar, por exemplo, o controle de uma empresa.

7.2.1 Carteiras multi-assinatura

Para cada fundo criado na plataforma, cria-se uma nova carteira *Ethereum*, que será a dona do *smart contract* que representa o fundo. Ou seja, apenas esta carteira tem autorização para realizar operações neste contrato. Como o processo de investimento é totalmente automatizado, a carteira precisa ficar acessível ao sistema, que teoricamente teria o poder de transferir *tokens* para qualquer endereço. Levando em consideração que

Figura 22 – Exemplo de uma transação de investimento.

Fonte: Etherscan

os *tokens* são bens digitais potencialmente muito valiosos, utiliza-se de uma carteira multi-assinatura para limitar o poder do sistema através de uma solução diretamente na *blockchain*, o que significa que não existe a possibilidade desse processo ser manipulado e não há a necessidade de uma terceira parte envolvida.

Cria-se uma carteira multi-assinatura para cada fundo que opta por essa funcionalidade. Tal carteira nada mais é que um contrato inteligente em que vários donos são definidos, assim como um número necessário de confirmações para que uma ação possa ser executada. Neste caso, dois donos são atribuídos: a carteira relacionada ao fundo, controlada pela Sppyns, e uma carteira controlada pelo gerente do fundo, a qual o sistema não possui acesso. Define-se, também, que todos os dois donos precisam aprovar uma ação para que esta seja executada.

Dessa maneira, o dono do contrato ERC-20 do fundo não é mais a carteira controlada pelo sistema da Sppyns, mas sim o contrato multi-assinatura, de forma que quando um usuário investe em um fundo que possui essa funcionalidade, o sistema submete a ação de transferência de *tokens* à carteira multi-assinatura. Tal transferência só será de fato realizada quando o gerente do fundo aprová-la, totalizando o número necessário de aprovações.

8 Segurança

A segurança é um aspecto muito importante em aplicações *web* em geral, por estarem disponíveis a todo o mundo. Para aplicações financeiras, isso torna-se ainda mais relevante, já que falhas de segurança podem levar à perda direta de dinheiro. Além disso, como ataques maliciosos podem trazer um retorno financeiro, a motivação dos invasores é ainda maior. Consequentemente, todo o planejamento e desenvolvimento da aplicação levou em conta esse fator, buscando seguir as melhores práticas e evitar ao máximo a criação de vulnerabilidades no sistema.

8.1 Aspectos Conceituais e Tecnologias

8.1.1 JSON Web Token

Popularmente conhecido como **JWT**, o *JSON Web Token* é um padrão aberto de *tokens* de acesso para representação segura de afirmações (*claims*, em inglês). Por utilizar técnicas de criptografia assimétrica, é possível armazenar diversas informações dentro de um *token*, como por exemplo a data de expedição, com a segurança de que não poderão ser adulteradas.

8.1.2 Teste de Intrusão

Teste de Intrusão, ou Teste de Penetração, consiste de um ciber-ataque simulado e autorizado, que busca simular um ataque malicioso, buscando avaliar a segurança de um sistema.

O objetivo desse tipo de teste é identificar vulnerabilidades no sistema que possam vir a ser exploradas por ataques realmente maliciosos, levando a problemas como roubo de dados, indisponibilização do serviço ou fazer o sistema se comportar diferente do planejado [28].

8.1.3 Ataque de negação de serviço

Comumente chamado de **DoS**, do inglês *Denial of Service*, esse ataque consiste em esgotar os recursos computacionais ou de rede da vítima, fazendo com que o serviço alvo fique temporariamente indisponível ou, ao menos, muito lento.

Tipicamente, isso acontece através do envio de muitas requisições para a vítima do ataque, que pode acabar sendo sobrecarregada e não ser capaz de responder a requisições legítimas [29].

8.2 Implementação e Resultado

Esta seção apresenta variadas medidas de segurança tomadas ao longo do desenvolvimento da aplicação.

8.2.1 Segurança extra para movimentação de dinheiro

Os fluxos das figuras 11 e 12 mostram que existe um passo de verificação inicial antes da realização de operações de movimentação financeira. Essa verificação garante que o usuário realizou seu login nos últimos cinco minutos, caso contrário a operação é bloqueada e o usuário é obrigado a refazer seu login, o que requer a autenticação multi-fator.

Esse passo extra tem como objetivo evitar a movimentação indevida após o usuário esquecer de fazer o *logout* em um computador público ou ter seu computador particular invadido. Essa verificação torna-se possível por utilizar-se de *JWT tokens* para identificação dos usuários na plataforma.

8.2.2 Resistência a DoS

Como descrito na seção de Aspectos Conceituais, os ataques *DoS* acontecem de forma geral através do envio exagerado de requesições para um sistema. Portanto, busca-se por requisições que sejam leves de serem feitas mas que ocupem o máximo possível do poder de computação da vítima, de forma a multiplicar a efetividade do ataque.

Nesse cenário, é importante evitar que consultas muito complexas sejam feitas ao servidor, o que tornaria ataques desse tipo muito efetivos. Como expõe-se uma API *GraphQL*, o usuário pode, a princípio, realizar consultas indeterminadamente complexas, principalmente em grafos cíclicos, onde teoricamente poderia se requisitar uma quantidade infinita de dados ao se repetir os mesmos campos em uma mesma requisição, como mostra o exemplo da Figura 23.

A biblioteca utilizada para construir o servidor *GraphQL* é a *Absinthe*[1], que suporta a especificação de um limite máximo de complexidade nas consultas, ferramenta que foi utilizada para minimizar a possibilidade de ataques de negação de serviço. A complexidade padrão é calculada somando 1 para cada campo requisitado, sendo utilizado 250 como complexidade máxima para essa aplicação, valor determinado após análise das consultas mais complexas presentes no sistema.

Além disso, identificou-se as consultas mais custosas para o servidor, geralmente por necessitarem se comunicar com serviços externos ou por realizar muitos cálculos envolvendo o banco de dados, e aumentou-se suas complexidades, de forma que não seja possível explorar a requisição de poucos porém complexos campos.

[1] https://github.com/absinthe-graphql/absinthe

```
{
  funds {
    manager {
      fund {
        manager {
          fund {
            manager {
              fund {
                manager {
                  fund {
                    name
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Figura 23 – Exemplo de uma consulta GraphQL em um grafo cíclico.

8.2.3 Teste de Intrusão Interno

Após o término do desenvolvimento de todas as funcionalidades requeridas pela aplicação, realizou-se um Teste de Intrusão Interno, em que se buscou encontrar vulnerabilidades no sistema, analisando todas as consultas possíveis na API visando descobrir a possibilidade de acesso de informações confidenciais.

A princípio, havia uma camada de Autorização apenas a nível de requisição, de forma que ao se fazer uma determinada consulta, verifica-se o nível de acesso do usuário em questão, bloqueando ou não essa requisição. Um exemplo dessa funcionalidade é permitir requisições de buscar um determinado usuário por ID apenas para usuários *admins* ou para o próprio usuário.

Essa estratégia de autorização é bastante importante e assegura a grande maioria das informações confidenciais. Porém, como a linguagem *GraphQL* provê uma API do tipo grafo, é possível através de uma requisição acessar diversos dados associados, o que podem trazer problemas menos óbvios de segurança.

O exemplo da Figura 24 mostra essa estratégia de autorização, que atua ao se explicitamente tentar buscar um determinado usuário. Entretanto, é possível buscar as informações de um usuário de forma indireta, como mostra a Figura 25, onde se faz uma

consulta pública por um fundo mas, através do gerente do fundo, chega-se nas informações de um usuário. Para este caso, foi necessário implementar também autorização à nível de objetos, de forma que cada objeto retornado pela consulta é verificado, impossibilitando o acesso de informações confidenciais.

```
{
  user(id: 1) {
    name
  }
}
```

```
{
  "data": {
    "user": null
  },
  "errors": [
    {
      "locations": [
        {
          "column": 0,
          "line": 2
        }
      ],
      "message": "unauthorized",
      "path": [
        "user"
      ]
    }
  ]
}
```

Figura 24 – Exemplo de uma consulta bloqueada pelo sistema de autorização à nível de requisição.

```
{
  fund(id: 1) {
    manager {
      user {
        name
      }
    }
  }
}
```

```
{
  "data": {
    "fund": null
  },
  "errors": [
    {
      "locations": [
        {
          "column": 0,
          "line": 2
        }
      ],
      "message": "Not authorized to access object user",
      "path": [
        "fund"
      ]
    }
  ]
}
```

Figura 25 – Exemplo de uma consulta bloqueada pelo sistema de autorização à nível de objeto.

Além da verificação manual, também foi utilizado o *Sobelow*[2], uma ferramenta de análise de segurança estática para a *framework Phoenix*, que procura vulnerabilidades

[2] https://github.com/nccgroup/sobelow

comuns em aplicações desse tipo. Apesar de alguns falso positivos, a ferramenta se mostrou bastante útil para a identificação de potenciais falhas de segurança. A Figura 26 mostra um exemplo de resultado obtido ao se executar a ferramenta.

```
Config.Secrets: Hardcoded Secret - High Confidence
File: config/db.secret.exs
Line: 6
Key: password

-----------------------------------------------

Config.HTTPS: HTTPS Not Enabled - High Confidence

-----------------------------------------------

DOS.BinToAtom: Unsafe atom interpolation - Low Confidence
File: lib/sppyns/accounts/accounts.ex
Line: 316
Function: load_user_transactions:311
Variable: currency
```

Figura 26 – Resultado textual da ferramenta Sobelow.

8.2.4 Teste de Intrusão Externo

Após os testes de intrusão interno e todas as correções necessárias, foi contratada uma equipe da McAfee[3] para realizar um Teste de Intrusão no sistema do tipo caixa preta, em que se simula um ataque sem nenhum acesso interno ao sistema, como código fonte ou documentação.

Admiravelmente, os testes não apontaram nenhuma vulnerabilidade no sistema, fato muito raro de acontecer segundo o relatório recebido. Dessa forma, considera-se o sistema seguro e pronto para ser utilizado em um ambiente de produção.

[3] https://www.mcafee.com/

9 Resultados e Perspectivas

O projeto, que já se encontra em produção, pode ser considerado um sucesso para a Jungsoft, que atendeu as expectativas do cliente e possui perspectivas de continuar trabalhando em conjunto, já com novas funcionalidades sendo planejadas.

O mesmo pode ser dito para a Sppyns, cuja base de funcionamento é o *software* desenvolvido, pelo qual já foram investidos mais de **930 mil reais** no momento da escrita desse documento, contando com **seis fundos** do tipo **CAS** e **um fundo** do tipo **TAS** ativos e mais de **150 usuários** cadastrados.

Outro indicador que mostra o potencial do projeto é a captação de investimentos externos na empresa, que já soma aproximadamente **1 milhão de reais**.

A Sppyns ainda se encontra em fase de operação assistida, sendo aceitos apenas usuários selecionados. Porém, o objetivo é tornar-se totalmente operacional durante o ano de 2020, o que certamente trará novos desafios, tanto operacionais como tecnológicos.

9.1 Próximos Passos

Diversas novas funcionalidades e melhorias para o sistema estão sendo planejadas ou já se encontram em fase inicial de desenvolvimento, destacam-se:

9.1.1 Portal de Parceiros

O Portal de Parceiros (*Partner Portal*, em inglês) é uma nova área do sistema para empresas parceiras da Sppyns, que indicarão seus clientes para investir através da plataforma, utilizando *links* de convite. Assim que um novo usuário cadastra-se na plataforma utilizando um desses *links*, o sistema o registra como associado à empresa parceira, que então poderá acompanhar os investimentos desse usuário e assisti-lo na tomada de decisões.

9.1.2 Suporte a múltiplas moedas por fundo

Como descrito no Capítulo 6, existem atualmente dois tipos de fundo: o CAS, que opera exclusivamente em *Bitcoin*, e o TAS, que opera em *Tether*. Planeja-se, porém, que cada fundo possa definir várias moedas aceitas para as operações de investir e resgatar. Essa funcionalidade dependerá da conversão de valores entre diferentes moedas em tempo real, além da provável necessidade de aplicar taxas extras relacionadas às conversões que precisarão ser realizadas pelo gerente do fundo.

9.1.3 Migração do Servidor

Devido a restrições e regras existentes na Suíça, todos os dados da aplicação deverão ser armazenados em território suíço, o que significa que o servidor e o banco de dados precisarão ser migrados.

Além disso, como o sistema de autenticação da Azure (*Azure Active Directory*) não permite definir onde armazena os dados pessoais dos clientes, será necessário alterar todo o sistema de autenticação da aplicação, de forma a garantir que os dados são armazenados em território suíço.

Referências

1 EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [S.l.]: Addison-Wesley, 2004. Citado na página 16.

2 DOCKER Overview. Disponível em: <https://docs.docker.com/engine/docker-overview/>. Acesso em: novembro, 2019. Citado na página 17.

3 GRAPHQL. Disponível em: <https://graphql.org/>. Acesso em: novembro, 2019. Citado na página 17.

4 SUPERVISOR. Disponível em: <https://hexdocs.pm/elixir/Supervisor.html>. Acesso em: março, 2020. Citado na página 20.

5 JUNG, R. *Platform and Methodology for Developing Modern Systems in Restricted Enterprise Environments, using Elixir/Erlang, Docker, CI/CD and Microservices*. Projeto de Fim de Curso — Universidade Federal de Santa Catarina, 2018. Citado na página 20.

6 OVERVIEW of Docker Compose. Disponível em: <https://docs.docker.com/compose/>. Acesso em: novembro, 2019. Citado na página 21.

7 DOCKERFILE reference. Disponível em: <https://docs.docker.com/engine/reference/builder/>. Acesso em: novembro, 2019. Citado na página 21.

8 ABOUT PostgreSQL. Disponível em: <https://www.postgresql.org/about/>. Acesso em: novembro, 2019. Citado na página 22.

9 PRá, G. *Desenvolvimento de Software embarcado para Aerogerador baseado em aerofólio cabeado*. Relatório de Estágio — Universidade Federal de Santa Catarina, 2018. Citado na página 23.

10 VUOLLET, P. *Hexagonal Architecture: What Is It and How Does It Work?* Disponível em: <https://blog.ndepend.com/hexagonal-architecture/>. Acesso em: novembro, 2019. Citado na página 24.

11 HEXAGONAL Architecture. Disponível em: <https://fideloper.com/hexagonal-architecture>. Acesso em: novembro, 2019. Citado na página 25.

12 VISãO Geral Azure. Disponível em: <https://azure.microsoft.com/pt-br/overview/>. Acesso em: novembro, 2019. Citado na página 24.

13 AZURE Active Directory. Disponível em: <https://azure.microsoft.com/pt-br/services/active-directory/>. Acesso em: novembro, 2019. Citado na página 24.

14 ARMAZENAMENTO de Blob. Disponível em: <https://azure.microsoft.com/pt-br/services/storage/blobs/>. Acesso em: novembro, 2019. Citado na página 25.

15 A Brief History of Mock Objects. Disponível em: <http://www.mockobjects.com/2009/09/brief-history-of-mock-objects.html>. Acesso em: novembro, 2019. Citado na página 28.

16 STENMAN, E. *The Erlang Runtime System*. [S.l.]: Open on GitHub, 2017. Citado na página 33.

17 GENSERVER. Disponível em: <https://hexdocs.pm/elixir/GenServer.html>. Acesso em: novembro, 2019. Citado na página 33.

18 BITCOIN. Disponível em: <https://bitcoin.org/en/>. Acesso em: novembro, 2019. Citado na página 33.

19 LEWIS, A. *The Basics of Bitcoins and Blockchains: An Introduction to Cryptocurrencies and the Technology that Powers Them*. 1. ed. [S.l.]: Mango Media, 2018. Citado na página 39.

20 THE misidentification of Satoshi Nakamoto. Disponível em: <https://theweek.com/articles/561540/misidentification-satoshi-nakamoto>. Acesso em: novembro, 2019. Citado na página 39.

21 LAMOUNIER, L. *Algoritmos de Consenso*. Disponível em: <https://101blockchains.com/pt/algoritmos-de-consenso/>. Acesso em: janeiro, 2020. Citado na página 39.

22 ETHEREUM Beginners. Disponível em: <https://ethereum.org/beginners/>. Acesso em: novembro, 2019. Citado na página 40.

23 ANTONOPOULOS, A. M.; WOOD, G. *Mastering Ethereum*. 1. ed. [S.l.]: O'Reilly, 2018. Citado na página 40.

24 WHAT is Geth? Disponível em: <https://ethgasstation.info/blog/what-is-geth/>. Acesso em: novembro, 2019. Citado na página 40.

25 GO Ethereum. Disponível em: <https://geth.ethereum.org/>. Acesso em: novembro, 2019. Citado na página 40.

26 SOLIDITY. Disponível em: <https://solidity.readthedocs.io/en/v0.5.12/>. Acesso em: novembro, 2019. Citado na página 40.

27 ERC: Token Standard 20. Disponível em: <https://github.com/ethereum/eips/issues/20>. Acesso em: novembro, 2019. Citado na página 41.

28 DOI, U. *Penetration Testing*. Disponível em: <https://www.doi.gov/ocio/customers/penetration-testing>. Acesso em: novembro, 2019. Citado na página 44.

29 NCCIC. *Understanding Denial-of-Service Attacks*. Disponível em: <https://www.us-cert.gov/ncas/tips/ST04-015>. Acesso em: novembro, 2019. Citado na página 44.