IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

RASTREABILIDADE DE EVIDÊNCIAS NO DOMÍNIO JURÍDICO

Oleksandr Sierov

Orientador
Luís Gonzaga Martins Ferreira

Dissertação apresentada
ao Instituto Politécnico do Cávado e do Ave
para obtenção do Grau de Mestre em Engenharia Informática

maio, 2024

IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

RASTREABILIDADE DE EVIDÊNCIAS NO DOMÍNIO JURÍDICO

Oleksandr Sierov

Orientador

Luís Gonzaga Martins Ferreira

Dissertação apresentada
ao Instituto Politécnico do Cávado e do Ave
para obtenção do Grau de Mestre em Engenharia Informática

maio, 2024

Declaração

Nome: Oleksandr Sierov
Endereço eletrónico: oleksandrsierov@gmail.com
Título da Dissertação: Rastreabilidade de evidências no domínio jurídico.
Subtítulo da Dissertação:
Orientador: Luís Gonzaga Martins Ferreira
Coorientador:
Ano de conclusão: maio, 2024
Designação do Curso de Mestrado: Mestrado em Engenharia Informática.

Nos exemplares das Dissertações /Projetos/ Relatórios de Estágio de mestrado ou de outros trabalhos entregues para prestação de Provas Públicas, e dos quais é obrigatoriamente enviado exemplares para depósito legal, deve constar uma das seguintes declarações:

☐ É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA DISSERTAÇÃO/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

☐ É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA DISSERTAÇÃO/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

☐ DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA DISSERTAÇÃO/TRABALHO

Instituto Politécnico do Cávado e do Ave, ___/___/_____

Assinatura: _____

Rastreabilidade de Evidências no Domínio Jurídico

Resumo

O aumento constante de casos judiciais destaca a necessidade de gerir evidências de prova de uma forma segura, uma vez que a potencial manipulação de evidências é uma preocupação real. Este estudo investiga a aplicação da tecnologia *Blockchain* para registar e rastrear evidências no contexto jurídico. A *blockchain* oferece segurança adicional e uma cadeia imutável e transparente para cada evidência, desde o momento em que é registada até ao momento em que possa ser utilizada num processo judicial.

O resultado deste trabalho foi uma solução de demonstração desenvolvida na *blockchain Ethereum*, com *smart contracts* para gerir e controlar toda a cadeia de registo de evidências. Uma aplicação *web* integradora disponibiliza um conjunto de serviços (API) que garante a preservação de dados complementares das evidências numa base de dados documental. Mecanismos orientados a eventos garantem o sincronismo entre as operações sobre a base de dados e a cadeia de registo.

A validação da solução incluiu uma experimentação comparativa entre a abordagem *blockchain* e uma abordagem convencional inerente à atividade forense, sobre o sistema *Linux*. A análise concentrou-se no controlo, usabilidade, integridade, validação de qualquer tentativa de adulteração. Foram identificados critérios favoráveis na abordagem *blockchain* em comparação com a convencional.

Rastreabilidade de Evidências no Domínio Jurídico

Abstract

The constant increase in legal cases highlights the need to manage evidence in a secure way, as the potential manipulation of evidence is a real concern. This study investigates the application of Blockchain technology to record and track evidence in the legal context. Blockchain offers additional security and an immutable, transparent chain for each piece of evidence, from the moment it is recorded until the moment it can be used in a court case.

The result of this work was a demonstration solution developed on the Ethereum blockchain, with smart contracts for managing and controlling the entire chain of evidence recording. An integrating web application provides a set of services (API) that guarantees the preservation of complementary evidence data in a document database. Event-driven mechanisms ensure synchronization between operations on the database and the registration chain.

The validation of the solution included a comparative experiment between the blockchain approach and a conventional approach inherent to forensic activity, on the Linux system. The analysis focused on control, usability, integrity and validation of any tampering attempts. Favorable criteria were identified in the blockchain approach compared to the conventional one.

Índice

1 Introdução **1**

1.1 Motivação . . . 1
1.2 Objetivos . . . 2
1.3 Estrutura do documento . . . 2
1.4 Metodologia de Investigação . . . 2
1.4.1 *Design Science Research* . . . 2
1.5 Plano de Trabalho . . . 3

2 Revisão da Literatura **5**

2.1 Registo de evidências . . . 5
2.1.1 O que é uma evidência? . . . 5
2.1.2 Evidência num processo jurídico . . . 5
2.1.3 Tipos de evidências . . . 6
2.1.4 Riscos no registo e armazenamento de uma evidência num processo jurídico . . . 6
2.1.5 Recolha de uma evidência . . . 7
2.1.6 *Chain of Custody* . . . 7
2.2 Tecnologia *Blockchain* . . . 9
2.2.1 Evolução e histórico da *Blockchain* . . . 10
2.2.2 Modelos de *Blockchain* . . . 10
2.2.3 Mecanismos de Consenso . . . 11
2.2.4 Vantagens da *Blockchain* . . . 12
2.2.5 Desvantagens da *Blockchain* . . . 13
2.2.6 Riscos da *Blockchain* . . . 13
2.2.7 Smart Contracts . . . 14
2.2.8 Plataformas *Blockchain* . . . 14
2.3 Aplicação da *Blockchain* no domínio jurídico . . . 17
2.4 *Ethereum* . . . 20
2.4.1 O que são *dApps* . . . 21
2.4.2 A *Blockchain Ethereum* . . . 21
2.4.3 *Ethereum Virtual Machine* . . . 22
2.5 Desenvolvimento de *dApps* na *Ethereum* . . . 23
2.5.1 Linguagem de programação . . . 23
2.5.2 *Framework Truffle* . . . 24
2.5.3 Ganache . . . 25

2.5.4 Drizzle . . . 25

3 Análise e Modelação **27**

3.1 Objetivo . . . 27
3.2 Engenharia de Requisitos . . . 27
3.2.1 Requisitos funcionais . . . 28
3.2.2 Requisitos não funcionais . . . 30
3.3 Casos de uso e diagrama de atividades . . . 30
3.3.1 Atores . . . 31
3.3.2 Diagrama de casos de uso de evidências . . . 31
3.3.3 Diagrama de casos de uso de entidades . . . 41
3.4 Arquitetura Funcional . . . 46
3.4.1 Arquitetura Tecnológica . . . 47
3.5 Modelo de Domínio . . . 51

4 Implementação do sistema **53**

4.1 *Framework Truffle* . . . 53
4.2 *Smart Contracts* . . . 54
4.2.1 *Manager* de evidências . . . 55
4.2.2 *Manager* de entidades . . . 65
4.3 *Back-end* . . . 68
4.3.1 Ligação MongoDB . . . 69
4.3.2 Ligação com *smart contracts* . . . 70
4.3.3 Autenticação e *token* . . . 71
4.3.4 Adicionar um arquivo digital . . . 72
4.3.5 Verificar e *download* de um arquivo digital . . . 74
4.4 *Monitoring Trigger* . . . 75
4.5 *Front-End* . . . 76
4.5.1 Ligação com *smart contracts* e *MetaMask* . . . 76
4.5.2 Ligação com *back-end* . . . 80
4.5.3 Utilização do *MetaMask* . . . 80
4.5.4 Aplicação *Web* . . . 82
4.6 *Deploy* da *dApp* . . . 85
4.6.1 Configuração com uma *blockchain testnet* . . . 86
4.6.2 *Deployment* . . . 89

5 Experimentação e Análise de Resultados **93**

5.1 Experimentação em *Linux* . . . 94
5.1.1 Plataforma tecnológica de suporte . . . 94
5.1.2 Ferramentas . . . 94
5.1.3 Processo . . . 95
5.1.4 Observações . . . 100
5.2 Experimentação em *blockchain* . . . 102
5.2.1 Plataforma tecnológica de suporte . . . 102

5.2.2 Processo 103
5.2.3 Observações 112
5.3 Comparação dos resultados 112

6 Conclusão e trabalhos futuros **115**
6.1 Trabalhos futuros 115

Bibliografia **117**

Lista de Figuras

1.1 Plano de trabalho 3

2.1 *Data Acquisition Process on Computer* - INTERPOL (2019) 7
2.2 *Acquisition Process Flow Chart* - INTERPOL (2019) 8
2.3 Blockchain Ilustration - Chain (2023) 10
2.4 Funcionamento do *Proof of Work* (PoW) 12
2.5 Vantagens dos *Smart Contracts* 15
2.6 Comparação entre *Blockchains* 16
2.7 *Blockchain of custody framework* 18
2.8 *CRAB Protocol* 20
2.9 *Ethereum account* 22
2.10 Constituintes de uma transação da *Ethereum* 23

3.1 Diagrama de casos de uso (Evidências) 32
3.2 Diagrama de Atividades UC 01 33
3.3 Diagrama de Atividades UC 02 34
3.4 Fluxograma do *Trigger* 35
3.5 Diagrama de Atividades UC 03 36
3.6 Diagrama de Atividades UC 04 37
3.7 Diagrama de Atividades UC 05 38
3.8 Diagrama de Atividades UC 06 39
3.9 Diagrama de Atividades UC 07 40
3.10 Diagrama de Atividades UC 08 40
3.11 Diagrama de casos de uso (Entidades) 41
3.12 Diagrama de Atividades UC 09 42
3.13 Diagrama de Atividades UC 10 43
3.14 Diagrama de Atividades UC 11 44
3.15 Diagrama de Atividades UC 12 45
3.16 Diagrama de Atividades UC 13 45
3.17 Diagrama de Atividades UC 14 46
3.18 Arquitetura Funcional (V.1) 47
3.19 Arquitetura Funcional (V.2) 47
3.20 Arquitetura tecnológica 48
3.21 Diagrama de Classes 51

4.1 Projeto *Truffle* . . . 53
4.2 a) *MetaMask*: Contexto da conta na aplicação . . . 81
4.3 b) *MetaMask*: Contexto da conta na aplicação . . . 81
4.4 *MetaMask*: Notificação com pedido de aprovação . . . 82
4.5 *Web*: Lista de registos de evidências . . . 83
4.6 *Web*: Adicionar um registo de evidência . . . 84
4.7 *Web*: Adicionar um arquivo digital . . . 85
4.8 *Web*: Adicionar um utilizador a um registo de evidência . . . 86
4.9 *Web*: Transferir o *ownership* de um registo . . . 87
4.10 *Web*: Histórico de uma evidência . . . 88
4.11 Step-by-Step: *Deployment* da *dApp* . . . 89

5.1 *Kali Linux Setup* . . . 94
5.2 Interface *Guymager* . . . 95
5.3 *Guymager acquire image* . . . 96
5.4 *Guymager acquire image options* . . . 96
5.5 *Guymager acquire image* terminado . . . 97
5.6 Ficheiros Criados pelo *Guymager* . . . 97
5.7 *GtkHash* . . . 98
5.8 *Mount* do volume do arquivo *.dd* . . . 99
5.9 Conteúdo do arquivo *.dd mounted* . . . 100
5.10 *Unmount* do volume do arquivo *.dd* . . . 100
5.11 *GtkHash* depois da manipulação da imagem recolhida . . . 101
5.12 Informações do ficheiro . . . 101
5.13 Aceder a aplicação . . . 103
5.14 *Login* no *MetaMask* concluído . . . 104
5.15 Lista de evidências no ponto inicial . . . 104
5.16 Adicionar um registo de evidência . . . 105
5.17 Notificação *MetaMask* para aprovação da transação . . . 105
5.18 Lista de evidências depois de adicionar uma evidência . . . 106
5.19 *Login* via utilizador e *password* . . . 106
5.20 View de *Upload* de arquivos . . . 107
5.21 View de *Upload* de arquivos com o arquivo selecionado . . . 107
5.22 Notificação *MetaMask* aprovação de adição de arquivo . . . 108
5.23 Ícone para aceder ao histórico . . . 108
5.24 Histórico do registo . . . 109
5.25 Meta-dados do arquivo adicionado . . . 109
5.26 Dados do arquivo adicionado . . . 110
5.27 Dados guardados pelo *Trigger* . . . 110
5.28 Adulteração dos dados binários do arquivo armazenado . . . 110
5.29 Adulteramento dos dados binários capturado pelo *trigger* . . . 111
5.30 Histórico do registo de evidências após a adulteração da base de dados . . . 112

Lista de Tabelas

2.1 Passos envolvidos no *CRAB Protocol* Fonte: (Tasnim et al., 2018) 19

3.2 Requisitos funcionais 29
3.4 Requisitos não funcionais 30

5.2 Configurações dos ambientes na plataforma *Azure* 102
5.4 Configurações da base de dados no *MongoDB Cloud* 102
5.5 Comparação de Resultados 113

Listagens de Código

4.1 Ficheiro de configuração do Truffle . . . 54
4.2 Ficheiro migrations do Truffle . . . 54
4.3 Estrutura de uma evidência . . . 55
4.4 Estrutura de arquivo de uma evidência . . . 55
4.5 Estrutura de registos de utilizadores permitidos . . . 55
4.6 GET do ultimo registo de uma evidência . . . 56
4.7 Registar uma evidência (*EvidencesChain.sol*) . . . 56
4.8 Registar uma evidência (*EvidencesManager.sol*) . . . 57
4.9 Adicionar um arquivo digital (*EvidencesChain.sol*) . . . 58
4.10 Adicionar um arquivo digital (*EvidencesManager.sol*) . . . 58
4.11 Modificador "*onlyOwner*" . . . 59
4.12 Adicionar "allowed users" (*EvidencesChain.sol*) . . . 59
4.13 Adicionar "allowed users" (*EvidencesManager.sol*) . . . 60
4.14 Adicionar "allowed users" (*EvidencesManagerAllowedUsers.sol*) . . . 60
4.15 Transferir o "ownership" para outro utilizador (*EvidencesChain.sol*) . . . 61
4.16 Transferir o "ownership" para outro utilizador (*EvidencesManager.sol*) . . . 61
4.17 Visualizar a lista de registos de evidências permitidas (*EvidencesChain.sol*) . . . 61
4.18 Visualizar a lista de registos de evidências permitidas (*EvidencesManager.sol*) . . . 62
4.19 Verificação de permissão de visualização do utilizador . . . 62
4.20 Visualizar todo o histórico do registo de uma evidência (*EvidencesChain.sol*) . . . 64
4.21 Visualizar todo o histórico do registo de uma evidência (*EvidencesManager.sol*) . . . 64
4.22 Informações sobre o arquivo (*EvidencesChain.sol*) . . . 64
4.23 Informações sobre o arquivo (*EvidencesManager.sol*) . . . 64
4.24 Estrutura de uma entidade . . . 65
4.25 GET do ultimo registo de uma entidade . . . 65
4.26 Efetuar o registo do sistema (*EvidencesChain.sol*) . . . 65
4.27 Efetuar o registo do sistema (*EntitiesManager.sol*) . . . 66
4.28 Aprovar o registo do utilizador (*EvidencesChain.sol*) . . . 66
4.29 Aprovar o registo do utilizador (*EntitiesManager.sol*) . . . 66
4.30 Atualizar a informação do registo (*EvidencesChain.sol*) . . . 67
4.31 Atualizar a informação do registo (*EntitiesManager.sol*) . . . 67
4.32 Visualizar todos os utilizadores registados (*EvidencesChain.sol*) . . . 67
4.33 Visualizar todos os utilizadores registados (*EntitiesManager.sol*) . . . 68
4.34 Visualizar todo o histórico de atualizações (*EvidencesChain.sol*) . . . 68

4.35 Visualizar todo o histórico de atualizações (*EntitiesManager.sol*) 68
4.36 Efetuar login no sistema (*EvidencesChain.sol*) 68
4.37 Connection String MongoDB .env 69
4.38 Usar variavel de ambiente em app.module.ts 69
4.39 Conectar Nest com Mongo (*app.module.ts*) 70
4.40 Serviço Ethereum Configurações (*ethereum.service.ts*) 70
4.41 Serviço Ethereum Call Contract (*ethereum.service.ts*) 70
4.42 Serviço Autenticação (*auth.service.ts*) 71
4.43 Endpoint de login (*auth.controller.ts*) 72
4.44 Upload de arquivos (*upload.service.ts*) 73
4.45 Endpoint para upload de arquivos (*upload.controller.ts*) 73
4.46 Download de arquivos (*upload.service.ts*) 74
4.47 Endpoints download e verificação de arquivos (*upload.controller.ts*) 75
4.48 Monitoring Trigger 76
4.49 Drizzle Options 77
4.50 Drizzle Config 78
4.51 Obter a lista de evidências com Drizzle 78
4.52 Enviar uma transação com Drizzle 79
4.53 Upload de arquivos digitais 80
4.54 Ficheiro de configuração do Truffle 87
4.55 Contract Migrations 90
5.1 Excerto do ficheiro .info 98

1. Introdução

Cada vez mais se assiste a eventos onde agressões, abusos, violações de leis e princípios, maus-tratos e atos criminosos, são frequentes. De acordo com Eurojust (2022), nos 8 anos que estão incluídos no estudo (entre 2014 até 2021), houve um crescimento médio de 15% por ano de abertura de novos processos judiciais. A responsabilização de quem, como e porquê, ou a penalização de quem comete esses crimes, são tarefas de elevada complexidade, quer pela dificuldade em encontrar evidências, quer pela salvaguarda das mesmas. A manipulação das evidências são processos difíceis de controlar.

Embora exista hoje grande facilidade em conseguir registos de ocorrências, seja por vídeo, áudio, fotografia, texto, ou outros meios, não é simples que esses registos sejam provas aceites em processos judiciais consequentes (Ho, 2015).

É neste contexto que se pretende explorar tecnologias capazes de controlar os processos, quer na sua operacionalização, quer na sua monitorização. Conseguir garantir que um processo jurídico, em que registos de evidências não possam ser manipulados e que, se necessário se possam rastreabilizar e perceber todo o processo que antecedeu determinado estado de um processo, são essenciais.

Este trabalho de investigação pretende, por isso, explorar a aplicação das tecnologias associadas à *Blockchain*, como forma de suportar estes requisitos. A imutabilidade e rastreabilidade exigida e garantida nas transações financeiras, poderão ser igualmente aplicadas nestes processos? Poderão as evidências registadas segundo este processo, constituir provas futuras em processos judiciais? Por exemplo, numa situação de conflito militar, as evidências fidedignas registadas, poderão ser garantidamente mantidas sem serem manipuladas?

Estas são algumas das questões que sustentam a relevância desta investigação e às quais se espera poder ter algum tipo de resposta.

1.1 Motivação

A motivação é um fator crucial para a realização de qualquer projeto ou meta, e no âmbito académico isso não é diferente. Esta secção descreve as três principais motivações do autor.

Inicialmente, a motivação académica de concluir o mestrado tem relação com a busca por aprimorar conhecimentos teóricos e práticos na área da tecnologia, nomeadamente na área relativa a *blockchain* e a rastreabilidade que a mesma é capaz de fornecer, além de ser um importante diferencial no mercado de trabalho. O mestrado oferece oportunidades de aprendizado e desenvolvimento de habilidades, além de ser um desafio pessoal que contribui para a construção e melhoria da carreira profissional.

O interesse pessoal na área da rastreabilidade de documentos é uma segunda motivação, isto deve se à importância de garantir a autenticidade, integridade e confidencialidade de informações, especialmente em ambientes digitais. A rastreabilidade de documentos é fundamental para a gestão documental.

Por fim, a terceira motivação é a exploração de domínios reais da aplicabilidade da *blockchain*, na criação de um registo público e imutável de transações, oferecendo segurança e transparência. O uso da *blockchain* na rastreabilidade de documentos é uma área promissora, sendo capaz de garantir a autenticidade e integridade de informações, além de possibilitar a descentralização da gestão documental. Isto provém dos fatores reais que todos temos acompanhado nas notícias mundiais. A guerra, que de momento ocorre no país de origem do autor, aumentou a sua motivação para procurar formas de garantir autenticidade, integridade, confidencialidade e rastreabilidade das evidências recolhidas em cenários de atividades fortes, podendo ser uma fonte para posteriores acusações em crimes de guerra.

1.2 Objetivos

Este trabalho tem como objetivos:

- Perceber os requisitos para o rastreamento de uma evidência numa cadeia *blockchain*.
- Explorar o estado da arte relativo à *blockchain*.
- Perceber e estudar a aplicabilidade da *blockchain* no domínio jurídico.
- Desenvolver uma arquitetura que suporte a rastreabilidade e registo de evidências com *blockchain*.
- Demonstrar a aplicabilidade da proposta com um caso de estudo de registo de evidências num domínio a determinar.
- Analisar e divulgar os resultados obtidos e as dificuldades encontradas durante a implementação.

1.3 Estrutura do documento

Esta secção descreve a organização deste documento para que o leitor compreenda previamente o seu conteúdo.

No **Capítulo 1** é realizada a introdução ao tema, bem como elaborada a motivação, são apresentados os objetivos deste trabalho, a metodologia utilizada e o plano de trabalho. Segue-se o **Capítulo 2** onde é desenvolvido o enquadramento teórico, ou seja, uma revisão de literatura; também serão apresentadas algumas soluções existentes. No **Capítulo 3** é feita uma análise para a resolução do problema. Juntamente serão apresentadas as diferentes modelagens realizadas para serem implementadas. O **Capítulo 4** descreve o processo da implementação da solução, cabendo ao **Capítulo 5** a descrição da experimentação a fim de comparar duas abordagens: *Linux* e *Blockchain*. Por fim, **Capítulo 6** apresenta a conclusão do trabalho realizado, assim como a descrição resumida dos pontos cruciais de todo o documento.

1.4 Metodologia de Investigação

Nesta dissertação, é pretendido ajudar a mitigar os problemas relacionados com o armazenamento, rastreabilidade, imutabilidade e transparência em todo o ciclo de vida de uma evidência propondo uma solução tecnológica baseada em *blockchain*. Para tal, e de forma a organizar a realização deste trabalho, foi realizada uma pesquisa sobre a metodologia de investigação que melhor se adaptaria a este problema, neste caso o *DSR (Design Science Research)*.

1.4.1 *Design Science Research*

Design Science Research (DSR) é uma abordagem de investigação que visa criar soluções e construir uma nova realidade para resolver problemas. O processo de *DSR* inclui tipicamente seis etapas: identificação do problema, definição de objetivos para uma solução, conceção e desenvolvimento de artefactos, demonstração da solução, avaliação da solução, e comunicação dos resultados (Pello, 2018).

Um aspeto chave do *DSR* é a ênfase na criação de artefactos como meio de fazer avançar o conhecimento. Os artefactos podem ser produtos físicos, sistemas de software, processos empresariais, ou qualquer outra construção tangível e intangível que seja concebida para satisfazer um objetivo específico. Outro aspeto importante do *DSR* é a sua natureza interdisciplinar. O *DSR* recorre a várias disciplinas, incluindo engenharia, informática, psicologia, sociologia e negócios, entre outras, para abordar problemas complexos de design. Esta abordagem interdisciplinar permite aos investigadores trazer diversas perspetivas e conhecimentos especializados sobre o problema do design, que resultam em soluções mais abrangentes e eficazes (Brocke et al., 2020).

De acordo com Pello (2018) e Brocke et al. (2020) as seis etapas deste projeto são:

1. **Identificação de problemas e motivação:** Define o problema da pesquisa específica e justifica o valor da solução. Justificar o valor da solução realiza duas coisas: motiva o investigador e o público da pesquisa a perseguir a solução e ajuda o público a apreciar a compreensão sobre o problema. Neste projeto pretende-se elaborar uma análise sobre a problemática existente na rastreabilidade e armazenamento de uma evidência.

2. **Definição dos objetivos:** Os objetivos de uma solução podem ser inferidos a partir da definição do problema e do conhecimento do que é possível e viável. Após a análise de documentos científicos, no âmbito deste projeto, pretende-se identificar os principais requisitos para implementar um sistema de armazenamento e rastreabilidade de evidências.

3. **Desenho e desenvolvimento:** Desenhar a arquitetura tecnologia e modelar aquilo que representará uma evidência, com a utilização da *blockchain*, de uma possível solução para permitir concretizar os objetivos definidos, nomeadamente, a rastreabilidade em uma evidência.

4. **Demonstração:** A demonstração será com base na realização de um caso de estudo, com a aplicação da solução proposta no rastreio de evidências coletadas em um ambiente de guerra e a simulação da sua utilização em um processo jurídico.

5. **Avaliação:** Definir formas de avaliar os resultados, como também validar se os objetivos foram concluídos perante os resultados obtidos na utilização da solução sobre o caso de estudo.

6. **Comunicação:** Criar impacto social para a problemática existente com o caso de estudo, apresentar em formato escrito sobre todo o processo de investigação e desenvolvimento, nomeadamente a dissertação, publicação dos resultados obtidos e da solução desenvolvida com indicação para desenvolvimentos futuros de modo a permitir futuros investigadores aperfeiçoar, a sua maneira, esta solução.

1.5 Plano de Trabalho

O presente capítulo tem como objetivo apresentar a organização, de forma temporal, das tarefas a serem realizadas ao longo dos seis meses previstos, após a aprovação desta proposta, pressupondo que a aceitação ocorra por volta do início do mês de abril.

A Figura 1.1 representa o diagrama de *Gantt* que mostra o plano de trabalho das principais tarefas a serem elaboradas.

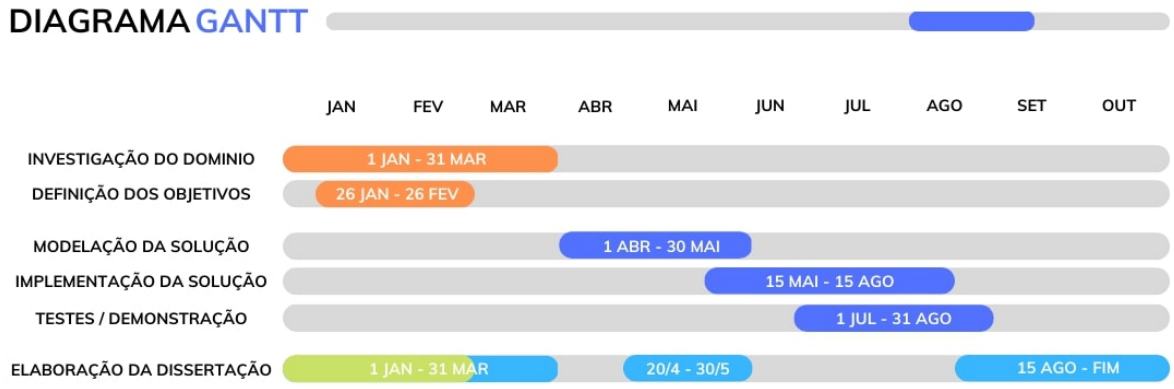


Figura 1.1: Plano de trabalho

Na Figura 1.1 é possível observar quatro cores diferentes nas tarefas. A cor verde na tarefa de elaboração da dissertação, representa a elaboração da proposta que serviu como ponto de partida para a elaboração do documento da dissertação. Enquanto se aguardava a aprovação da proposta foi realizada a preparação do documento oficial da dissertação com a passagem das informações relevantes deste documento, ou seja, os capítulos um e dois.

A cor azul-claro simboliza o processo de elaboração do documento final da dissertação. Essa tarefa não tem um ponto de início e fim definidos, mas sim vários. Essa distribuição permite uma melhor forma de abordar a elaboração do documento, permitindo atualizar periodicamente os capítulos à medida que se modele e desenvolva uma solução.

A cor de laranja, estão as tarefas "Investigação do domínio" e "Definição de objetivos" que envolveram a exploração, análise e levantamento de soluções existentes, bem como a definição de requisitos e objetivos. Parte dessas tarefas estavam incluídas na proposta de dissertação.

Por último, mas igualmente importante, a azul-escuro, estão representadas as tarefas da elaboração da solução, ou seja, as tarefas "Modelação da solução", "Implementação da solução" e "Testes/Demonstração", onde foram realizados todos os modelos necessários, foram abordadas as questões de implementação e realizados os testes da solução, fruto deste trabalho, por fim a demonstração, utilizando um caso de estudo a definir.

2. Revisão da Literatura

A revisão de literatura é um elemento fundamental neste trabalho, proporcionando uma análise abrangente das principais contribuições académicas e pesquisas relevantes relacionadas ao domínio em foco. Foram utilizadas diversas fontes, como *Google Scholar, Scientific Electronic Library Online, b-on, RCAAP - Repositórios Científicos de Acesso Aberto de Portugal*, motor de pesquisa *Google*, entre outros. Esta secção apresenta o o que de mais relevante se encontrou na leitura efetuada.

2.1 Registo de evidências

Segundo Watson & Jones (2013) *"(...) Secure evidence storage is dedicated storage space for the sole purpose of securely storing evidence relating to any forensic cases (...). The secure evidence storage facility is the physical embodiment of the chain of custody, supported by robust procedures for management of evidence. Evidential storage must be the most secure area of storage (...) and the most rigorously controlled area (...). All access to it must be regularly reviewed and restricted to the minimum possible number of employees. (...)"*. Apesar de Watson & Jones (2013) se referir a um laboratório físico, o conceito de armazenamento de evidências mantém-se ao longo dos anos e ao longo das plataformas que abordam este objetivo.

Continuando, Watson & Jones (2013) diz, *"(...) the secure evidence storage facility must be constructed so that it can defeat any forced or otherwise unauthorized entry as well as being resistant to any environmental threats. (...)"*. Segundo o autor, o repositório ou o local de armazenamento de evidências necessita ser um local seguro, protegido, resistente a entradas indevidas e restrito ao mínimo de pessoas possíveis.

Mas então, afinal o que é uma evidência para ser tão importante assegurar a sua segurança?

2.1.1 O que é uma evidência?

Segundo Heinrich & Jerry (2022), evidência, na lei, é qualquer dos elementos materiais ou afirmações de facto que possam ser submetidas a um tribunal como meio de apurar a verdade de qualquer alegado assunto de facto que esteja sob investigação. Citando Heinrich & Jerry (2022), *"(...) judicial evidence has always been a human rather than a technical problem. (...) Since the means of acquiring evidence are clearly variable and delimited, they can result only in a degree of probability and not in an absolute truth in the philosophical sense (...)"*. Devido à forma variável de obtenção de evidências, que será mencionado no ponto 2.1.3, conseguir garantir a verdade absoluta de uma evidência torna se praticamente impossível, isto porque, desde a recolha de uma evidência, até a sua apresentação no tribunal, existe uma enorme possibilidade de "poluição" de evidências. Daí provém a enorme importância de assegurar o armazenamento de evidências.

2.1.2 Evidência num processo jurídico

Segundo mencionado no ponto 2.1.1 evidência pode ser qualquer elemento material ou afirmações de facto, mas, não é assim tão linear. Num processo jurídico, para uma evidência ser considerada válida e ser aceite no tribunal existem três condições de aceitabilidade (Ho, 2015):

- **Relevância:** A relevância é um conceito chave na investigação dos factos jurídicos, desempenha um papel significativo na lei da evidência e determina quais as provas que podem ser recebidas pelo tribunal em processos judiciais. É um conceito relacional, sendo um facto considerado relevante apenas em relação a outro facto, frequentemente descrita como a tendência da prova para tornar um facto mais ou menos provável. O conceito de relevância é por vezes entendido como significado matemático de probabilidade, contudo, alguns defendem uma definição mais ampla de relevância como prova que fornece uma razão para acreditar que uma proposta de facto material para o caso é verdadeira ou falsa.

- **Materialidade e Factos em questão:** A materialidade dos factos num processo jurídico é determinada pela lei aplicável a esse caso, como a lei que define o crime numa ação penal ou a lei que estabelece os elementos de uma reivindicação jurídica num julgamento civil. Um facto pode ser relevante para outro facto, mas sua admissibilidade como prova depende da sua materialidade e da questão da sua disputa. Não é permitido apresentar provas para provar factos imateriais e a exigência de que um facto deve ser objeto de disputa para ser admissível como prova, pode variar de acordo com a jurisdição.

- **Admissibilidade:** O conceito de admissibilidade e relevância em processos judiciais são frequentemente utilizados de forma intercambiável, mas são conceitos distintos. A relevância refere-se à ligação lógica entre a evidência e uma proposta factual que é material e está em causa num caso. A evidência que é irrelevante ou imaterial para o caso é frequentemente referida como "inadmissível". Contudo, a prova também pode ser excluída por outras razões, *"(...) Two prominent examples of such rules of admissibility or rules of exclusion are the rule against hearsay evidence and the rule against character evidence. (...)"*. Isto significa que o tribunal pode excluir provas, mesmo que sejam relevantes para o caso.

2.1.3 Tipos de evidências

Existem diferentes formas de evidência que podem ser apresentadas perante um processo jurídico, tais como, testemunho, boatos, confissões, prova documental, documentos comerciais, evidências reais, entre outros (Browne, 2005).

Nesta dissertação serão abordadas as evidências reais que podem ser utilizadas num processo jurídico, pois é o foco de recolha neste estudo, as evidências reais podem ser (Information, 2022; INTERPOL, 2019):

- **Objetos materiais:** Por exemplo, num caso de homicídio é a arma do crime. Mas devido à possibilidade do objeto ser destruído, e no caso deste estudo, algo físico será impossível de armazenar, então fotografias do objeto, por exemplo, podem ser apresentadas em processo judicial ao invés do objeto em si.

- **Fotografias:** As fotografias podem ser introduzidas como prova do que apresentam. Mas a autenticidade de uma fotografia deve ser provada antes de ser admitida como prova.

- **Gravações de vídeo:** Da mesma forma como as fotografias, as gravações de vídeo também têm que ser autenticas e genuínas para serem consideradas como prova.

- **Evidências digitais:** Engloba dados armazenados em dispositivos digitais, como computadores e *smartphones*, que podem ser pertinentes em investigações ou casos jurídicos. Isso inclui uma variedade de dispositivos, como servidores, sistemas de vídeo vigilância, *drones* e consolas de jogos, que podem conter informações importantes para a resolução de questões legais.

2.1.4 Riscos no registo e armazenamento de uma evidência num processo jurídico

Após analisar o que é uma evidência, conseguimos compreender os riscos associados ao seu registo e armazenamento. As evidências, por serem um artefacto, com valor sobre um julgamento, após passarem pelas três condições de aceitação (secção 2.1.2), também são alvo de "ataques" como, por exemplo, i)

perda ou destruição da evidência; ii) manipulação da evidência; iii) falsificação da evidência; iv) forja de evidências; pelo que serão prejudicados na existência de provas de justiça ou prejudicar negativamente alguém inocente num processo jurídico. Daí provem a forte necessidade de existir um processo seguro de registo e armazenamento de evidências, e uma possibilidade de rastreabilidade para garantir identificar os envolvidos em procedimentos sobre os registos. A esse processo dá se o nome de *Chain of Custody* (secção 2.1.6) (Ho, 2015; Information, 2022).

2.1.5 Recolha de uma evidência

Para demonstrar o processo de recolha de evidências, foi utilizado o artigo *"INTERPOL Global guidelines for digital forensics laboratories"* (Diretrizes Globais da INTERPOL para Laboratórios de Ciências Forenses Digitais) (INTERPOL, 2019). Este artigo inclui o processo comum para efetuar a aquisição de dados (Figura 2.1), e um fluxograma que detalha o processo de recolha (Figura 2.2).



Figura 2.1: *Data Acquisition Process on Computer* - INTERPOL (2019)

Passos para a recolha/aquisição de dados segundo a INTERPOL (2019):

- **Identificar o suporte de armazenamento**: O examinador deve preparar um suporte de armazenamento compatível com tamanho de dados suficiente, antes da entrega. Se a prova for grande, o examinador poderá ter de preparar vários suportes de armazenamento para armazenar o ficheiro de imagem.
- **Imagem da exposição**: Para criar uma imagem de uma exposição para fins forenses, é crucial ligá-la inicialmente a um bloqueador de escrita para evitar quaisquer alterações e manter a integridade dos dados. A maioria das ferramentas forenses inclui esta funcionalidade. O ficheiro com a imagem deve então ser guardado num suporte de armazenamento preparado. Para garantir a segurança dos suportes de armazenamento e do ficheiro de imagem, utilize um formato de rotulagem normalizado, colocando todas as provas adquiridas em *hash* com *SHA256*.
- **Verificar a prova e o ficheiro de imagem**: Depois do ficheiro de imagem ter sido criado, o examinador tem de verificar se este pode ser executado com o uso do software forense e que os valores de *hash* tanto da prova como do ficheiro de imagem correspondem.
- **Documentar todas as acções**: O último passo no exame e análise do computador é documentar o processo, as ferramentas utilizadas, os valores de *hash*, a data e a hora, bem como as iniciais do examinador nas notas de caso.

No fluxograma apresentado na Figura 2.2, é possível compreender os passos descritos de forma mais detalhada. Vale ressaltar o passo descritivo da criação da cópia da imagem. Caso a ferramenta tenha sido verificada e validada pelo tribunal, apenas uma cópia é criada, caso o contrário então serão feitas pelo menos duas cópias. Após a conclusão da criação das cópias, ocorre o processo de validação dos *clones* em relação às *hashes* calculadas para garantir que o procedimento tenha sido executado corretamente. Somente após essa validação é que todo o conteúdo físico é cuidadosamente guardado e um relatório é elaborado para o transporte até o cofre de evidências, a fim de garantir que a *chain of custody* não seja quebrada.

2.1.6 *Chain of Custody*

A *chain of custody* (cadeia de custódia) é um processo crucial no campo da investigação criminal. Refere-se ao conjunto de procedimentos documentados que garantem a integridade, autenticidade e rastreabilidade das evidências desde o momento em que são recolhidas até a sua apresentação em tribunal.

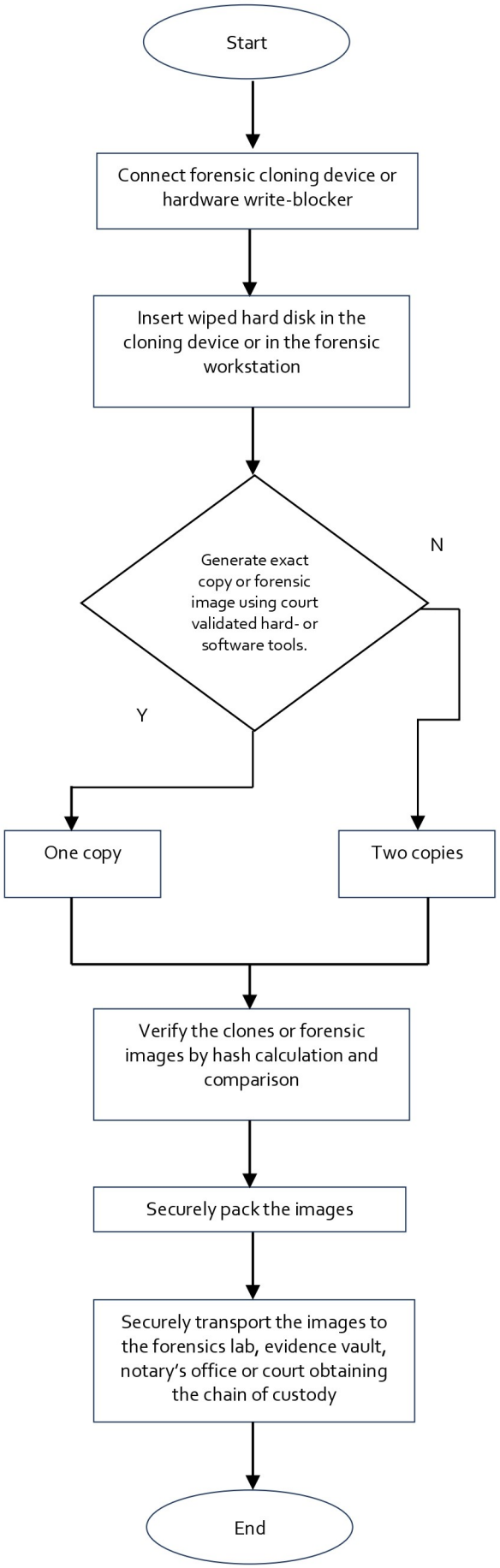


Figura 2.2: *Acquisition Process Flow Chart* - INTERPOL (2019)

A *chain of custody* é o registo completo e detalhado de todas as pessoas que tiveram acesso a uma evidência num determinado momento, desde sua recolha inicial até sua apresentação em tribunal. A importância da cadeia de custódia reside no fato de que ela garante a integridade e a admissibilidade das evidências, pois documenta o seu manuseio de forma imparcial e transparente, evitando adulterações, contaminações ou perdas (Providers, 2019; Tommaso et al., 2023).

Etapas da *chain of custody*

1. **Recolha inicial:** A recolha inicial é o primeiro passo da cadeia de custódia. Nessa etapa, a evidência é identificada, isolada e preservada de forma a garantir sua integridade. É essencial documentar a data, hora, local e responsável pela recolha, bem como qualquer informação relevante sobre o estado da evidência (P. & P., 2022).
2. **Transporte e armazenamento:** Durante o transporte da evidência, devem ser implementadas medidas de segurança para evitar danos, contaminação ou perda. O uso de embalagens adequadas e o registo detalhado de cada transferência são essenciais para manter a cadeia de custódia intacta. O armazenamento seguro das evidências também é fundamental, garantindo que apenas pessoal autorizado tenha acesso a elas (T. & C., 2023).
3. **Documentação e registo:** Durante todo o processo, é fundamental documentar todas as atividades relacionadas à evidência. Cada manipulação, transferência, análise e devolução devem ser registadas de forma precisa e detalhada. Isso inclui informações sobre as pessoas envolvidas, datas, horários, condições ambientais e qualquer outra informação relevante (Y. & R., 2022).
4. **Análise forense:** Durante a análise forense, é crucial manter a cadeia de custódia intacta. Os examinadores devem garantir que qualquer alteração ou manipulação seja registada devidamente, incluindo a técnica utilizada, os resultados obtidos e as conclusões alcançadas. Isso permite uma revisão transparente e objetiva do trabalho realizado (A. & J., 2023).

A cadeia de custódia desempenha um papel fundamental no registo de evidências, garantindo sua integridade, autenticidade e rastreabilidade. Agora que compreendemos o que é necessário para assegurar a admissibilidade de evidências em tribunal, podemos explorar a tecnologia da *blockchain* e validar que parte do que a *blockchain* oferece, se enquadra perfeitamente no processo mais importante no mundo de evidências, *chain of custody*.

2.2 Tecnologia *Blockchain*

Inicialmente, *blockchain* surgiu num projeto de Stuart Haber e W. Scott Stornetta em 1991, cientistas da computação que introduziram uma solução computacionalmente prática para documentos digitais com o seguinte conceito: *"(...) uma vez que eles eram gerados com registro de data que não podiam ser adulterados. (...)"* (Binance, 2018). Apesar de ter sofrido ainda algumas melhorias (em 1992) este conceito não foi amplamente utilizado até 2008, ano em que surgiu a *Bitcoin* de Satoshi Nakamoto.

Segundo IBM (2023), *"(...) Blockchain is a shared, immutable ledger that facilitates the process of recording transactions and tracking assets in a business network. (...)". Blockchain* é uma base de dados descentralizada e distribuída. É descentralizada, pois a informação é armazenada em vários *nodes* da rede ao invés ser armazenada num único lugar (ex:.base de dados centralizada); é distribuída porque cada *node* da rede tem uma cópia completa da cadeia de blocos, permitindo assim que todas e quaisquer transações sejam validadas de maneira descentralizada garantindo uma maior segurança e imutabilidade dos dados.

"(...) A blockchain - originally block chain - is a continuously growing list of records, called blocks, which are linked and secured using cryptography (...)"(Li et al., 2019). Os blocos são encadeados criptograficamente utilizando tecnologia de apontadores *hash*, ou seja, cada bloco é assinado digitalmente e encadeado ao bloco anterior, incluindo também o valor de *hash* desse bloco (Liu et al., 2017) (Figura 2.3). A *blockchain* foi projetada de tal forma, que se torna genuinamente resistente à modificação de dados (Li et al., 2019) isto devido ao facto de que novos blocos apenas podem ser anexados ao final da cadeia, os existentes não podem ser excluídos nem atualizados (Liu et al., 2017).

De uma forma geral, a teoria da *blockchain* pode ser definida através dos três seguintes conceitos (Tsai, 2021).

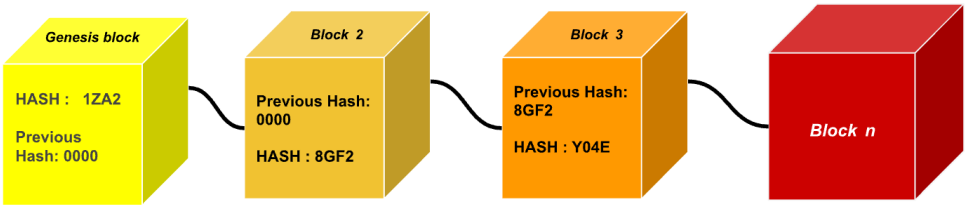


Figura 2.3: Blockchain Ilustration - Chain (2023)

- **Arquitetura ponto-a-ponto**

  "(...) A peer-to-peer (P2P) service is a decentralized platform whereby two individuals interact directly with each other, without intermediation by a third party. (...)"(Hayes, 2021)

- **Armazenamento de dados**

  O armazenamento de dados é efetuado através de um carimbo temporal e uma verificação de transação. Os registos das transações são armazenados em blocos. Cada bloco contém os valores de *hash*, data e hora da transação, mensagem de transação, cada bloco contém um *nonce* diferente para calcular um novo valor de *hash*. O novo bloco gerado é adicionado à cadeia anterior através da verificação da transação, de modo a ligar uma longa cadeia de blocos de dados.

- **Mecanismo de consenso**

  O mecanismo de consenso na rede P2P utiliza criptografia de chave privada e assinatura digital para garantir a segurança das partes envolvidas na negociação.

Devido a estes conceitos, a *blockchain* consegue garantir a imutabilidade e a segurança dos dados, fornecendo também a possibilidade de rastreabilidade das transações efetuadas dentro de um determinado bloco em formato temporal.

2.2.1 Evolução e histórico da *Blockchain*

O conceito de *Blockchain* foi introduzido pela primeira vez em 2008 com a publicação do *white paper* do *Bitcoin* por *Satoshi Nakamoto*, que descrevia um sistema de criptomoeda descentralizada baseada numa cadeia de blocos de transações verificadas por consenso (Nakamoto, 2008). Desde então, o *Blockchain* tem evoluído significativamente e tem sido aplicado em várias áreas além das criptomoedas.

Ao longo dos anos, várias pesquisas e inovações têm sido realizadas para aprimorar o conceito de *Blockchain* e adaptá-lo para diferentes casos de uso. Por exemplo, a introdução de diferentes tipos de *Blockchain*, como público, privado e consórcio (secção 2.2.2), tem permitido uma maior flexibilidade na sua aplicação em diferentes contextos (Bano et al., 2017). A evolução dos algoritmos de consenso também tem sido um tema de pesquisa ativa, com a introdução de alternativas ao consenso de prova de trabalho utilizado pelo *Bitcoin*, como o consenso de prova de participação *Proof of Stake*, que promete maior eficiência energética e escalabilidade (Swan, 2015).

Além disso, a *Blockchain* tem sido objeto de interesse de várias organizações e consórcios que buscam explorar o seu potencial em diferentes setores. Por exemplo, o *Ethereum*, outra criptomoeda baseada em *Blockchain*, que tem sido uma plataforma popular para o desenvolvimento de contratos inteligentes e aplicações descentralizadas (Ethereum, 2023a).

2.2.2 Modelos de *Blockchain*

Segundo Binance (2020), existem três principais modelos de *blockchain*:

- ***Blockchain* pública:** É aberta a qualquer pessoa e permite a participação de qualquer indivíduo ou entidade na rede. É conhecida por sua descentralização e transparência, onde as transações são registadas publicamente e verificadas pela comunidade. As *blockchains* públicas são amplamente

utilizadas em criptomoedas, como o *Bitcoin* e o *Ethereum*. Embora sejam altamente transparentes, a privacidade pode ser menor uma vez que todas as transações são visíveis para todos os participantes e a velocidade de transação pode ser mais lenta devido ao processo de consenso descentralizado.

- ***Blockchain* privada:** É controlada por uma única entidade ou grupo de entidades e o acesso à rede é restrito. É geralmente usada em ambientes corporativos, onde as entidades podem ter controlo total sobre as transações e a privacidade dos dados. A eficiência é alta, pois o número de participantes é limitado, e a velocidade de transação pode ser rápida. No entanto, a natureza centralizada pode resultar em menor transparência e dependência de uma única entidade ou grupo.
- ***Blockchain* de consórcio:** É controlada por um grupo de entidades com interesses comuns, como empresas ou organizações governamentais. A descentralização é maior do que nas *blockchains* privadas, pois o controlo é compartilhado entre várias entidades. A transparência e a privacidade podem variar, dependendo das regras e regulamentos definidos pelo consórcio. As *blockchains* de consórcio são frequentemente usadas em setores específicos, como finanças, saúde ou logística, onde as entidades podem colaborar e compartilhar dados de forma mais eficiente.

2.2.3 Mecanismos de Consenso

Na *blockchain* a *hash* é gerada a partir dos mecanismos de consenso. São os mecanismos de consenso que influenciam a forma como as transações são verificadas, a energia a ser utilizada, o uso da rede, a velocidade de transação, que, por consequência, influenciam nas vantagens e desvantagens de utilização da *blockchain* (Rosenberg, 2022).

Existem múltiplos tipos de mecanismos de consenso, mas neste trabalho serão apresentados apenas os quatro mais relevantes, *Proof of Work (PoW)*, *Proof of Stake (PoS)*, *Delegated Proof of Stake (DPoS)* e *Practical Byzantine Fault Tolerance (PBFT)* (Lamounier, 2018; Rosenberg, 2022).

- ***Proof of Work (PoW)***

  O mecanismo de consenso *PoW* foi introduzido pela *Bitcoin* e é amplamente utilizado em criptomoedas. Neste método, os participantes/*miners* da rede competem para resolver problemas computacionais complexos. O primeiro a encontrar a solução correta tem o direito de adicionar um novo bloco à *blockchain* e é recompensado. Embora o *PoW* tenha demonstrado robustez e segurança, apresenta desvantagens, como alto consumo de energia e escalabilidade limitada (Nakamoto, 2008).

- ***Proof of Stake (PoS)***

  O mecanismo de consenso *PoS* atribui o direito de adicionar um novo bloco à blockchain com base na participação de criptomoedas que um utilizador possui e "aposta" na rede. Em vez de competir por recursos computacionais, os *validators* são escolhidos aleatoriamente com base na quantidade de moedas mantidas. O *PoS* é mais eficiente em termos de consumo de energia e escalabilidade em comparação com o *PoW*, mas também apresenta desafios, como a possibilidade de ataques de *Nothing at Stake* e centralização (Lamounier, 2018).

- ***Delegated Proof of Stake (DPoS)***

  No mecanismo de consenso *DPoS*, os detentores de *tokens* votam em delegados para validar transações e produzir blocos. Esses delegados, ou "*nodes* eleitos", têm a responsabilidade de manter a rede e assegurar o consenso. O *DPoS* oferece alta velocidade de transação e escalabilidade, mas pode levar à centralização, já que apenas um número limitado de delegados é selecionado (Lamounier, 2018).

- ***Practical Byzantine Fault Tolerance (PBFT)***

  O mecanismo de consenso *PBFT* permite que um grupo de *nodes* chegue a um consenso, mesmo que alguns deles sejam maliciosos ou apresentem falhas. Ele utiliza um processo de votação e acordos em etapas para garantir a integridade das transações. O *PBFT* é conhecido por sua alta tolerância a falha(s) bizantina(s) e rápida finalidade de transação. No entanto, sua escalabilidade pode ser limitada à medida que o número de *nodes* aumenta (Lamounier, 2018).

Na Figura 2.4 é apresentado o modelo de funcionamento do mecanismo de consenso *Proof of Work* (PoW). O *Proof of Work* funciona através da resolução de "*puzzles*" ou equações matemáticas que são

enviadas pela *Network* para os *miners*. Estes, por sua vez, competem pela resolução dos mesmos fazendo diversas submissões, a *Network* por sua vez valida as resoluções e no caso da existência de uma resolução válida envia uma *Award* ao *miner* que a resolveu.

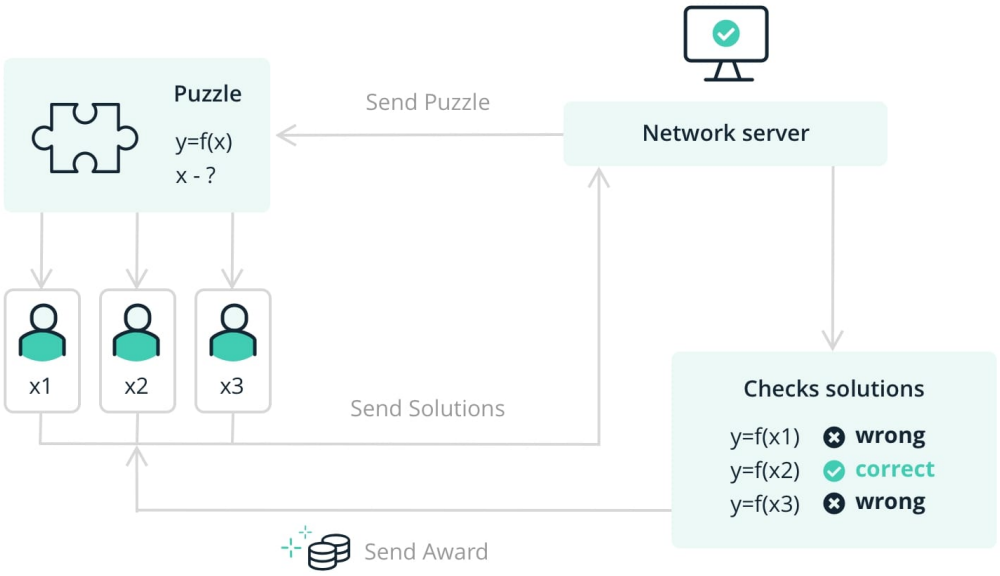


Figura 2.4: Funcionamento do *Proof of Work* (PoW)
Fonte: (Moreland, 2022)

2.2.4 Vantagens da *Blockchain*

Da mesma forma como uma base de dados comum a *blockchain* armazena dados/informações em formato digital. Mas não podemos chamar a *blockchain* como uma base de dados comum, pois esta fornece enumeras vantagens em comparação com uma base de dados tradicional. De uma forma breve, as vantagens de uma *blockchain* são (Budhi, 2022; Pratt, 2021):

- **Imutabilidade**

  A *blockchain* suporta a imutabilidade. Uma vez que as transações ficam registadas não podem ser apagadas ou alteradas. Todas as transações são marcadas/carimbadas com hora e data ficando na cadeia permanentemente. Desta forma, a *blockchain* pode ser utilizada para acompanhar a informação ao longo do tempo, garantindo uma auditoria segura e fiável da informação, sendo extremamente útil no estudo.

- **Transparência & Descentralização**

  "(...) Blockchain really proves its value when there's no central actor who enables trust, explained Daniel Field, head of blockchain at UST, a global provider of digital technology and services. (...)" (Budhi, 2022). Devido a esta descentralização, qualquer entidade da rede pode verificar os dados dos registos na *blockchain*, como consequência existe uma enorme confiança na rede.

- **Privacidade**

  A privacidade é uma das chaves principais das possibilidades da *blockchain*. Devido à sua descentralização e imutabilidade, os dados armazenados não são associados a um nome ou entidade tornando os dados anónimos, resolvendo assim o problema da privacidade que existe em bases de dados tradicionais.

- **Rastreabilidade**

  Rastreabilidade refere-se à capacidade de rastrear o histórico, utilização ou localização de um produto, ou material ao longo de todo o seu ciclo de vida. Implica a criação de um registo claro e

abrangente de todos os aspetos de um produto, incluindo a sua origem, processamento, transporte, armazenamento e utilização final. A rastreabilidade ajuda a garantir a qualidade, segurança e responsabilidade dos produtos, é essencial em indústrias como a alimentar e farmacêutica, onde a segurança e integridade dos produtos é fundamental. Em resumo, a rastreabilidade é a capacidade de rastrear um produto do início ao fim, fornecendo um histórico completo do seu percurso. *"(...) Blockchain creates an irreversible audit trail, allowing easy tracing of changes on the network. (...)"* (Olsena & Borit, 2013). Como a imutabilidade é um facto, a forma como os dados são armazenados também (recorrendo às validações dos dados e a carimbagem dos mesmo com a utilização da data e hora) a rastreabilidade torna se uma vantagem-facto possível nas *blockchain*.

- **Custos Reduzidos**

  Custos reduzidos resume-se após a implementação da *blockchain* em cenários de organizações, pois cria eficiências no processamento de transações, reduz a necessidade de tarefas manuais tais como agregar dados, bem como também facilita nos processos de relatórios e auditorias.*"(...) Experts pointed to the savings that financial institutions see when using blockchain, explaining that blockchain's ability to streamline clearing and settlement translate directly into process cost savings. (...)"* (Pratt, 2021).

- **Rapidez na operação**

  Ao eliminar os intermediários e como a operação é elaborada de *P2P* existe uma vantagem significativa em questões do tempo do que num método convencional.

2.2.5 Desvantagens da *Blockchain*

Como qualquer tecnologia que contém vantagens, também existem desvantagens na *blockchain* (Iredale, 2020):

- **Escalabilidade**

  As *blockchains* não são escaláveis e as transações são concluídas em função do congestionamento da rede, que por sua vez, em caso de haver muitas pessoas ou *nodes* a serem adicionados a rede, existe a maior hipótese de abrandamento do funcionamento.

- **Consumo de energia**

  O Consumo de energia é uma desvantagem, mas tudo depende do algoritmo de consenso (secção 2.2) utilizado pela *blockchain*.

- **Imutabilidade dos dados**

  Apesar de ser uma enorme vantagem, certas entidades podem ver esta "vantagem" como uma potencial desvantagem no que toca a uma possível necessidade de alteração de um certo registo, pois será impossível a modificação dos dados assim que foram adicionados na rede.

- **Ineficiência e lentidão**

  Tal como mencionado, devido ao problema de escalabilidade, a *blockchain* poderá ficar bastante lenta em casos de enormes quantidades de dados a serem adicionados em simultâneo.

2.2.6 Riscos da *Blockchain*

Numa investigação para utilização da *blockchain* numa implementação, terá que se analisar também os potenciais riscos associados à mesma.

Existem várias categorias de riscos que se associam às *blockchains*, das quais seis mais comuns são (Li et al., 2017):

- **Vulnerabilidade dos 51%**

  A *blockchain* depende de um mecanismo de consenso (abordado na secção 2.2.3) estabelecer a confiança entre os participantes da rede. No entanto, o mecanismo de consenso tem uma vulnerabilidade

chamada "ataque de 51%". Em cadeias de bloqueio baseadas em *PoW*, se um único *miner* controlar mais de 50% do poder total de *hashing*, pode lançar um ataque de 51%. Isto pode acontecer se a potência mineira se concentrar em algumas *pools*. Em cadeias de bloqueio baseadas em *PoS*, pode ocorrer um ataque de 51% se um único *miner* controlar mais de 50% das *cripto-coins*. Um atacante pode usar esta vulnerabilidade para conduzir vários ataques, tais como inverter as transações, iniciar gastos duplos, modificar a ordem de transação, dificultar as operações de outros *miners* e impedir a confirmação de transações normais.

- **Segurança de chave privada**

  A tecnologia *blockchain* baseia-se em chaves privadas como forma de identidade e segurança, com os utilizadores a gerar e manter estas chaves em vez de serem mantidas por terceiros. Uma vulnerabilidade no algoritmo *Elliptic Curve Digital Signature Algorithm* (Algoritmo de Assinatura Digital de Curvas Elípticas) (ECDSA) utilizado na *blockchain* pode permitir aos atacantes recuperar chaves privadas, o que pode levar à perda ou roubo de chaves, resultando no risco de adulteração ou perda de controlo sobre as contas. Além disso, sem instituições centralizadas para localizar e recuperar chaves roubadas, torna-se difícil abordar estas questões.

- **Atividade criminosa**

  A *Bitcoin*, devido ao seu anonimato, tem sido utilizada em atividades ilegais, tais como resgates, em que os atacantes a utilizam como forma de pagamento por extorsão, e *underground market*, onde é utilizada como moeda de compra e venda de bens ilegais. Além disso, também tem sido utilizado para lavagem de dinheiro devido às suas características de pagamento virtual em rede e ao baixo risco de deteção. Algumas plataformas e aplicações tais como *"Silk Road"* e *"Dark Wallet"* facilitam ainda mais estas atividades ilegais, fornecendo anonimato e encriptação nas transações.

- **Duplo gasto**

  O mecanismo de consenso da *blockchain* pode validar transações, mas ainda não consegue evitar a duplicação de gastos, o que ocorre quando um consumidor utiliza a mesma moeda criptográfica várias vezes para transações. Uma forma de isto acontecer é através de um ataque de competição (*race attack*), onde um atacante explora o tempo entre o início e a confirmação de duas transações para lançar rapidamente um ataque.

- ***Criminal Smart Contracts***

  Os criminosos podem aproveitar contratos inteligentes (secção 2.2.7) para uma variedade de atividades maliciosas, que podem constituir uma ameaça à nossa vida quotidiana. Os *CSC (Criminal Smart Contracts)* podem facilitar a fuga de informação confidencial, roubo de chaves criptográficas e vários crimes do mundo real (por exemplo, homicídio, fogo posto, terrorismo, etc.)

- **Vulnerabilidades em *smart contracts***

  Os *smart contracts*, como programas que correm na *blockchain*, podem ter vulnerabilidades causadas por defeitos dos mesmos. Algumas das potenciais vulnerabilidades identificadas são: dependência de transações; dependência dos carimbos temporais; exceções mal resolvidas.

2.2.7 Smart Contracts

De acordo com *Nick Szabo*, criador do conceito por trás dos *smart contracts "(...) the most primitive type of smart contract is the vending machine in which transactions are based on simple automation.(...) with its automated mechanisms, accepts the coins, returns the change and finally hands over the sold item. The vending machine therefore completes the transaction on its own when the necessary prerequisites are met, that is, a sufficient amount of money has been handed over to the machine.(...) Additionally, since the items for sale are situated within the vending machine, it is capable of protecting the contract from unauthorised changes. (...)"* de Lauslahti et al. (2017).

Por outro lado, temos uma definição de Lauslahti et al. (2017) mais ajustada ao tema *blockchain "(...) smart contracts are defined as digital programs, based on the blockchain consensus architecture, which will self-execute when the terms of the agreement are met, and due to their decentralised structure are also self-enforcing and tamper-proof. (...)"*. Também o autor indica que a definição principal de *smart*

contracts ainda não se encontra totalmente compreendida *"(...) A fully established definition for smart contracts has yet to be formed, and the official legal status of smart contracts is not perfectly clear. (...)"*

Na Figura 2.5 são apresentadas as principais vantagens dos *smart contracts* de acordo com Aggarwal & Kumar (2021):

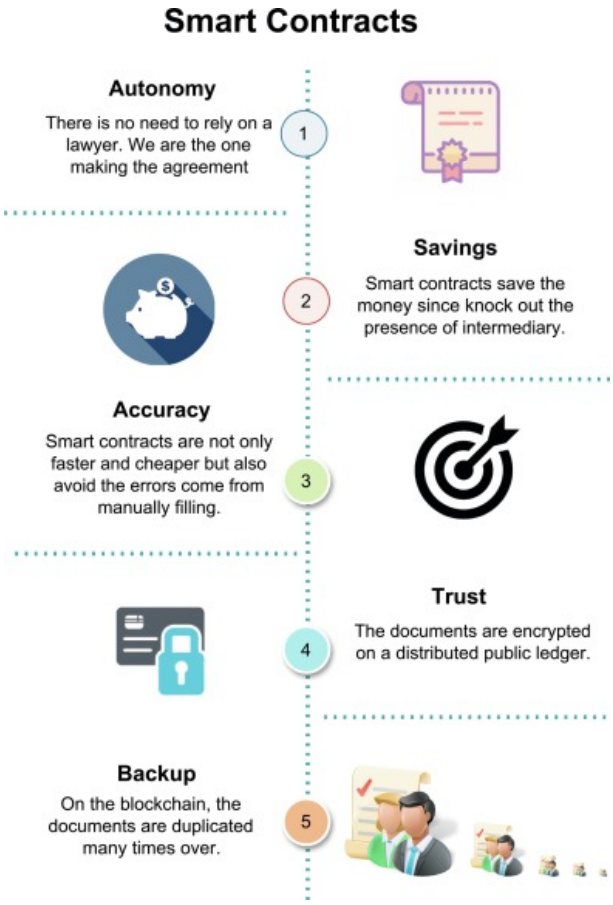


Figura 2.5: Vantagens dos *Smart Contracts*
Fonte: (Aggarwal & Kumar, 2021)

2.2.8 Plataformas *Blockchain*

De momento, existem inúmeras implementações da tecnologia *blockchain*. Neste secção, será realizada uma comparação entre algumas delas para ser possível compreender qual será a mais adequada para a utilização neste projeto.

A Figura 2.6 mostra a comparação entre as principais *blockchains* existentes em 2020 aos olhos de Anwar (2020).

De acordo com o estudo de Anwar (2020), é possível compreender que a *Ethereum* é a única *blockchain permissionless* da lista. De acordo com Shah (2022) *"Permissionless blockchain ledgers are notable for their radical decentralization. This includes mechanisms for full transparency of transactions, open-source development models, and a lack of central authority. Permissionless blockchains (...) benefits: Open Architecture (...); Transparency (...); Pseudo-Anonymity (...); Radical Decentralization (...); Network Resilience (...)"*.

Apesar do mecanismo de consenso da *Ethereum* estar na Figura 2.6 indicado como *Proof of Work* (PoW), recentemente a plataforma mudou o seu mecanismo de consenso para *Proof of Stake* (PoS) para reduzir os altos consumos de energia do PoW (Ethereum, 2023a).

Em março de 2023, Lawton (2023) publicou a sua versão de *"Top 9 blockchain platforms to consider in 2023"*. Tendo em conta a data da sua publicação, é possível obter de uma forma mais atualizada as

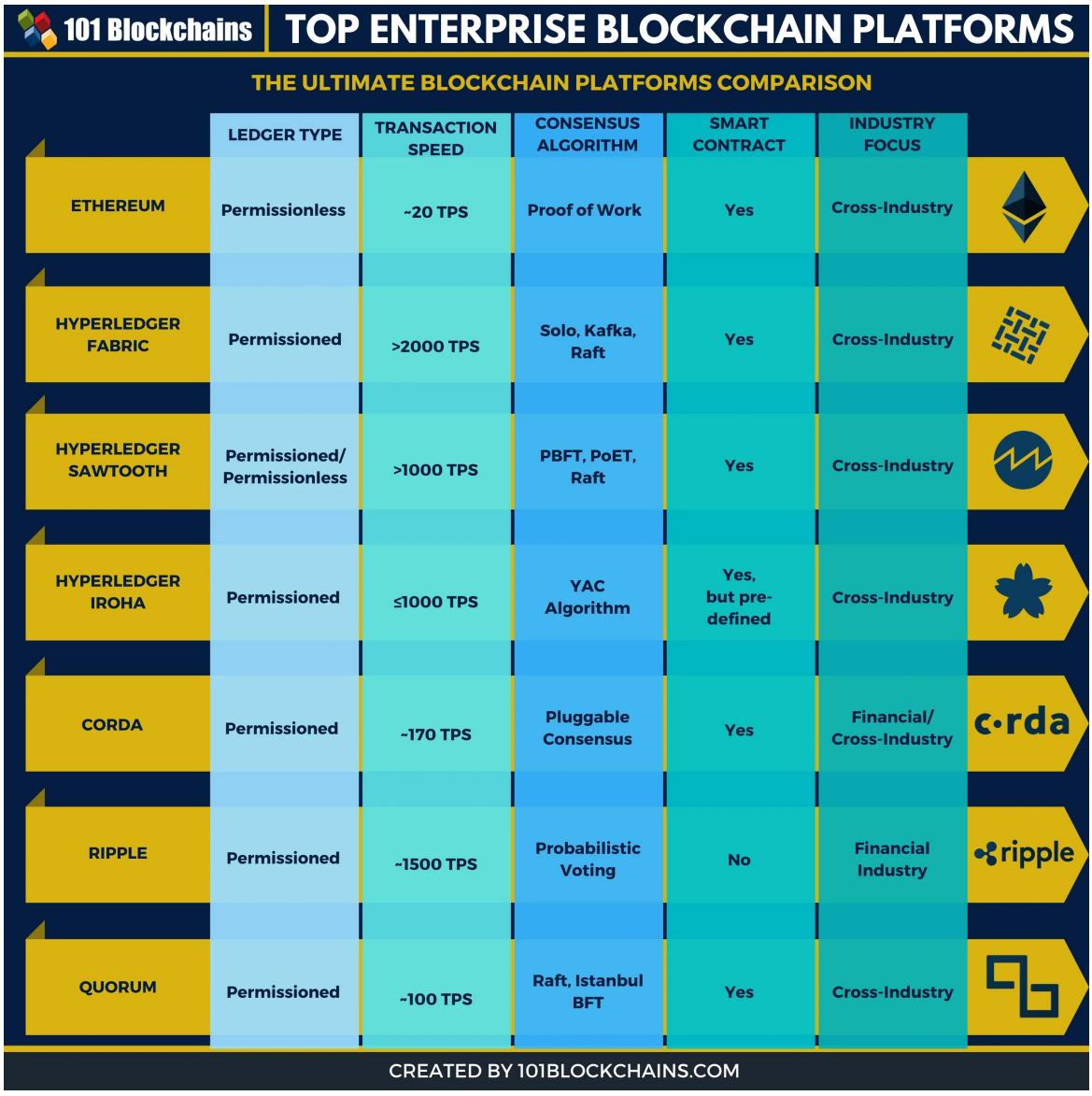


Figura 2.6: Comparação entre *Blockchains*
Fonte: (Anwar, 2020)

informações sobre as plataformas que estão ao dispor. As plataformas apresentadas são: *Ethereum; IBM Blockchain; Hyperledger Fabric; Hyperledger Sawtooth; R3 Corda; Tezos; EOSIO; Stellar; ConsenSys Quorum;* não serão analisadas as 9 plataformas indicadas no artigos, mas sim, três das mesmas.

Ethereum

O *Ethereum* é uma das plataformas blockchain mais antigas e estabelecidas, introduzida em 2013. Oferece uma blockchain descentralizada que suporta contratos inteligentes. Embora tenha vantagens em termos de descentralização, o *Ethereum* enfrenta desafios como tempos de processamento lentos e custos de transação mais altos em comparação com outras plataformas. Além de ser usado como uma plataforma para aplicações empresariais, o *Ethereum* possui a sua própria criptomoeda chamada *ether*. A comunidade do *Ethereum* é ativa e tem adotado o mecanismo de consenso de prova de participação *PoS*, que é mais eficiente em termos de consumo de energia. Além disso, a comunidade está a trabalhar num mecanismo de fragmentação para aumentar a capacidade de armazenamento, melhorar a escalabilidade e reduzir as taxas de rede. Essas atualizações são esperadas para serem implementadas gradualmente a partir de 2023 até 2024 (Lawton, 2023).

IBM Blockchain

A *IBM Blockchain* é uma rede *blockchain* privada e descentralizada que tem obtido sucesso principalmente com clientes empresariais que possuem menor aversão ao risco. Ela se destaca por permitir uma integração mais eficiente com tecnologias *cloud* em comparação com outras redes descentralizadas (Lawton, 2023).

A plataforma de desenvolvimento da *IBM Blockchain* foi projetada para ser flexível, funcional e personalizável. A empresa também se concentrou em criar uma interface amigável para simplificar tarefas críticas, como configuração, teste e implantação rápida de *smart contracts* (Lawton, 2023).

A *IBM* tem visto progressos significativos em setores como serviços financeiros, bancários e cadeia de fornecimentos. Exemplos de aplicações *blockchain* bem-sucedidos incluem o *IBM Food Trust*, que registrou mais de 18 milhões de transações representando mais de 17.000 produtos, e a *Blockchain Community Initiative* na Tailândia, que oferece suporte a serviços como obrigações de pagamento e leilões empresariais para 22 bancos tailandeses (Lawton, 2023).

Hyperledger Fabric

O *Hyperledger Fabric* é um conjunto de ferramentas para criar aplicações *blockchain*, desenvolvido pela *Linux Foundation* com foco em casos de uso empresariais. Ela possui uma arquitetura modular e suporta *blockchain* privadas, oferecendo melhorias em segurança e velocidade. Além disso, possui um modelo de *smart contract* aberto e flexível, capaz de suportar diferentes modelos de dados. O *Hyperledger Fabric* também prioriza a privacidade dos dados, permitindo a segregação de transações em canais e a partilha de informações privadas de forma seletiva. Ela se destaca por transações de alta velocidade e baixa latência. A comunidade em torno da *Hyperledger Fabric* está ativamente a trabalhar na melhoria do *software*, adicionando recursos relacionados a algoritmos de consenso, opções de privacidade para conformidade com o *General Data Protection Regulation* e melhorias operacionais (Lawton, 2023).

2.3 Aplicação da *Blockchain* no domínio jurídico

Esta secção ficou reservada para a apresentação de artigos publicados que de certa forma se assemelham com esta dissertação. Artigos que buscam encontrar uma solução de utilização da *blockchain* num contexto jurídico.

***Multisource traffic incident reporting and evidence management in Internet of Vehicles using machine learning and blockchain* (Philip & Saravanaguru, 2023)**

Este artigo discute como a tecnologia de *machine learning* e *blockchain* podem ser usadas para gerir relatos e evidências de incidentes de tráfego num ambiente de Internet das Coisas (IoT). O autor do artigo sugere que a combinação dessas tecnologias pode ser utilizada para recolher e validar informações de múltiplas fontes, como veículos conectados e câmaras de trânsito, e armazená-las de maneira segura e confiável usando *blockchain*. Além disso, o artigo apresenta um sistema proposto que usa técnicas de *machine learning* para analisar essas informações e fornecer alertas para potenciais incidentes. Ele argumenta que essa abordagem poderia ajudar a melhorar a segurança no trânsito e a eficiência na gestão de incidentes.

***LEChain: A blockchain-based lawful evidence management scheme for digital forensics* (Li et al., 2021)**

"(...) Lawful evidence management in digital forensics is of paramount importance in police investigations because such evidence is used to convict suspects of crimes. (...). Digital forensics is a legal procedure to collect, analyze, store, and report digital evidence. Such evidence is essential to police investigations because it illustrates the facts of cyber-crime scenes and connects suspects to criminal activities. (...)" Este artigo fala numa solução chamada *LEChain* que é um *blockchain-based lawful evidence management scheme* que visa supervisionar todo o fluxo de evidências e todos os dados do tribunal, como, por exemplo votos e resultados de julgamento, desde a recolha de evidências e o acesso durante as investigações

policiais até ao voto dos jurados em julgamentos. O artigo aborda a utilização *"short randomizable signatures"* como forma de autenticação anónima para proteger a privacidade das testemunhas na recolha de evidências. Também, garante um controlo de acesso refinado aos dados das evidências e do tribunal com base na criptografia com políticas de atributos.

***The Application of Blockchain of Custody in Criminal Investigation Process* (Tsai, 2021)**

Este artigo concentra se na aplicação de uma *blockchain of custody framework* no processo de investigação criminal para melhorar a segurança e transparência das evidências digitais. A *framework* é construida com base nos *smart contracts* da plataforma *Ethereum* para garantir a autenticidade e integridade das evidências digitais durante as várias etapas da investigação, incluindo a investigação preliminar, gestão de casos e fases judiciais. O papel dos investigadores também é considerado na *framework* para implementar o controlo de acessos à criação, transferência e modificação das evidências. Esta *framework* proposta foi testada numa *blockchian Ethereum* privada e os resultados mostraram que ela pode impedir a adulteração ou "contaminação" das evidências digitais, garantindo a sua defensibilidade legal através de uma gestão eficaz de privilégios.

Na Figura 2.7 é possível visualizar o modelo desta *framework*.

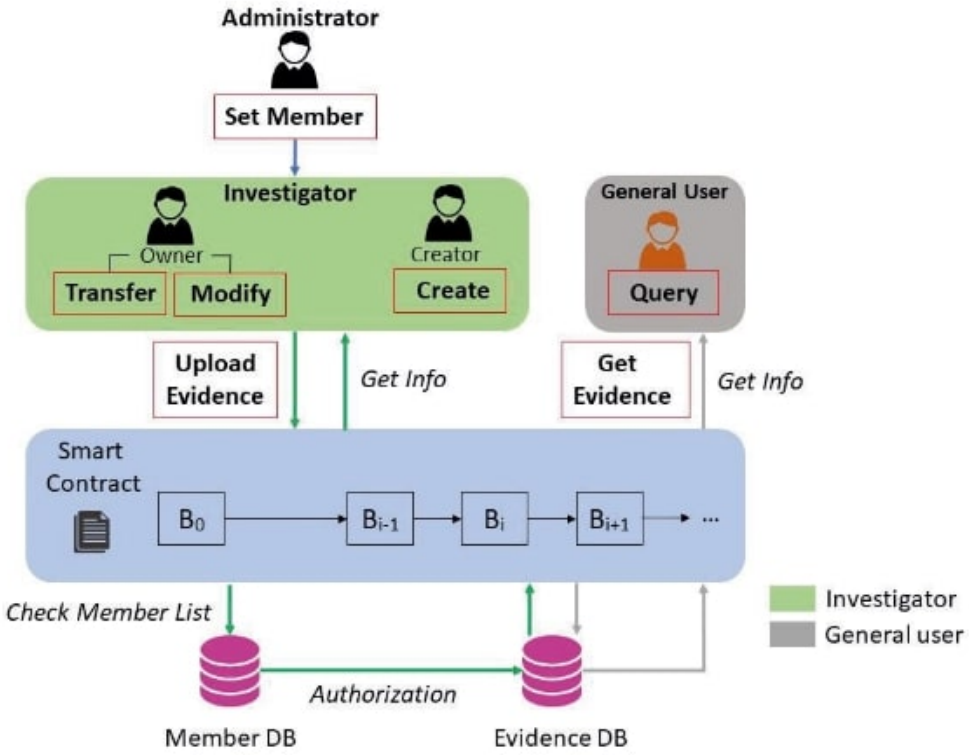


Figura 2.7: *Blockchain of custody framework*
Fonte: (Tsai, 2021)

***CRAB: Blockchain Based Criminal Record Management System* (Tasnim et al., 2018)**

Este artigo apresenta um sistema de armazenamento de registos criminais com o uso da tecnologia *blockchain* para armazenar os dados, o que ajuda a atingir integridade e segurança. O sistema *CRAB*, nome dado pelos autores, apresenta formas pela qual as autoridades podem manter eficientemente os registos dos criminosos. As autoridades (por exemplo, agências de aplicação da lei e tribunais) serão capazes de adicionar e aceder dados criminais. Os utilizadores gerais (por exemplo, organizações e/ou indivíduos selecionados, aeroportos, centros de aplicação de vistos, etc.) terão acesso aos dados para poderem procurar registos criminais. O

artigo propõe um *"CRAB-Protocol"* para atingir a integridade e segurança do armazenamento efetuado. Uma *overview* do protocolo é feita na Figura 2.8 com os passos descritos na Tabela 2.1:

Passos	**Descrição**
Step 1	*Sender* acessa o sistema com o uso do *ID* e *Password*
Step 2	*Sender* assina digitalmente U_{D} (Dados Criminais) e envia para FU (Modulo de autenticação, verificação e encriptação de dados)
Step 3	V_{D} (Dados Criminais Verificados) são encriptados e enviados para *Data Storage* com CID (*Criminal identification data*)
Step 4	T_{D} (Transaction data) é enviado para a *blockchain*
Step 5	B_{id} (Numero do Bloco) é enviado da *blockchain* para FU
Step 6	UAD'(Consiste em CID,B_{id} e *Enc(Key)*) é enviado para o *Local Server*
Step 7	*Receiver* acessa o sistema com o uso do *ID* e *Password*
Step 8	FU faz um pedido de dados ao *Local Server* com o CID
Step 9	UAD' é retornado ao FU
Step 10	FU faz um pedido de dados ao *Data Storage* com o CID
Step 11	*Data Storage* envia a FU os dados encriptados
Step 12	T_{D} é enviado para a *blockchain*
Step 13	B_{id} é retornado a FU
Step 14	FU envia UAD' atuailizado para o *Local Server*
Step 15	*Decrypted data* é enviado para o *receiver*

Tabela 2.1: Passos envolvidos no *CRAB Protocol* Fonte: (Tasnim et al., 2018)

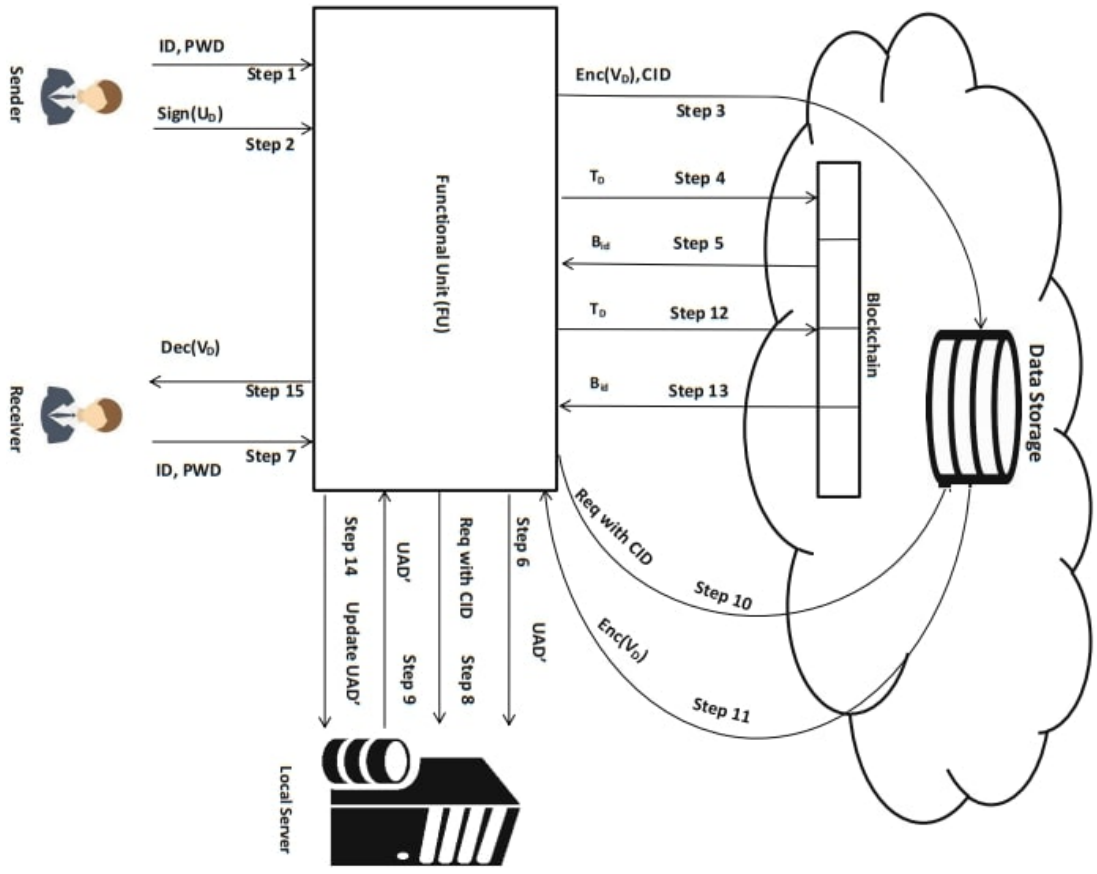


Figura 2.8: *CRAB Protocol*
Fonte: (Tasnim et al., 2018)

***Hala Systems: Blockchain Case Study for Saving Lives and Mitigating War Crimes* (Citizens-Information, 2021)**

A *Hala Systems* é um caso de estudo de como a tecnologia *blockchain* pode ser utilizada para salvar vidas e mitigar crimes de guerra. A empresa desenvolveu uma plataforma baseada em *blockchain* para registar evidências imutáveis de eventos e ações relacionados às condições na Síria. Além disso, a equipa da *Hala* está trabalhar em soluções para alcançar cidadãos que não têm acesso a telemóveis ou energia, como o uso de sirenes modificadas com componentes de acesso remoto. A empresa continua a realizar pesquisas para desenvolver a sua solução, com o objetivo de torná-la ainda mais escalável e eficaz no registo de informações imutáveis e precisas.

2.4 *Ethereum*

O *Ethereum* tem como objetivo fornecer uma infraestrutura flexível e poderosa para o desenvolvimento de *dApps*, oferecendo um ambiente seguro e confiável para a execução de *smart contracts*. É um protocolo que oferece uma alternativa para a criação de aplicações descentralizadas. Ele permite o desenvolvimento rápido, a segurança para aplicações pequenas e a interação eficiente entre diferentes aplicações. O *Ethereum* utiliza uma *blockchain* com uma linguagem de programação *Turing-complete* embutida (*Solidity, Vyper*, etc), o que permite a criação de *smart contracts* e aplicações descentralizadas com próprias regras arbitrárias para propriedade, formatos de transação e funções de transição de estado. Além disso, ele oferece maior poder e flexibilidade em comparação com o *Bitcoin*, permitindo a criação de sistemas de moedas, reputação e outros protocolos (Ethereum, 2023c).

2.4.1 O que são *dApps*

As *dApps*, ou aplicações descentralizadas, são aplicações digitais que utilizam a tecnologia *blockchain* para manter os dados dos utilizadores fora do controlo de organizações. Assim como as criptomoedas, as *dApps* são descentralizadas. Elas armazenam dados numa rede de computadores chamados *"nodes"* e são projetadas para oferecer uma variedade de serviços, como redes sociais, jogos, ferramentas de produtividade e serviços financeiros descentralizados (George, 2023).

O *Ethereum* tem sido a plataforma dominante para hospedar *dApps* e facilitar sua criação. As *dApps* são geralmente de código aberto, permitindo que qualquer pessoa veja e use o código subjacente (George, 2023).

As dApps oferecem aos utilizadores maior segurança, uma vez que os criadores da aplicação não têm controlo sobre seu uso e não podem vender os dados dos utilizadores. Além disso, existem as *DAOs* (Decentralized Autonomous Organizations), organizações autónomas descentralizadas, que podem ser vistas como uma espécie de *dApp*. Elas têm como objetivo utilizar um arranjo complexo de *smart contracts* para realizar as funções de uma organização tradicional sem a necessidade de executivos e hierarquias empresariais. Elas determinam políticas exclusivamente por meio de um sistema de votação ponderada, no qual os membros que guardam mais *tokens* possuem maior poder de voto. A ideia subjacente é que aqueles que comprometeram mais fundos com uma *DAO* terão mais probabilidade de participar honestamente, pelo bem da organização (George, 2023).

2.4.2 A *Blockchain Ethereum*

A *blockchain Ethereum* é baseada numa estrutura de registo distribuído, onde todas as transações e *smart contracts* são registados de forma transparente e imutável. Isso garante a segurança e a confiabilidade das transações, permitindo que os *dApps* operem de maneira confiável e descentralizada.

Ether

Ether, também conhecido como *ETH*, é a moeda nativa da *blockchain Ethereum*. O ETH possui várias utilizações no ecossistema *Ethereum*, como pagamento de taxas para atividades na *blockchain Ethereum*.

Enviar ETH de uma pessoa para outra, por exemplo, requer que o remetente gaste uma quantidade de ETH para enviar a transação pela blockchain. É essencialmente um pagamento para aqueles que fornecem recursos para manter a blockchain em funcionamento (Ethereum, 2023c; Cointelegraph, 2023).

Gas

Gas é a unidade que mede o esforço computacional necessário para realizar operações na rede *Ethereum*. Toda transação na *Ethereum* requer recursos computacionais e, portanto, exige o pagamento de uma taxa em *Gas*. Essa taxa é necessária para executar a transação, independentemente de ela ser bem-sucedida ou não (Ethereum, 2023b).

Accounts

"(...) An Ethereum account is an entity with an ether (ETH) balance that can send transactions on Ethereum. Accounts can be user-controlled or deployed as smart contracts. (...)" (Ethereum, 2023b).

Existem dois tipos de *Ethereum accounts* (Ethereum, 2023b):

- *Externally-owned account* (EOA): É constituída por um par de chaves criptográficas: pública e privada. Estas ajudam a provar que uma transação foi realmente assinada pelo remetente e evitam falsificações. A sua chave privada é a que utiliza para assinar transações, pelo que lhe confere custódia sobre os fundos associados à sua conta.
- *Contract Account*: O endereço do contrato é normalmente fornecido quando um contrato é implementado na cadeia de blocos Ethereum. O endereço provém do endereço do criador e do número de transações enviadas a partir desse endereço.

Na Figura 2.9 está representada a constituição de uma conta de *Ethereum* (Ethereum, 2023b):

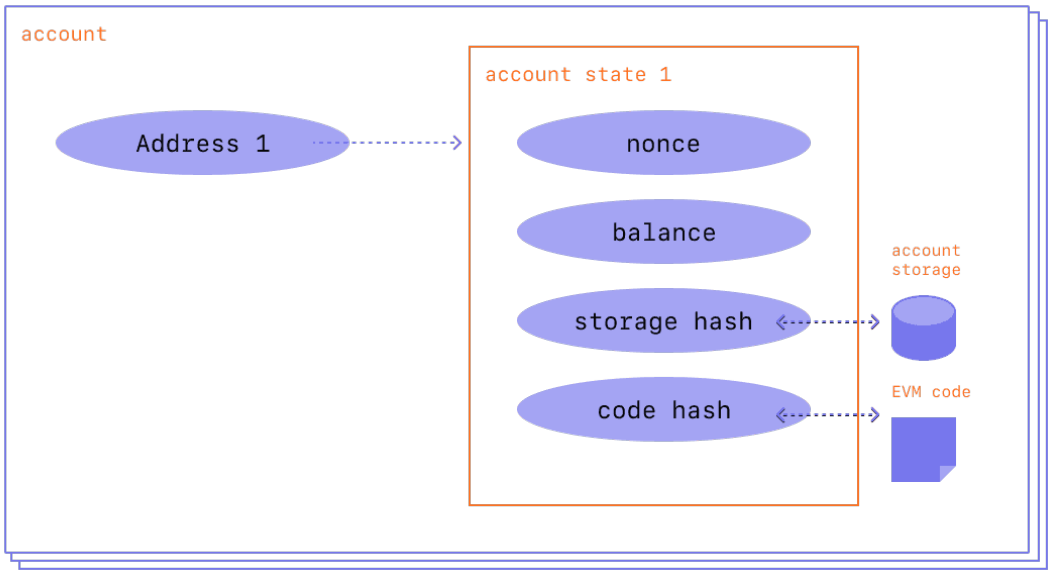


Figura 2.9: *Ethereum account*
Fonte: (Ethereum, 2023b)

- nonce: Contador que indica o número de transações enviadas a partir de uma *Externally-owned account* (EOA) ou o número de contratos criados por uma *contract account*. Apenas uma transação com um determinado *nonce* pode ser executada para cada conta, protegendo contra ataques de repetição em que as transações assinadas são repetidamente difundidas e re-executadas.
- *balance*: O número de *wei* pertencentes a este endereço. *Wei* é uma denominação de *ETH* e há 1e+18 *wei* por *ETH* ou seja uma *wei* equivale a 0.000000000000000001 *ETH*.

- *code hash*: Refere-se ao código de uma conta na máquina virtual *Ethereum* (*EVM*).

- *storage hash*: Conhecido como *hash* de armazenamento. Uma *hash* de 256 bits do *node* raiz de uma *Merkle Patricia Trie* que codifica o conteúdo de armazenamento da conta. Esta *trie* codifica a *hash* do conteúdo do armazenamento desta conta e está vazia por defeito.

Transactions

"(...) An Ethereum transaction refers to an action initiated by an externally-owned account, in other words an account managed by a human, not a contract. (...)" (Ethereum, 2023b).

Enviar *ETH* para alguém serve como exemplo de uma transação. Isso altera a rede para refletir a transferência de propriedade do *ETH*, o que requer a participação de *miners* na *blockchain*, que são pagos pelo trabalho com uma taxa chamada taxa de gás *(gas fee)*. Aqueles que iniciam as transações são os responsáveis por pagar as taxas sobre essas transações. As transações na rede *Ethereum* são publicamente visíveis em exploradores de blockchain do *Ethereum* (Cointelegraph, 2023).

Na Figura 2.10 estão representados os elementos de uma transação na *Ethereum* (Ethereum, 2023b):

Elements	Description
Recipient	The receiving address
Signature	The identifier of the sender
Data	The arbitrary data field is an optional field that can be filled in with any data
Value	The amount of ETH to be sent from the sender to the receiver
GasLimit	The maximum amount of gas units that the transaction can consume
MaxPriorityFeePerGas	The maximum amount of gas that can be given to the miner as a tip
MaxFeePerGas	The maximum amount of gas that can be offered in exchange for the transaction (including maxPriorityFeePerGas and baseFeePerGasi.e., reserve price determined by the blocks)

cointelegraph.com

Figura 2.10: Constituintes de uma transação da *Ethereum*
Fonte: (Cointelegraph, 2023)

2.4.3 *Ethereum Virtual Machine*

De acordo com Cointelegraph (2023); Ethereum (2023b) *"(...) EVM is a computation engine that functions as a decentralized computer with millions of applications that can be executed. The EVM is the core framework of the Ethereum blockchain. It essentially dictates how the overall system runs and maintains itself, taking changes into account. (...)"*

A *EVM* é uma componente fundamental da rede *Ethereum*. É uma máquina virtual *Turing complete* que funciona como um ambiente de execução para os *smart contracts* na blockchain *Ethereum*. A *EVM* foi projetada para ser um ambiente seguro, determinístico e isolado, e é usada por todos os *nodes Ethereum* para manter o consenso na *blockchain* (Cointelegraph, 2023; Ethereum, 2023b).

2.5 Desenvolvimento de *dApps* na *Ethereum*

O desenvolvimento de aplicações descentralizadas na plataforma *Ethereum* envolve a criação de aplicações que operam na rede *Ethereum* e executam *smart contracts* escritos em linguagens como *Solidity*.

O desenvolvimento de *dApps* esta sujeito ao seguintes passos: Escolha da linguagem de programação; Configuração do ambiente de desenvolvimento; Escolha da *framework*; Escrita do(s) *smart contract(s)*; Testes; Implementação do(s) *smart contract(s)* na rede de testes; Desenvolver a interface de utilizador; Implementação na *Mainnet*; Manutenção e atualizações.

2.5.1 Linguagem de programação

A escolha da linguagem de programação é fundamental, pois será a base para todo o processo de desenvolvimento. Neste capítulo, é feita uma abordagem sobre as diversas linguagens de programação utilizadas para o desenvolvimento de *smart contracts* para a *blockchain Ethereum*.

De acordo com *"Top 5 Blockchain Programming Languages for Smart Contracts"* de Kashettar (2023), compreendemos que existem cinco linguagens de programação amplamente utilizadas no desenvolvimento de *smart contracts*. Das cinco apresentadas somente três são linguagens de programação aptas para serem executadas numa *blockchain Ethereum* e de acordo com Ethereum (2023b); Chainlink (2023) as duas linguagens mais ativas e sustentadas são *Solidity* e *Vyper*.

- *Solidity*: É uma linguagem de programação poderosa projetada especificamente para escrever *smart contracts* na *blockchain Ethereum*. Com *Solidity*, os *developers* podem definir as regras e o comportamento de aplicações descentralizadas (dApps). Foi apresentada publicamente pela primeira vez em novembro de 2014 na *Devcon0*. A gestão de versões para *Solidity* foi incluído no código em 9 de julho de 2015, marcando a versão 0.0.1. A linguagem de programação *Solidity* é um projeto de código aberto, gerido por uma equipa central e patrocinado pela *Ethereum Foundation* (Solidity, 2023).
  - Prós (Chainlink, 2023):
    - Alto nível e Turing-completa - significa que os *developers* podem escrever código de forma mais rápida.
    - Sintaxe do *Solidity* também é semelhante ao *JavaScript*.
    - Medidas de proteção embutidas, o que pode evitar erros custosos.
    - É a linguagem mais amplamente usada no *Web3*. Os *developers* de *Solidity* têm acesso a mais bibliotecas e ferramentas, melhor documentação e suporte em fóruns de *developers*.
  - Contras (Chainlink, 2023):
    - Para *developers* com pouca experiência em programação orientada a objetos, a sintaxe do *Solidity* pode ser confusa.
    - Há algumas peculiaridades no desenvolvimento em *Solidity* com as quais os *developers* podem não estar acostumados, como a falta de suporte nativo para números decimais.
- *Vyper*: É a mais recente linguagem de programação para blockchain e foi criada especificamente para o desenvolvimento de *smart contracts* na *blockchain Ethereum*. Ela é similar ao *Solidity*, mas

com um foco na simplicidade e segurança. Devido ao seu foco ser a segurança contém um limitado número de recursos disponíveis para os *developers* (Kashettar, 2023).

- Prós (Chainlink, 2023):
  - Tem uma aparência e sensação semelhante ao *Python* - torna um dos melhores pontos de partida para *developers* de *Python*.
  - Também possui uma linguagem e implementação de compilador simples, o que contribui para a legibilidade e auditabilidade do código.
  - Conta com muitas das mesmas ferramentas e recursos disponíveis para o *Solidity*.
- Contras (Chainlink, 2023):
  - Ainda não possui o amplo suporte da comunidade que o *Solidity* possui.
  - Existem ferramentas nativas do *Solidity* que ainda não estão disponíveis no *Vyper*.
  - Carece de recursos como modificadores, herança de classes e chamadas recursivas, e a linguagem de programação não é Turing-completa.

2.5.2 *Framework Truffle*

Truffle é uma *framework* para construir, testar e implantar aplicações na rede Ethereum, fundado por Tim Coulter. *"(...) A world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier. (...)"* (Suite, 2023c).

Algumas das principais características do *Truffle* são (Suite, 2023c):

- Compilação, vinculação, implementação e gestão de binários de *smart contracts* integrados.
- Depuração avançada com *breakpoints*, análise de variáveis e funcionalidade de passo a passo.
- Possibilidade do uso de *console.log* em *smart contracts*.
- Implementações e transações através do *MetaMask* com o *Truffle Dashboard* para proteger a mnemónica.
- Executor de *scripts* externos que executa *scripts* dentro de um ambiente *Truffle*.
- Consola interativa para comunicação direta com *smart contracts*.
- Testes automatizados de contratos para desenvolvimento rápido.
- *Framework* de implementação e migrações *scriptável* e extensível.
- Gestão de redes para implementação em qualquer número de redes públicas e privadas.
- Gestão de pacotes com *NPM* (*Node Package Manager*), utilizando o padrão *ERC190*.
- Pipeline de construção configurável com suporte para integração estreita.

2.5.3 Ganache

Ganache é uma *blockchain* local privada para o desenvolvimento rápido de aplicações distribuídas *Ethereum* e *Filecoin*. *Ganache* pode ser utilizado em todo o ciclo de desenvolvimento; permitindo assim que se desenvolva, implemente e teste os *dApps* num ambiente seguro e determinístico (Suite, 2023b).

As principais características do *Ganache* são (Suite, 2023b):

- Os blocos podem ser minerados instantaneamente, *on-demand* ou num intervalo.
- Suporte *Ethereum JSON-RPC*.
- *Fork* qualquer rede *Ethereum* sem esperar para sincronizar.
- *Forking* de *Mainnet* e *testnet* de configuração zero.
- *Impersonate* de qualquer conta (sem necessidade de chaves privadas!).

2.5.4 Drizzle

Drizzle é um conjunto de bibliotecas *front-end* que facilita e torna mais previsível a criação de interfaces para aplicações descentralizadas. O núcleo do *Drizzle* é baseado em *Redux*, o que lhe oferece acesso às excelentes ferramentas de desenvolvimento associadas ao *Redux*. (Suite, 2023a).

As principais características do Drizzle são (Suite, 2023a):

- Dados de contrato totalmente *reactive*, incluindo estado, eventos e transações.
- Declarativo, para não desperdiçar ciclos com dados desnecessários.
- Mantém o acesso à funcionalidade subjacente. O Web3 e os métodos do contrato mantêm-se.

3. Análise e Modelação

Este capítulo serve de base ao processo de criação de uma aplicação descentralizada de demonstração com o intuito de fornecer uma possível solução para o problema apresentado.

Aqui serão descritos alguns dos requisitos e casos de uso a serem desenvolvidos para demonstrar a rastreabilidade no processo jurídico de registo de evidências, fornecer garantias no armazenamento das mesmas em formato digital e a possibilidade de validar contra a adulteração.

3.1 Objetivo

Como discutido em capítulos anteriores, o problema persiste na forma como as evidências são armazenadas e como são salvaguardadas no mundo real, juntamente com a questão de 'quem' poderá ou deverá ser responsabilizado por problemas nas comprovações da verdade e veracidade de uma evidência física. O objetivo do desenvolvimento consiste na criação de uma aplicação descentralizada destinada a demonstrar a rastreabilidade no processo de registo de evidências em formato digital. No capítulo 2.1 "Registo de Evidências", foram mencionadas as etapas de uma *Chain of Custody*, e foi baseado nessas informações que foi montado um sistema de forma a que respondesse as necessidades dessa cadeia.

O objetivo final é estabelecer um ecossistema com o propósito de fornecer às entidades legais um meio confiável de armazenamento de arquivos digitais e registo de evidências. O ecossistema pode ser visto em 3 partes/camadas: a camada de negócio (*blockchain smart contracts*), camada do *front-end* e a camada do *back-end* .

A camada principal ou camada de negócio, será desenvolvida em *smart contracts*. É aqui que vão ser definidas as regras de negócio da aplicação, capazes de garantir que não poderá ocorrer adulteração no processo de registo de evidências o que, por sua vez, permite também aplicar a rastreabilidade em todo o processo. Será também o dever dos *smart contracts* de armazenar os dados sobre os utilizadores e as evidências.

O *back-end* terá como principal objetivo a gestão do armazenamento de arquivos digitais (um arquivo digital poderá representar uma evidência como um todo). Devido a possibilidade do arquivo ter um tamanho superior ao permitido pelos *smart contracts*, existe a necessidade de montar o *back-end* com este propósito. O objetivo neste ponto será garantir a integridade dos documentos armazenados, utilizando uma combinação de *back-end* e *smart contracts* (camada de negócios).

No *back-end*, também será criado um "controlador" da base de dados, devido à necessidade de garantir ao máximo a integridade dos arquivos a serem armazenados. Para isso, serão utilizados *triggers* para monitorizar o conteúdo da nossa base de dados, permitindo-nos detetar quaisquer mudanças indesejadas.

O *front-end* terá o objetivo de comunicar com ambas as partes (*back-end e smart contracts*) para apresentar ao utilizador final aquilo que se encontra armazenado nos mesmos.

3.2 Engenharia de Requisitos

A *Requirements Engineering* (Engenharia de Requisitos) (RE) é um processo fundamental na área de Tecnologia da Informação (TI) que se concentra na definição, documentação e manutenção dos requisitos de um projeto de engenharia. O principal objetivo é compreender as necessidades dos clientes. Isso resulta em soluções mais claras e eficazes para os problemas (Betrybe, 2022).

A RE também desempenha um papel crucial na validação das especificações do projeto, gerindo os requisitos à medida que evoluem no sistema. Isso garante que os processos ocorram de maneira ágil e eficiente. A RE utiliza princípios, ferramentas, métodos e notações comprovados para descrever o comportamento desejado de um sistema e suas restrições. Em essência, ela serve como um guia disciplinado para criar soluções de TI alinhadas com as necessidades do cliente (Betrybe, 2022).

3.2.1 Requisitos funcionais

Os **requisitos funcionais** são declarações das funcionalidades do sistema que devem fornecer como o sistema deverá responder a cada entrada e como se deverá comportar em dadas situações (Sommerville, 2015). Estes encontram-se classificados pela metodologia ***MoSCoW*** que identifica o grau de importância dos requisitos.

- ***M****ust have* (Deve): Representa as funcionalidades ou requisitos que são essenciais e obrigatórios para o sucesso do projeto. São os itens que não podem ser comprometidos e devem ser implementados.
- ***S****hould have* (Deveria): Representa as funcionalidades ou requisitos que são importantes, mas não críticos para a entrega inicial do projeto. Eles podem ser adiados para uma versão posterior, se necessário.
- ***C****ould have* (Poderia): Representa as funcionalidades ou requisitos que são desejáveis, mas não essenciais. Eles podem ser incluídos se houver tempo e recursos disponíveis.
- ***W****on't have*(Não Deve): Representa as funcionalidades ou requisitos que foram discutidos, mas decidiu-se que não serão implementados nesta fase do -projeto. Eles são excluídos deliberadamente.

Na Tabela 3.2, estão listados os requisitos funcionais encontrados após uma breve análise das funcionalidades principais necessitadas.

ID	Descrição	MoSCoW
RF 01	Como utilizador, quero criar um registo de uma evidência no sistema.	Must
RF 02	Como utilizador, quero ter a possibilidade de visualizar os registos de evidências efetuadas.	Must
RF 03	Como utilizador, quero ter a possibilidade de visualizar os registos de evidências publicas.	Must
RF 04	Como utilizador, quero ter a possibilidade de visualizar o histórico de alterações efetuadas sobre um registo.	Must
RF 05	Como utilizador, quero ter a possibilidade de identificar o utilizador responsável pela(s) atualização(ões) efetuadas sobre um registo.	Must
RF 06	Como utilizador, quero ter a possibilidade de incluir/anexar arquivo(s) digital(ais) num registo de evidência.	Must
RF 07	Como utilizador, quero ter a garantia de que os anexos digitais se encontram integralmente intactos.	Must
RF 08	Como utilizador, quero ter a possibilidade de efetuar o download dos arquivo(s) digital(ais) de um registo de evidência.	Must
RF 09	Como utilizador, quero ter a possibilidade de definir o tipo de classificação de uma evidência (publica ou apenas pessoas autorizadas).	Must
RF 10	Como utilizador, quero ter a possibilidade de adicionar entidades que têm acesso as evidências classificadas como "Apenas pessoas autorizadas"	Must
RF 11	Como *owner* de uma evidência, quero ter a possibilidade de transferir o *ownership* a um utilizador autorizado no sistema.	Must
RF 12	Como utilizador, quero ter a possibilidade de me registar no sistema.	Must
RF 13	Como administrador, quero ter a possibilidade de aprovar registos efetuados.	Must
RF 14	Como utilizador, quero ter a possibilidade de visualizar todos os utilizadores registados no sistema.	Must
RF 15	Como utilizador, quero ter a possibilidade de editar os meus dados.	Must
RF 16	Como utilizador, quero ter a possibilidade de visualizar as edições efetuadas aos registos de utilizadores.	Must
RF 17	Como utilizador, quero ter a possibilidade de efetuar o *login* no sistema via *username/password*.	Must
RF 18	Como administrador, quero ter controlo sobre a base de dados via *triggers*	Must

Tabela 3.2: Requisitos funcionais

3.2.2 Requisitos não funcionais

Os **requisitos não funcionais** são os constrangimentos às funcionalidades ou funções do sistema, como standards, restrições de tempo e restrições do processo de desenvolvimento, incluindo as propriedades comportamentais que o sistema deve ter, como desempenho, segurança e fiabilidade (Sommerville, 2015).

De seguida, foi efetuado um levantamento de requisitos não funcionais, os quais são apresentados na Tabela 3.4.

ID	Descrição	MoSCoW
RnF 01	Implementação de *Smart Contract* para a *blockchain ethereum* em *solidity*	Must
RnF 02	Implementação de *Back End* utilizado a *framework NestJS*.	Must
RnF 03	Implementação de *Front End* utilizado a *framework ReactJS*.	Must
RnF 04	Os dados devem ser persistidos na *blockchain* a exceção de arquivos digitais.	Must
RnF 05	Utilização de encriptação SHA-256 para os arquivos digitais.	Must
RnF 06	Utilização de uma base de dados não relacional (MongoDB) para armazenamento de arquivos digitais.	Must
RnF 07	O Sistema contem autorização *directly linked with MetaMask*.	Must

Tabela 3.4: Requisitos não funcionais

3.3 Casos de uso e diagrama de atividades

Os casos de uso descrevem as funcionalidades esperadas do sistema, focando no "o quê" em vez do "como". Podem ser representados de forma textual ou visual. Esta técnica ajuda a projetar o sistema sob a perspetiva do utilizador final, facilitando a comunicação do comportamento do sistema em termos compreensíveis para o utilizador (Paradigm, 2022).

O diagrama de caso de uso é uma representação visual geralmente simples, que resume as relações entre atores e sistemas. Não detalha a ordem das etapas para atingir os objetivos de cada caso de uso. É enfatizado que um diagrama de caso de uso deve ser simples e conter apenas algumas formas (Paradigm, 2022).

Os diagramas de casos de uso são úteis nas seguintes situações (IBM, 2021):

- **Antes de iniciar um projeto:** Criar diagramas de caso de uso para modelar um negócio, de forma que todos os participantes do projeto compartilhem uma compreensão dos trabalhadores, clientes e atividades do negócio.
- **Durante a recolha de requisitos:** Criar diagramas de casos de uso para captar os requisitos do sistema e apresentar aos outros o que o sistema deve fazer.
- **Durante as fases de análise e design:** Usar os casos de uso e atores dos diagramas de caso de uso para identificar as classes que o sistema requer.
- **Durante a fase de teste:** Usar os diagramas de casos de uso para identificar testes para o sistema.

Citando Lucidchart (2021), "(...) Os diagramas de atividade, junto com diagramas de caso de uso, são considerados diagramas de comportamento porque descrevem o que é necessário acontecer no sistema sendo modelado. Diagramas de atividade ajudam a unir as pessoas das áreas de negócios e de desenvolvimento de uma organização para entender o mesmo processo e comportamento. (...)".

Considera-se criar um diagrama de atividade para (Lucidchart, 2021):

- Demonstrar a lógica de um algoritmo.
- Descrever as etapas realizadas num caso de uso.

- Ilustrar um processo de negócio ou fluxo de trabalho entre utilizadores e o sistema.
- Simplificar e melhorar qualquer processo ao esclarecer casos de uso complicados.
- Modelar elementos de arquitetura de software, como método, função e operação.

3.3.1 Atores

No contexto de análise de sistemas e definição dos casos de uso, é importante analisar os atores. Os atores são partes externas ao sistema que interagem com o mesmo de alguma forma. Neste desenvolvimento foram identificados dois atores principais:

- Utilizadores: Os utilizadores identificam todos os participantes externos que vão utilizar a plataforma. Os utilizadores serão as entidades legais que estão envolvidas em casos jurídicos.
- Administradores: Os administradores, que por sua vez também poderão ser vistos como utilizadores, são as entidades legais responsáveis pelo sistema. Estes são os responsáveis pelos dados que requerem aprovação adicional, como por exemplo, validação dos utilizadores registados.

3.3.2 Diagrama de casos de uso de evidências

Para uma visualização mais focada e simplificada, foram criados dois diagramas de casos de uso separados: um diagrama aborda os casos de uso relacionados com o registo de evidências e o segundo diagrama aborda os casos de uso relacionados com o registo de utilizadores.

Na Figura 3.1 estão representados os casos de uso *UC* correspondentes ao processo de gestão das evidências.

Analisemos os casos de uso (UC) identificados:

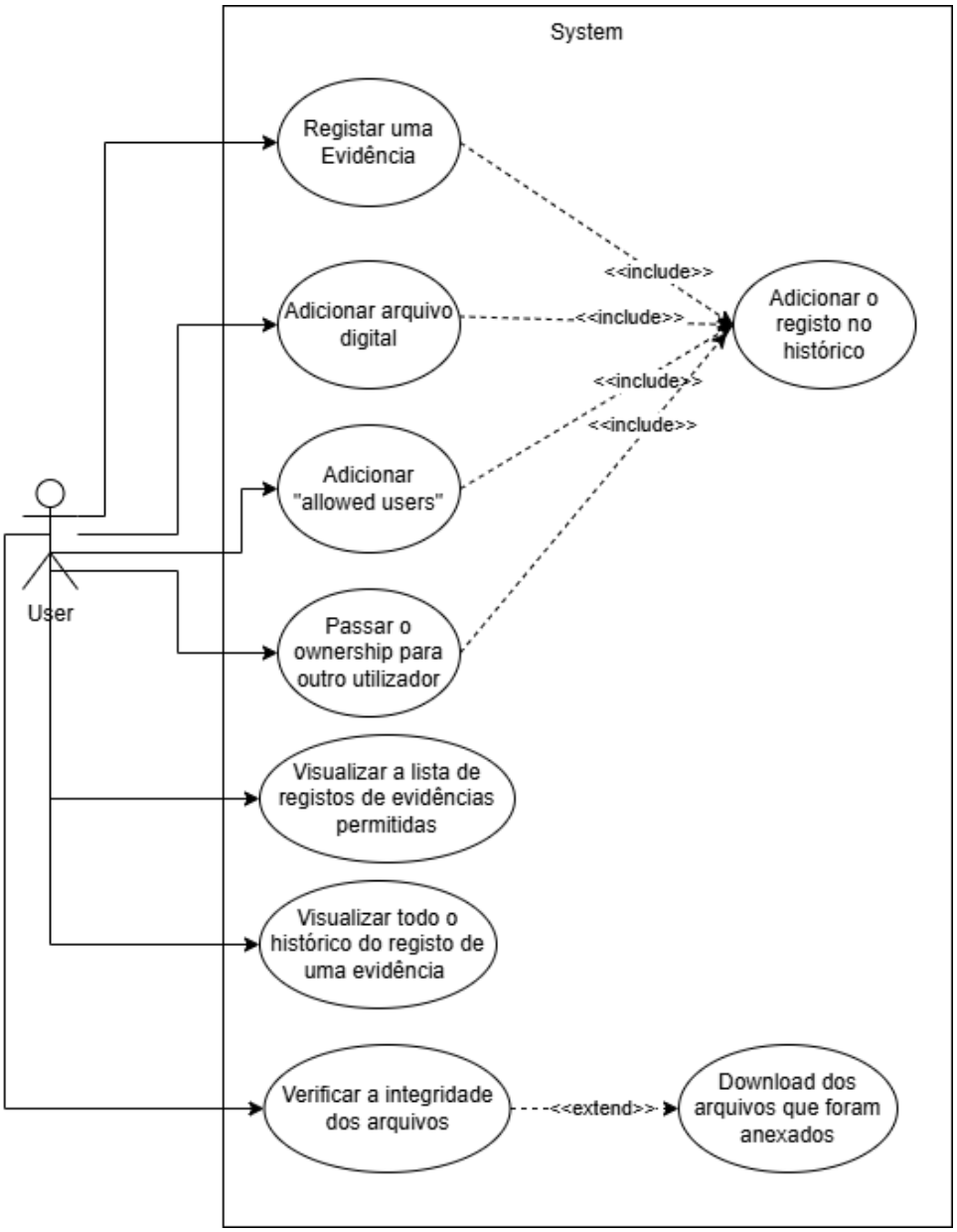


Figura 3.1: Diagrama de casos de uso (Evidências)

UC 01 - Registar uma evidência

Este caso de uso corresponde aos requisitos funcionais 01 e 09 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema, existe a capacidade de criar e registar uma evidência. Esse registo é armazenado na *blockchain*, juntamente com o identificador do utilizador que efetuou o registo. Automaticamente, esse registo é armazenado como o primeiro registo no histórico de registos da evidência em questão. O registo da evidência contém dados relacionados à própria evidência e também inclui informações sobre o tipo de classificação da evidência (por exemplo, "Apenas pessoas autorizadas").

Na Figura 3.2, está representado o diagrama de atividades correspondente ao caso de uso 01. É possível compreender que o utilizador deve estar registado na aplicação com o *MetaMask* (o *MetaMask* é um software de carteira de criptomoedas utilizado para interagir com a *blockchain Ethereum*, como será mencionado de forma mais detalhada no capítulo seguinte) para poder prosseguir com o registo. Em seguida, o utilizador preenche o formulário correspondente ao registo da evidência, este é validado no *front-end* e, após ser confirmado como válido, o registo pode ser guardado. Nesse momento, a transação é enviada para o *smart contract*, onde ocorrem diversas validações antes de o registo ser armazenado. O utilizador precisa estar registado no sistema para que a transação seja válida; caso contrário, o processo

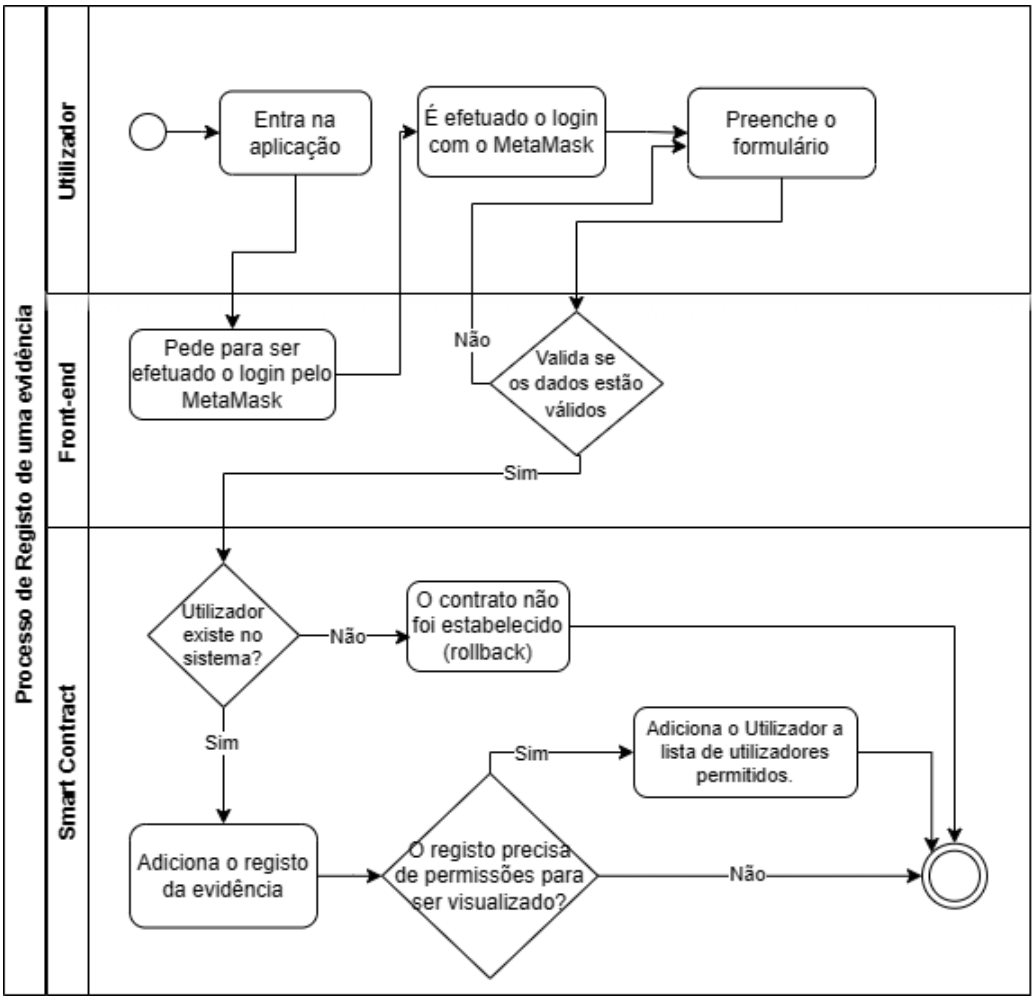


Figura 3.2: Diagrama de Atividades UC 01

é revertido. Se o utilizador existir, então é verificado o tipo de registo a ser adicionado. Se o registo for somente visível para utilizadores autorizados, o próprio utilizador é adicionado à lista de utilizadores permitidos para a visualização do registo.

UC 02 - Adicionar um arquivo digital

Este caso de uso corresponde ao requisito funcional 06 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema e sendo o proprietário de um registo de evidência (ser o proprietário significa que o registo está sob a sua responsabilidade), existe a capacidade de anexar um arquivo digital, independentemente do tipo (por exemplo, documento, imagem, vídeo, gravação de áudio), ao registo de uma evidência. O arquivo é armazenado fora da *blockchain*, a uma base de dados não relacional. Devido ao tamanho do arquivo, não seria possível armazenar todo o seu conteúdo na *blockchain*. É realizado o cálculo da *hash* sobre os dados presentes no arquivo, e a *hash* gerada é armazenada como um novo registo na *blockchain*. Além disso, os dados do utilizador que realizou essa adição são guardados para referência.

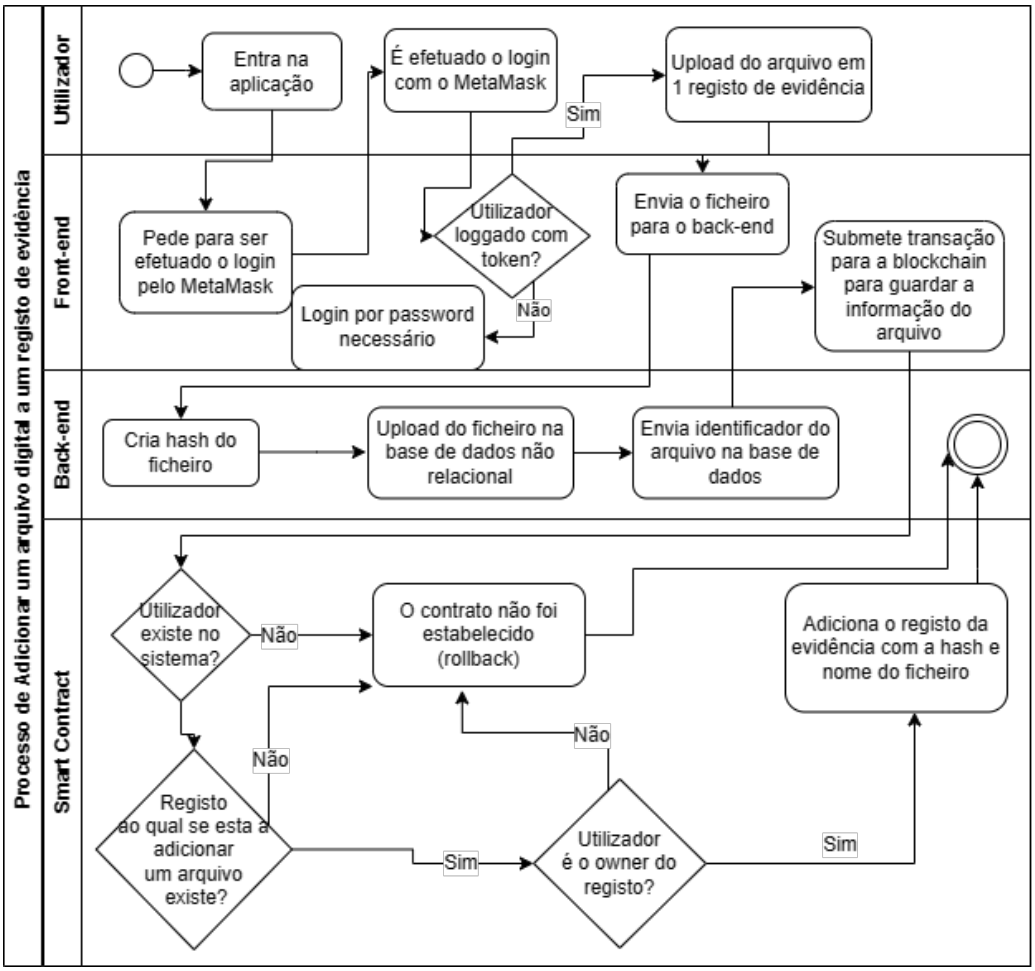


Figura 3.3: Diagrama de Atividades UC 02

Na Figura 3.3, está representado o diagrama de atividades correspondente ao caso de uso 02. Da mesma forma que no caso de uso anterior, o utilizador deve estar conectado à aplicação através do *MetaMask*. Aqui, o processo permanece o mesmo, incluindo a autenticação no *MetaMask*. Além disso, o utilizador deve efetuar o *login* no *back-end* utilizando o seu nome de utilizador e palavra-passe, que foram usados no registo (UC 08). Após isso, o utilizador pode selecionar os ficheiros que deseja adicionar ao registo de uma evidência e submetê-los. Os arquivos são enviados para o *back-end*, onde será gerada a *hash* correspondente ao conteúdo de cada arquivo. Após os arquivos terem sido armazenados na base de dados, a função retorna ao *front-end* a *hash* gerada e o identificador do arquivo na base de dados. Em seguida, os dados cruciais para a identificação do arquivo na base de dados, juntamente com a *hash* correspondente gerada previamente, são enviados para o *smart contract*. No *smart contract*, é realizada a validação da existência de um registo de utilizador, verificando se existe pelo menos um registo de evidência correspondente e se o utilizador é o responsável pelo registo. Se todas essas condições forem confirmadas, os dados relativos ao arquivo são adicionados ao *smart contract*. Caso contrário, a transação é revertida.

UC 02.1 *Triggers* de controlo da base de dados

Este caso de uso corresponde ao requisito funcional 18 da tabela de requisitos funcionais (Tabela 3.2). Como administrador do sistema, existe a possibilidade de analisar as alterações ocorridas na base de dados para garantir a integridade dos arquivos adicionados por utilizadores. Por meio de *triggers*, orientados a escutar a coleção da base de dados a ser utilizada no sistema, é necessário garantir que não existam alterações indesejadas e a quando existirem serem rapidamente detetadas.

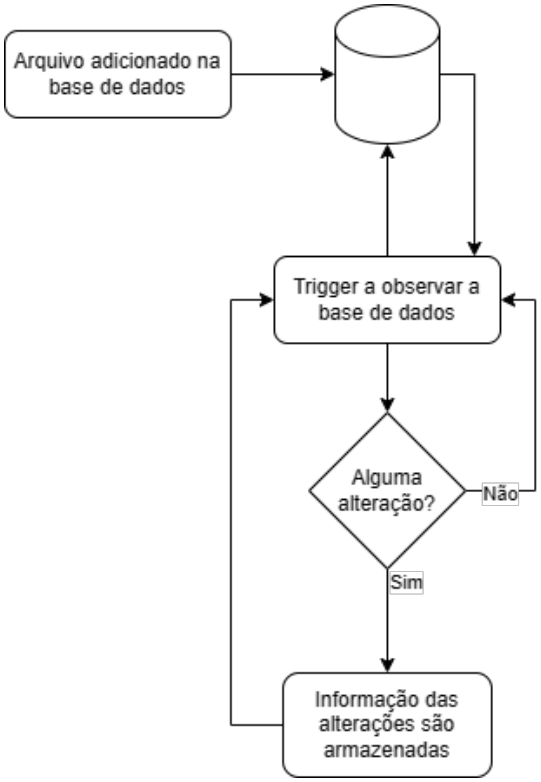


Figura 3.4: Fluxograma do *Trigger*

Na Figura 3.4, encontra-se representado o fluxograma correspondente ao caso de uso 02.1. Embora não seja visível para o utilizador comum, a importância deste caso de uso para um administrador do sistema é inquestionável. Para reforçar a integridade dos arquivos armazenados, é crucial manter o controlo sobre as áreas onde as falhas de segurança podem ser mais impactantes.

Para alcançar esse objetivo, utilizamos um *trigger* que monitoriza as alterações na base de dados do sistema. Esse *trigger* registra todos os eventos ocorridos, incluindo informações cruciais sobre as alterações. Isso significa que quando um arquivo é adicionado, o evento de adição é registado. Da mesma forma, se alguém tentar realizar uma alteração num arquivo previamente armazenado, seja de forma maliciosa ou não, o evento de alteração será registado. Essa funcionalidade também permite recuperar o arquivo originalmente armazenado, caso seja necessário.

UC 03 - Adicionar *"allowed users"*

Este caso de uso corresponde ao requisito funcional 10 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema e sendo o proprietário de um registo de evidência, é possivel adicionar utilizadores aos registos sob a sua responsabilidade, permitindo-lhes visualizar esses registos. Essa atualização é registada na *blockchain* com a identificação de que um utilizador foi adicionado aos utilizadores permitidos.

Na Figura 3.5, está representado o diagrama de atividades correspondente ao caso de uso 03. Da mesma forma que nos casos de uso anteriores, o utilizador deve estar conectado à aplicação através do *MetaMask*. O utilizador, por meio de um formulário, adiciona o(s) endereço(s) de utilizador(es) registados no sistema para que estes tenham permissão para visualizar um determinado registo. Após

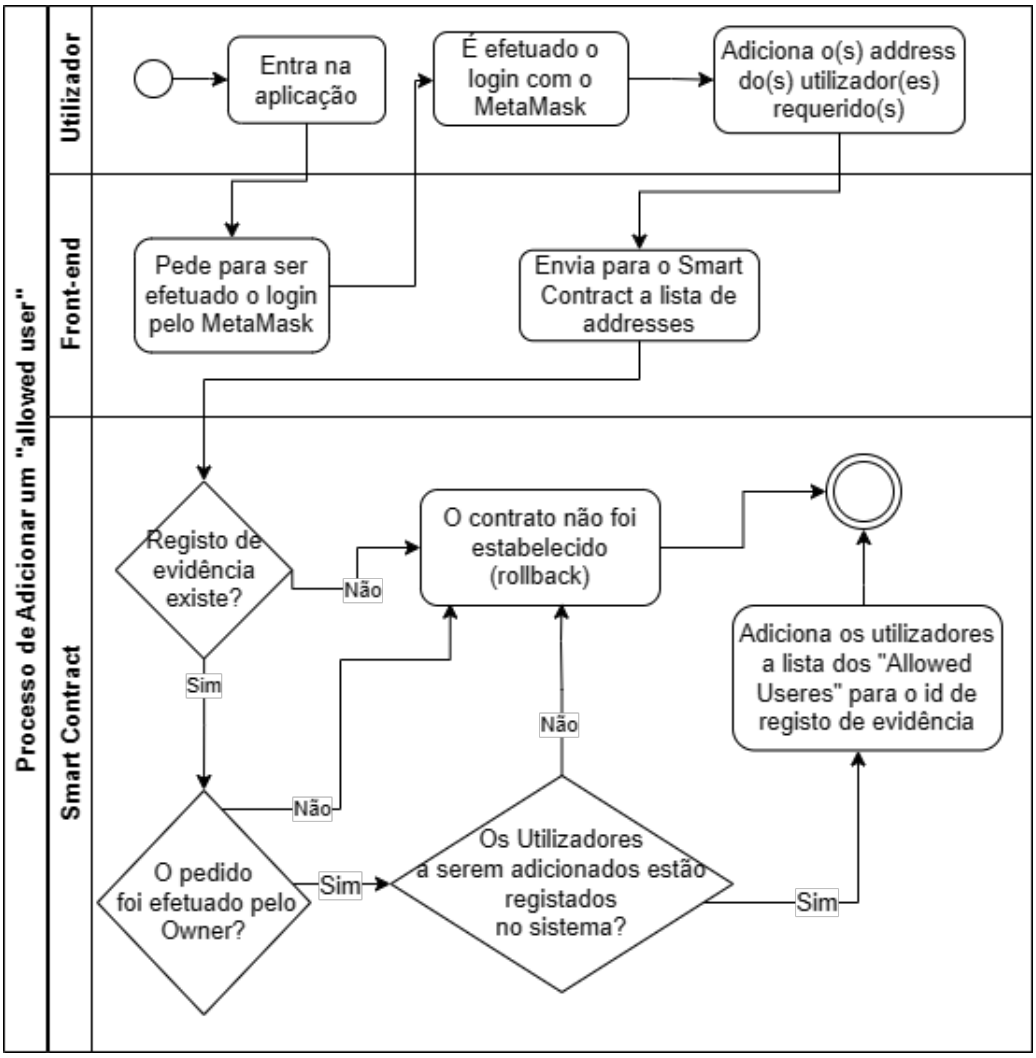


Figura 3.5: Diagrama de Atividades UC 03

inserir os endereços desejados e submetê-los ao *smart contract*, algumas validações são realizadas no *smart contract*, incluindo: o registo de evidência existe?; o pedido é efetuado pelo responsável pelo registo? Os utilizadores a serem adicionados à lista dos permitidos estão registados no sistema? Se todas as validações forem confirmadas, o registo dos endereços que foram adicionados como *"allowed users"* para um determinado registo de evidência é guardado no *smart contract*. Caso contrario, a transação é revertida.

UC 04 - Transferir o *ownership* para outro utilizador

Este caso de uso corresponde ao requisito funcional 11 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema e sendo o proprietário de um registo de evidência, existe a capacidade de transferir a responsabilidade (*ownership*) desse registo para outro utilizador do sistema. Isso significa que a análise que o proprietário precisava realizar e incluir no registo foi concluída, e ele pode "transferi" a evidência para outro utilizador (por exemplo, um agente da lei adicionou todos os dados sobre uma evidência e a transferiu para um membro da equipa de ciência forense para análise). Essa transferência de responsabilidade é registada na *blockchain* como parte do histórico do registo da evidência.

Na Figura 3.6, está representado o diagrama de atividades correspondente ao caso de uso 04. Da mesma forma que nos casos de uso anteriores, o utilizador deve estar conectado à aplicação através do *MetaMask*. O utilizador, por meio de um formulário, adiciona o endereço do utilizador ao qual pretende transferir a responsabilidade sobre um registo. Após isso, submete a solicitação para o *smart*

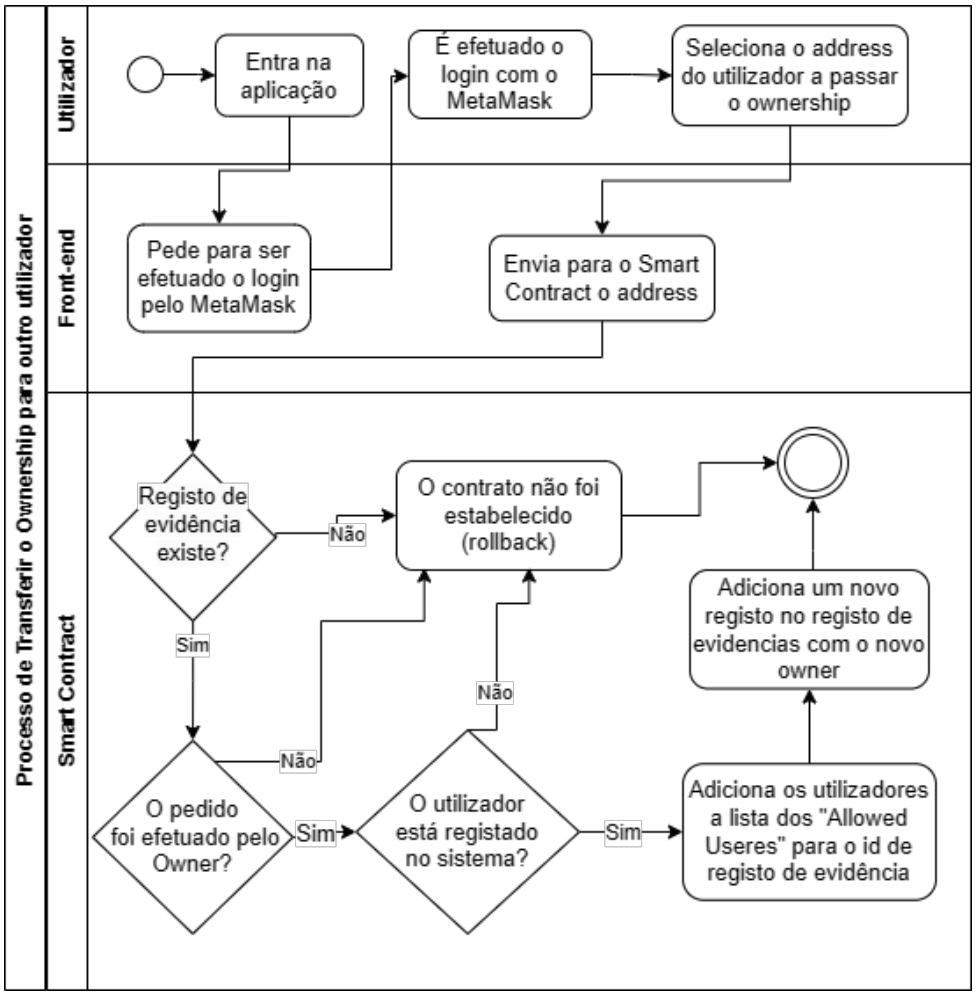


Figura 3.6: Diagrama de Atividades UC 04

contract. No *smart contract*, são realizadas validações para que a transação seja concluída com sucesso, tais como: o registo de evidência existe no sistema?; o utilizador que fez a solicitação de transferência de responsabilidade sobre um registo é o responsável atual pelo registo?; o novo responsável está registado no sistema?; Se todas as validações forem confirmadas, o utilizador é adicionado aos *"allowed users"* e também é definido como o novo responsável pelo registo de evidência.

UC 05 - Visualizar a lista de registos de evidências permitidas

Este caso de uso corresponde aos requisitos funcionais 02 e 03 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema, deve ser capaz de visualizar todos os registos de evidências públicas e aqueles aos quais fui incluído na lista dos utilizadores permitidos, apresentados em formato de tabela.

Na Figura 3.7, está representado o diagrama de atividades correspondente ao caso de uso 05. O utilizador deve estar conectado à aplicação através do *MetaMask*. Após isso, pode abrir a lista dos registos de evidências. Nesse momento, é enviado um pedido ao *smart contract* com o pedido de retorno dos registos existentes no sistema. No *smart contract*, são realizadas validações para determinar quais registos o utilizador tem acesso. Se o utilizador não existir, então o contrato não é estabelecido e a lista será vazia. Da mesma forma, se não existirem registos de evidências no sistema, o contrato não é estabelecido e a lista também será vazia. No entanto, se o utilizador existir e existirem registos de evidências, são verificados quais registos o utilizador tem permissão para visualizar e estes são retornados.

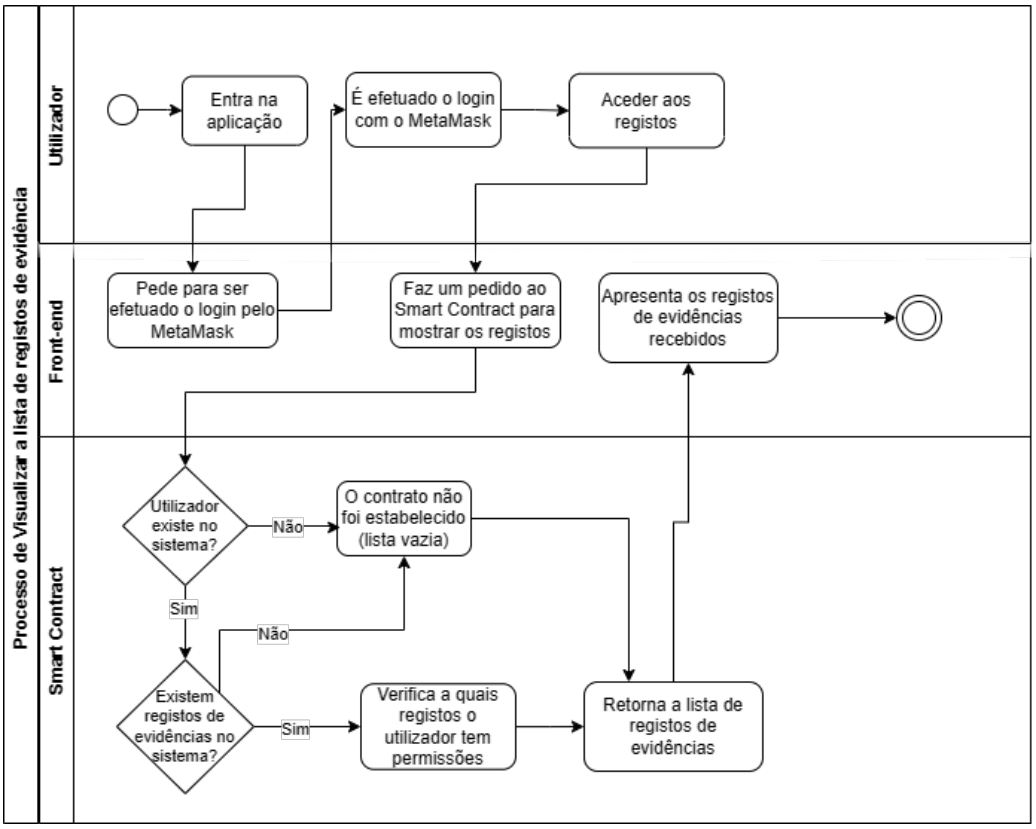


Figura 3.7: Diagrama de Atividades UC 05

UC 06 - Visualizar todo o histórico do registo de uma evidência

Este caso de uso corresponde aos requisitos funcionais 04 e 05 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema, deve ser capaz de visualizar o histórico de alterações de qualquer registo, seja ele público ou aquele ao qual fui incluído na lista dos utilizadores permitidos. Além disso, deve ser capaz de visualizar os responsáveis pelas alterações efetuadas nos registos.

Na Figura 3.8, está representado o diagrama de atividades correspondente ao caso de uso 06. A maior parte do processo é equivalente ao caso de uso 05. Após o utilizador visualizar a lista dos registos aos quais tem acesso, pode escolher um registo de evidencia e expandir o mesmo para visualizar todo o histórico de ocorrências relacionado ao mesmo.

UC 07 - Verificar a integridade dos arquivos

Este caso de uso corresponde aos requisitos funcionais 07 e 08 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema, deve ser capaz de visualizar se os arquivos incluídos no registo, quer sejam públicos ou aqueles aos quais fui incluído na lista dos utilizadores permitidos, são válidos ou se foram de alguma forma corrompidos. Além disso, deve ser capaz de efetuar o download desses arquivos, se desejado.

Na Figura 3.9, está representado o diagrama de atividades correspondente ao caso de uso 07. A maior parte do processo é equivalente ao caso de uso 06. Após o utilizador visualizar o histórico de um determinado registo, se esse registo contiver algum arquivo digital, é automaticamente solicitada a verificação da integridade desse arquivo. Recebidos os dados sobre o arquivo e a *hash* que foi gerada no momento do *upload*, esses dados são enviados para o *back-end* para que o arquivo seja verificado. No *back-end*, o arquivo é extraído da base de dados e é gerada uma nova *hash*. Após a *hash* ter sido gerada, é efetuada uma comparação com a *hash* original; se as *hashes* forem iguais, então o documento está integralmente intacto.

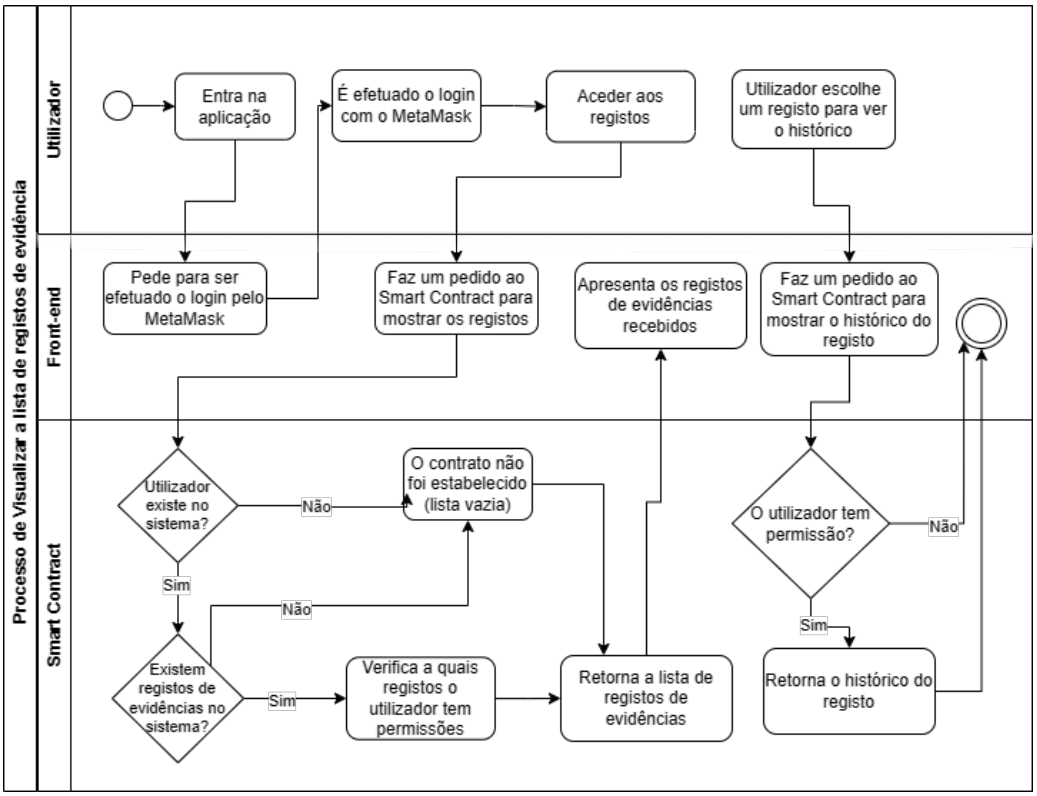


Figura 3.8: Diagrama de Atividades UC 06

UC 08 - Download dos arquivos anexados

Este caso de uso corresponde ao requisito funcional 08 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador autenticado no sistema, deve ser capaz de efetuar o download dos arquivos incluídos num registo de evidência ao qual tenho acesso.

Na Figura 3.10, está representado o diagrama de atividades correspondente ao caso de uso 08. Após o processo de verificação dos arquivos na listagem do histórico de uma determinada evidência, o utilizador deve ter a possibilidade de transferir os arquivos armazenados. Para realizar essa ação, o utilizador deve estar autenticado via *MetaMask* e também fazer *login* através do seu nome de utilizador e palavra-passe. Após a autenticação, o utilizador pode transferir o arquivo desejado. No *front-end*, é validado se o utilizador está autenticado e, em seguida, é enviado um pedido para efetuar o *GET* do ficheiro ao *back-end*. Se o ficheiro for válido, ele é transferido para o computador do utilizador.

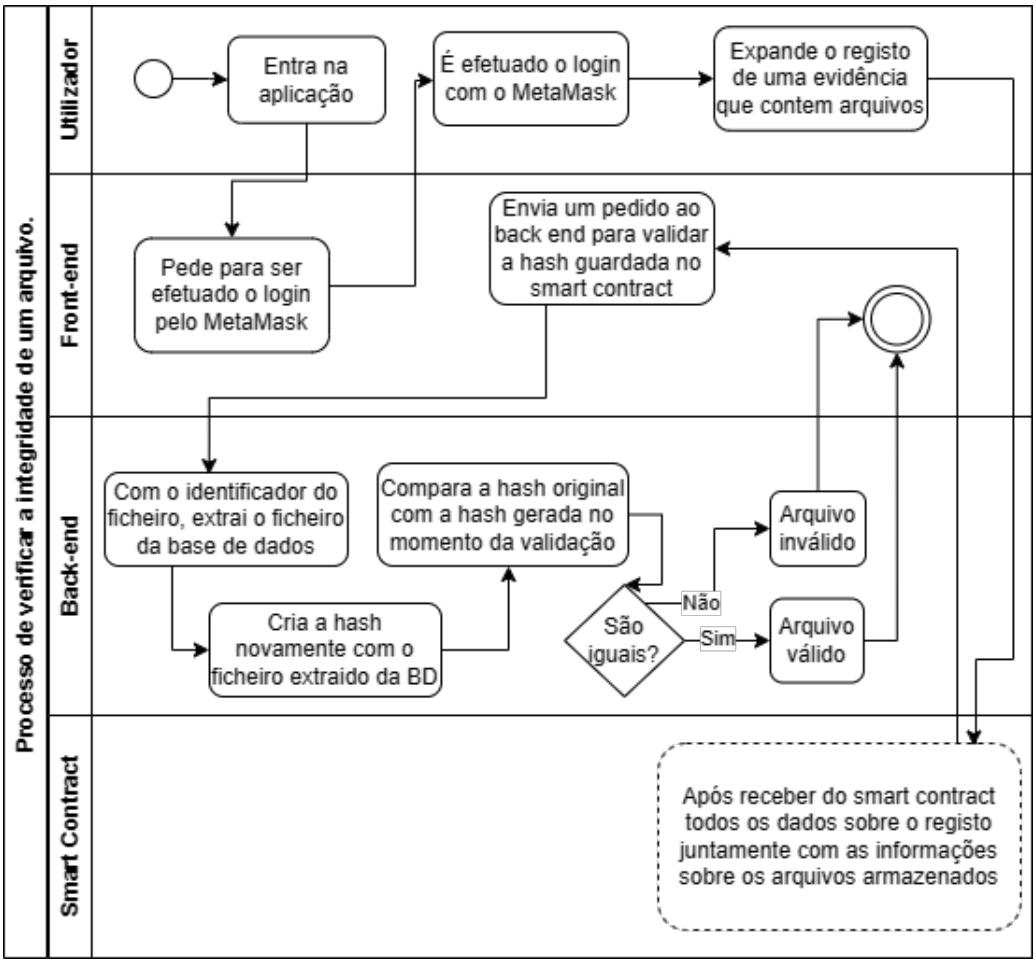


Figura 3.9: Diagrama de Atividades UC 07

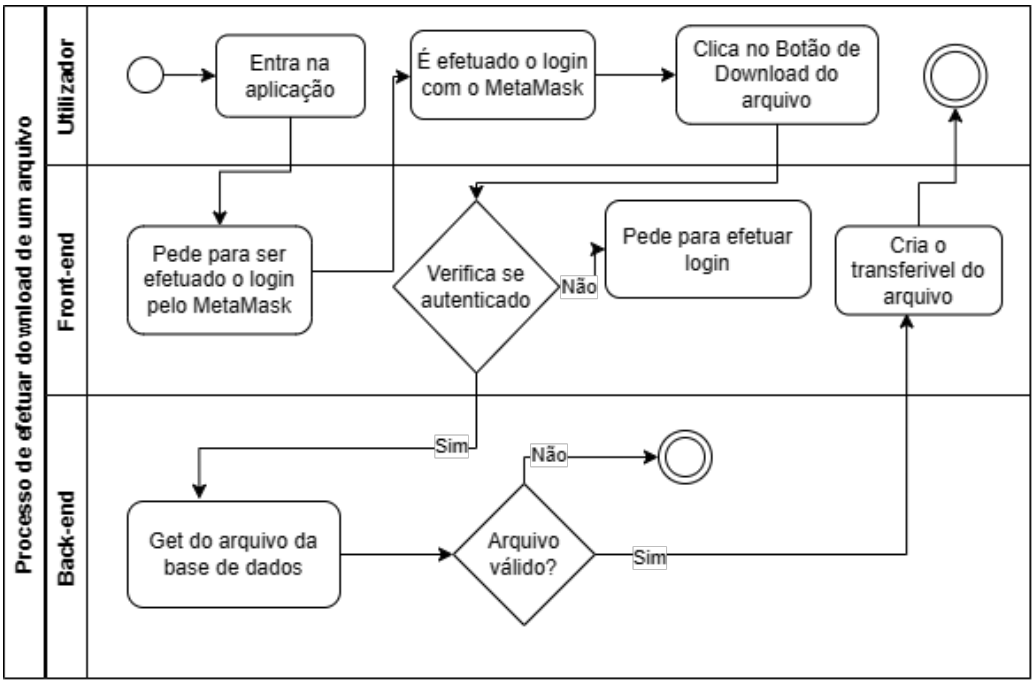


Figura 3.10: Diagrama de Atividades UC 08

3.3.3 Diagrama de casos de uso de entidades

Na Figura 3.11 estão representados os casos de uso correspondentes ao processo de gestão dos entidades.

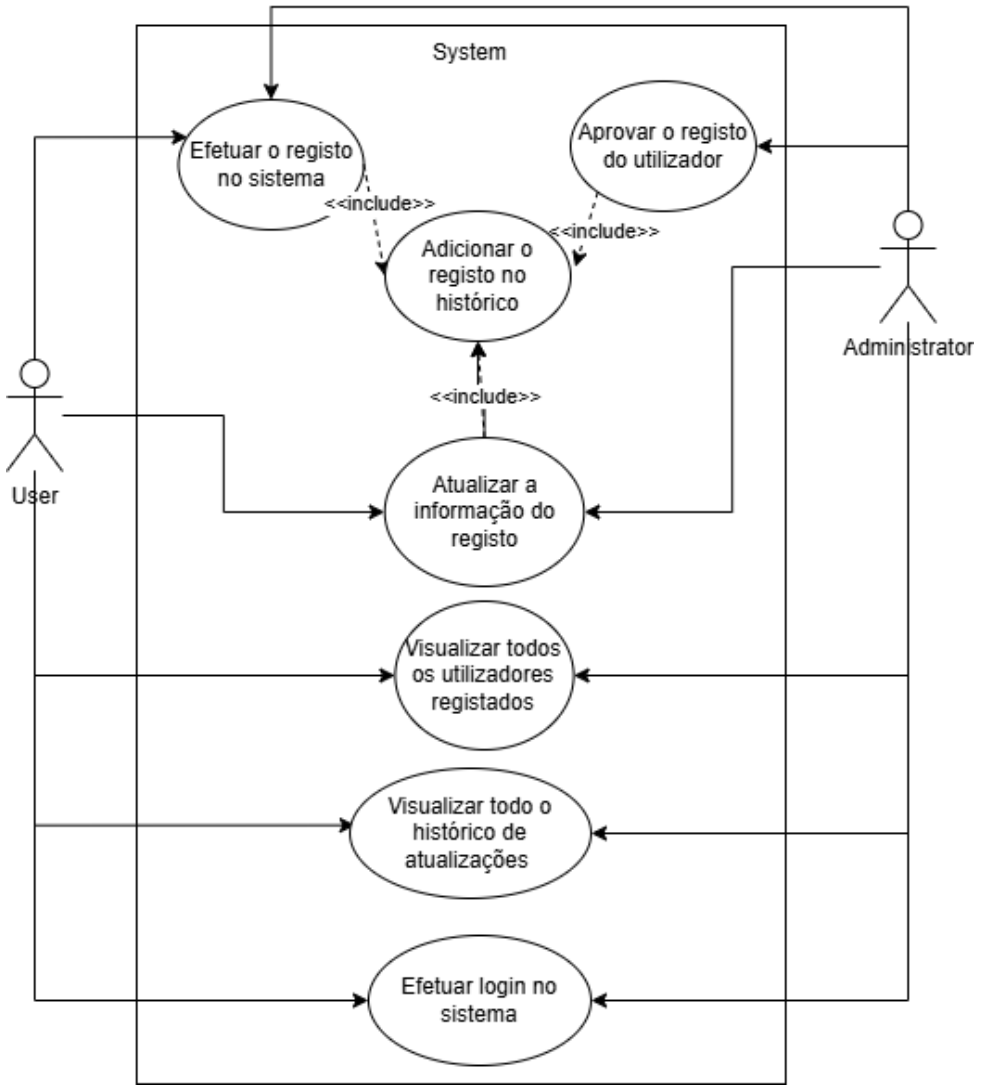


Figura 3.11: Diagrama de casos de uso (Entidades)

UC 09 - Efetuar o registo do *user* no sistema

Este caso de uso corresponde ao requisito funcional 12 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador, devo ter a capacidade de me registar no sistema. Esse registo é armazenado na blockchain e marcado como um registo pendente de aprovação. Automaticamente, esse registo é armazenado como o primeiro registo no histórico de registos do utilizador em questão.

Na Figura 3.12, está representado o diagrama de atividades correspondente ao caso de uso 09. O utilizador deve estar autenticado com a aplicação *MetaMask* para conseguir efetuar o registo no sistema. Em seguida, o utilizador preenche o formulário de registo de utilizador, e quando o formulário for válido, pode prosseguir com a solicitação de registo. Os dados do formulário são enviados para o *smart contract*. No *smart contract*, é verificado se já existe um utilizador registado com os mesmos dados e, se isso acontecer, a transação é cancelada devido à existência de um utilizador duplicado. Caso não exista um utilizador com os mesmos dados, o registo é armazenado na *blockchain* e marcado como pendente para aprovação.

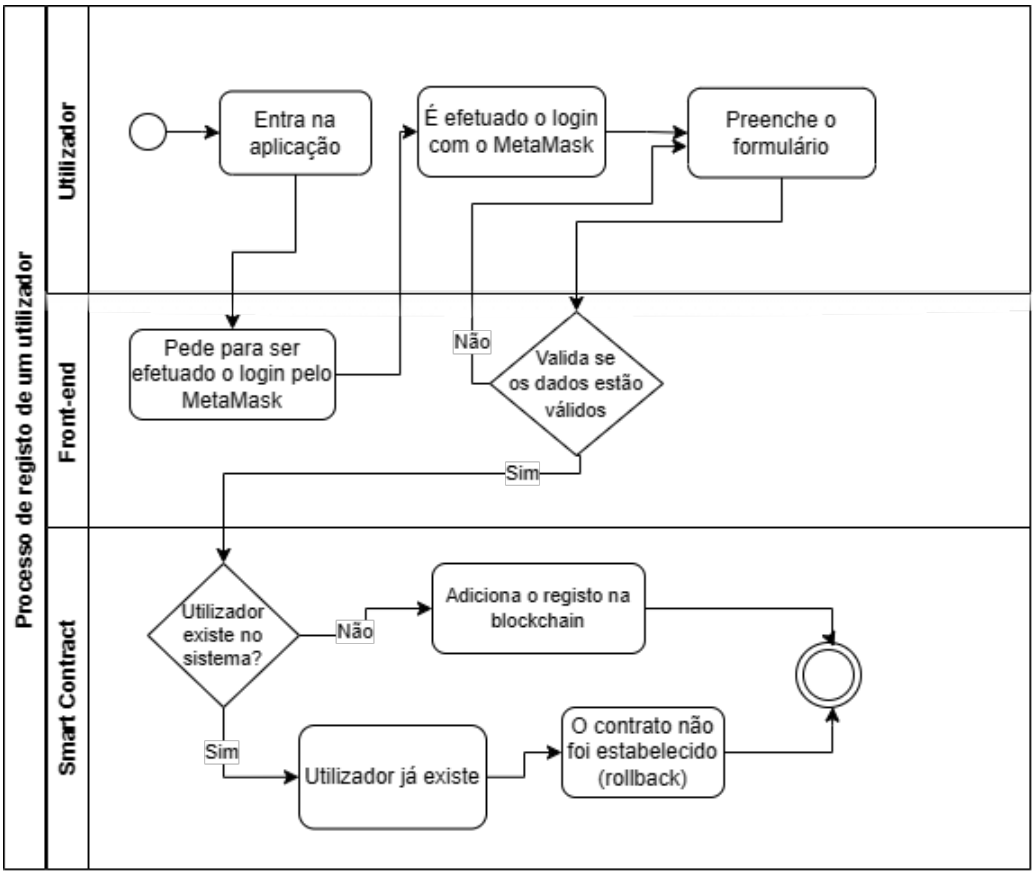


Figura 3.12: Diagrama de Atividades UC 09

UC 10 - Aprovar o registo do utilizador

Este caso de uso corresponde ao requisito funcional 13 da tabela de requisitos funcionais (Tabela 3.2). Como administrador, devo ter a capacidade de aprovar os registos que foram submetidos por utilizadores no sistema. Essa aprovação é registada na blockchain, juntamente com o identificador do responsável pela aprovação.

Na Figura 3.13, está representado o diagrama de atividades correspondente ao caso de uso 10. O administrador deve estar autenticado com a aplicação *MetaMask* para conseguir efetuar aprovações. Após selecionar um utilizador pendente de aprovação da lista de utilizadores, o administrador clica em aprovar o utilizador. Nesse momento, é enviada uma transação para o *smart contract*. No *smart contract*, existem algumas validações que devem ser respeitadas antes do contrato ser concluído com sucesso. As validações incluem: o aprovador é um administrador? Caso essa validação não seja confirmada, a transação é revertida. O utilizador que está sendo aprovado é um utilizador existente no sistema? Caso contrário, a transação é revertida. O utilizador está pendente de aprovação? Caso contrário, a transação também é revertida. Se todas as validações forem confirmadas, um registo é adicionado ao histórico do utilizador, indicando a aprovação e o responsável pela mesma.

UC 11 - Atualizar a informação do registo

Este caso de uso corresponde ao requisito funcional 15 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador, devo ter a capacidade de editar os dados do meu registo. O utilizador pode proceder à edição do seu registo, e essa edição é registada na blockchain e marcada como um registo pendente de aprovação.

Na Figura 3.14, está representado o diagrama de atividades correspondente ao caso de uso 11. O utilizador deve estar autenticado com a aplicação *MetaMask* para conseguir efetuar a alteração do registo no sistema. Em seguida, o utilizador preenche o formulário para editar o seu registo, e quando o formulário for válido, pode prosseguir com a solicitação da edição do registo. Os dados do formulário são enviados

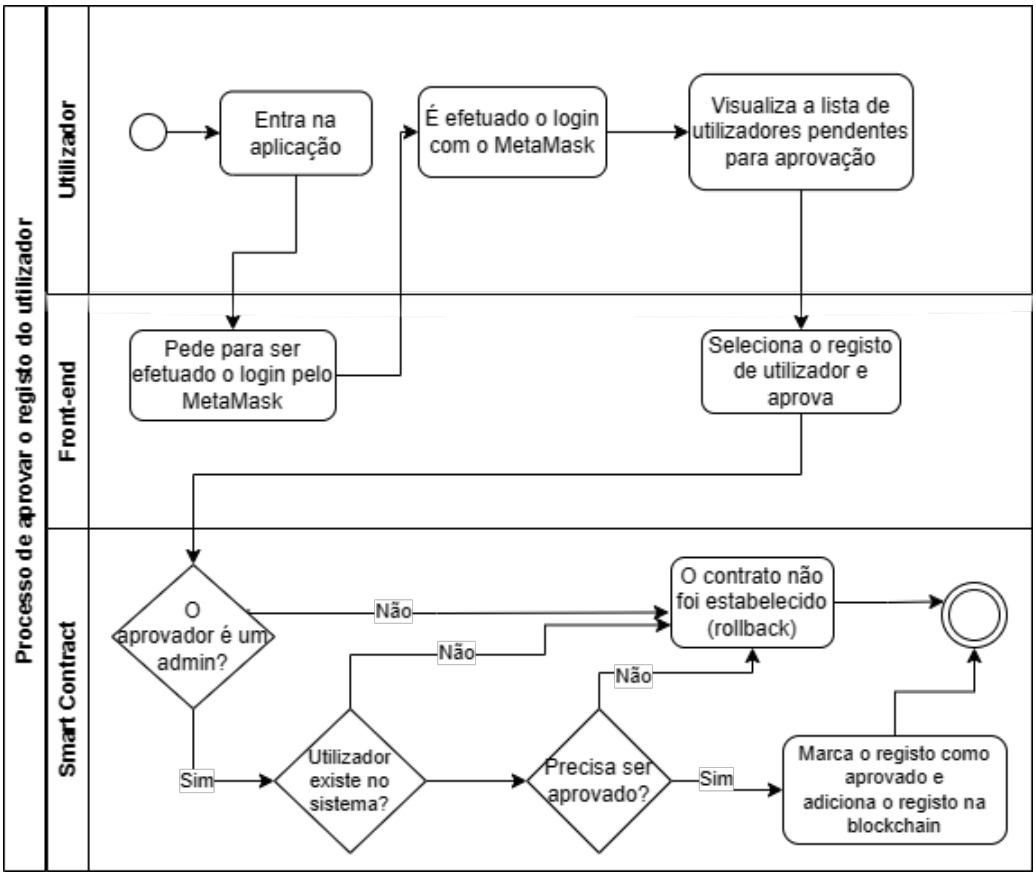


Figura 3.13: Diagrama de Atividades UC 10

para o *smart contract*. No *smart contract*, é verificado se o utilizador existe e está registado, e são validados os dados de autenticação do mesmo. Caso as validações não se verifiquem, a transação é cancelada. Se todas as validações forem concluídas com sucesso, o registo é armazenado na *blockchain* no histórico do registo de utilizador e marcado como pendente para aprovação. É importante notar que os registos anteriores do utilizador não são editados, mas sim acrescentados novos.

UC 12 - Visualizar todos os utilizadores registados

Este caso de uso corresponde ao requisito funcional 14 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador, devo ter a capacidade de visualizar todos os utilizadores registados no sistema.

Na Figura 3.15, está representado o diagrama de atividades correspondente ao caso de uso 12. O utilizador deve estar autenticado com a aplicação *MetaMask*. Em seguida, o utilizador seleciona para visualizar a lista de utilizadores existentes no sistema. Nesse momento, é efetuado um pedido ao *smart contract* para trazer os registos. No *smart contract*, são realizadas algumas verificações. Primeiro, é verificado se existem registos de utilizadores. Se não existirem, o contrato não foi estabelecido e a lista será vazia. Se existirem registos, é verificado se o utilizador que fez a solicitação é um administrador do sistema. Se o utilizador for um administrador, todos os registos de utilizadores, tanto pendentes quanto aprovados, são retornados. Se o utilizador não for um administrador, apenas os registos de utilizadores aprovados são retornados.

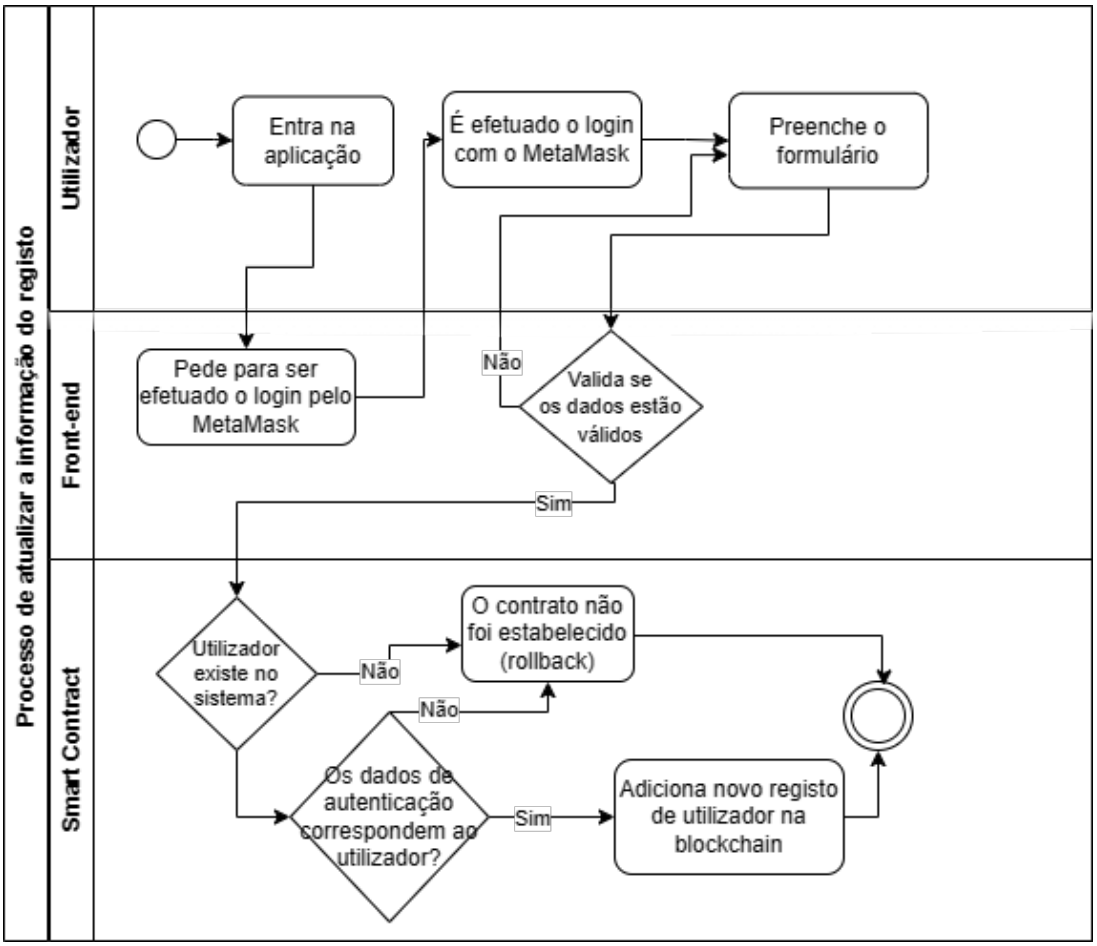


Figura 3.14: Diagrama de Atividades UC 11

UC 13 - Visualizar todo o histórico de atualizações

Este caso de uso corresponde ao requisito funcional 16 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador, devo ter a capacidade de visualizar todo o histórico de atualizações ocorridas sob um registo de utilizador.

Na Figura 3.16, está representado o diagrama de atividades correspondente ao caso de uso 13. O utilizador deve estar autenticado com a aplicação *MetaMask*. O utilizador abre a lista de utilizadores existentes no sistema e, nesse momento, ocorre o processo do caso de uso 13. Em seguida, o utilizador seleciona um registo de utilizador para expandir. Nesse momento, é efetuado um pedido ao *smart contract* para trazer os registos de histórico do utilizador. No *smart contract*, é verificado se o registo de utilizador existe. Se existir, é retornado todo o histórico de alterações do registo de utilizador selecionado. Caso não exista nenhum registo, o contrato não foi estabelecido e a lista do histórico será vazia.

UC 14 - Efetuar *login* no sistema

Este caso de uso corresponde ao requisito funcional 17 da tabela de requisitos funcionais (Tabela 3.2). Como utilizador, estando autenticado no sistema, devo ter a capacidade de efetuar o *login* no sistema utilizando o meu *username* e *password*.

Na Figura 3.17, está representado o diagrama de atividades correspondente ao caso de uso 14. O utilizador deve estar autenticado com a aplicação *MetaMask*. O utilizador preenche o formulário de login e clica em "Efetuar Login". O pedido é enviado para o *back-end*, onde é inicialmente validado se o *payload* contém todos os valores necessários. Se o *payload* for válido, é enviado um pedido para o *smart contract* para efetuar a autenticação do utilizador. No *smart contract*, é verificado se o endereço enviado corresponde a algum utilizador registado. Se não corresponder, o contrato não foi estabelecido e

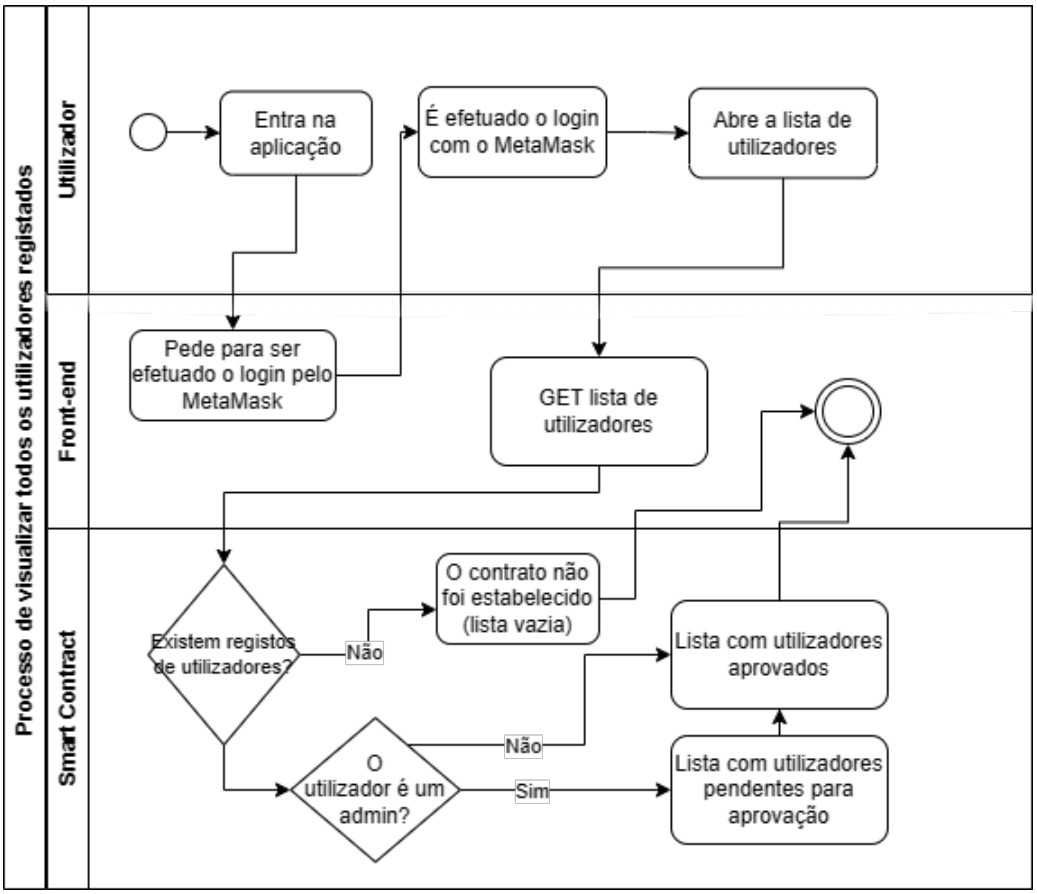


Figura 3.15: Diagrama de Atividades UC 12

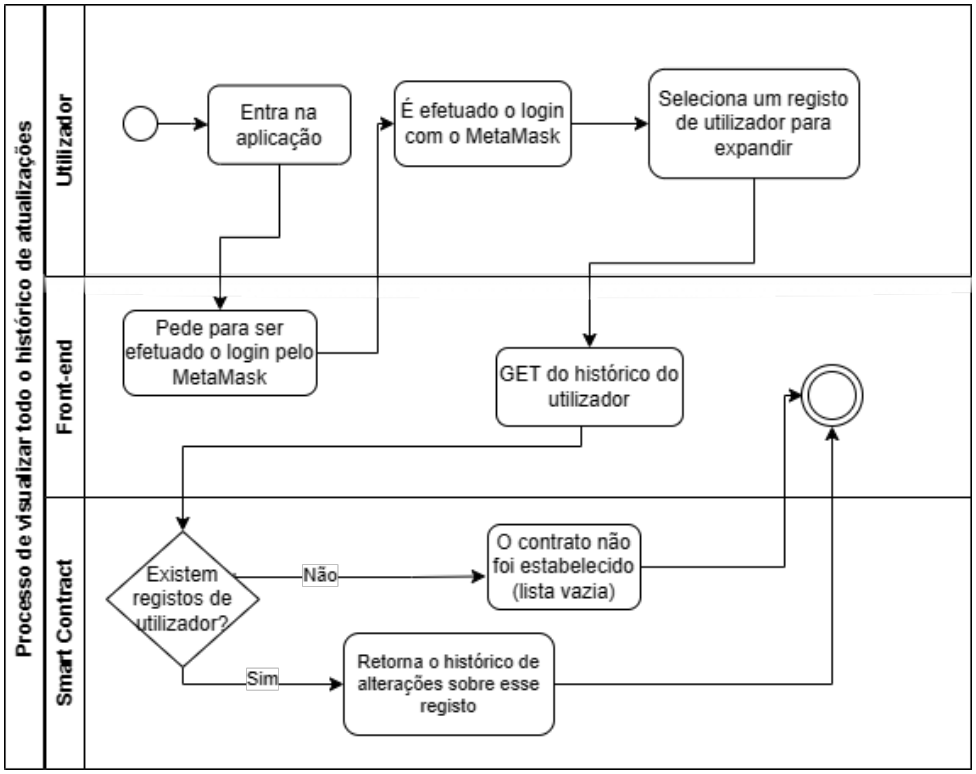


Figura 3.16: Diagrama de Atividades UC 13

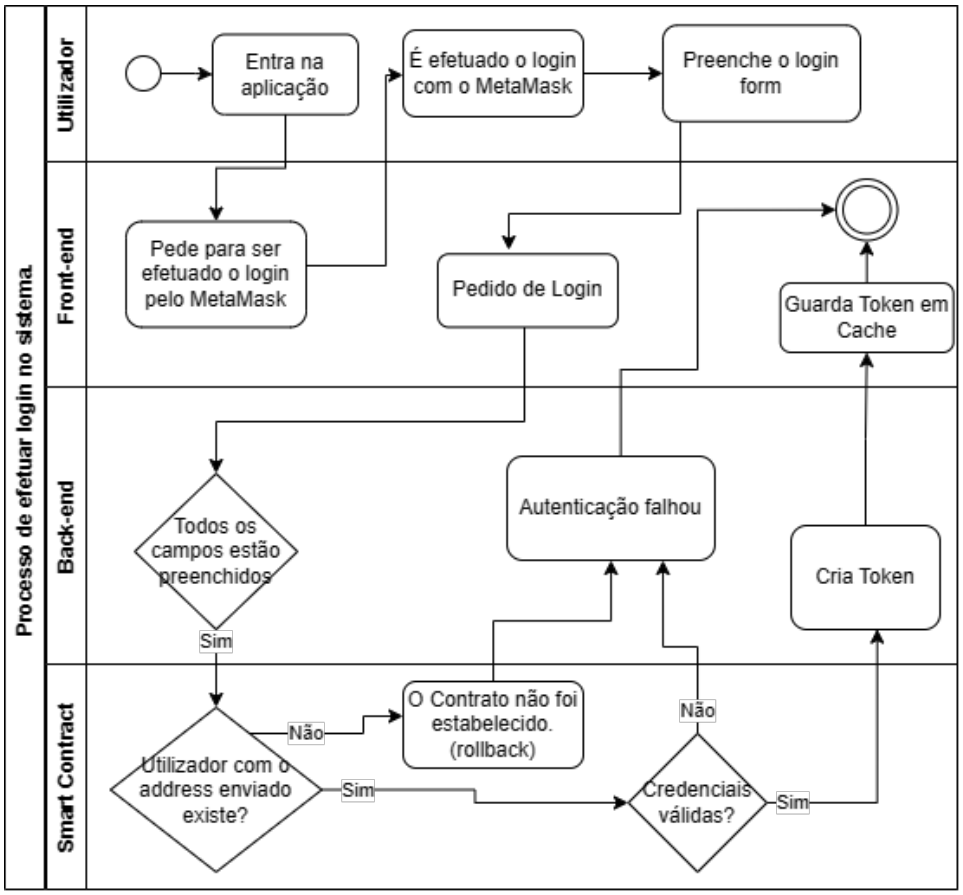


Figura 3.17: Diagrama de Atividades UC 14

a transação é anulada, o que significa que a autenticação falhou. Se existir um registo de utilizador, são verificadas as credenciais. Se corresponderem, o utilizador é autenticado com sucesso e é retornado para o *back-end* o o registo do utilizador autenticado. No *back-end*, é criado um *token* para esse utilizador e a resposta é enviada de volta para o *front-end*. No *front-end*, o *token* é armazenado em *cache* para manter o utilizador autenticado.

3.4 Arquitetura Funcional

Até este ponto, foram realizadas análises detalhadas do sistema em várias etapas. Inicialmente, foi feito o levantamento de requisitos (Secção 3.2). Em seguida, foram apresentadas as funcionalidades a implementar, por meio de diagramas de casos de uso e diagramas de atividades para ilustrar os requisitos identificados (Secção 3.3).

Nesta secção, será apresentada a arquitetura do sistema num nível elevado de abstração, permitindo a visualização dos diferentes componentes e sua interação. Além disso, discutiremos a versão inicial da arquitetura que foi apresentada num póster desenvolvido para um workshop realizado na Escola Superior de Tecnologia do IPCA, com o objetivo de introduzir o tema da presente dissertação. O póster em questão pode ser consultado nos anexos deste trabalho.

Na Figura 3.18 está representada, a ideia geral da proposta desta dissertação. Nesta abordagem inicial, não foi considerada uma interface *WEB* para uma interação mais simples com o utilizador. Além disso, podemos observar que a *API* atua como uma ponte entre os utilizadores e o *smart contract*. Embora não seja a arquitetura final, este diagrama permite-nos entender um pouco mais sobre a ideia de rastreabilidade das evidências.

Na Figura 3.19, está representada a versão atual da arquitetura da solução. Nesta versão, é possível visualizar as ligações efetuadas pelo *front-end* e pelo *back-end*. O utilizador (*User*) só tem conexão com

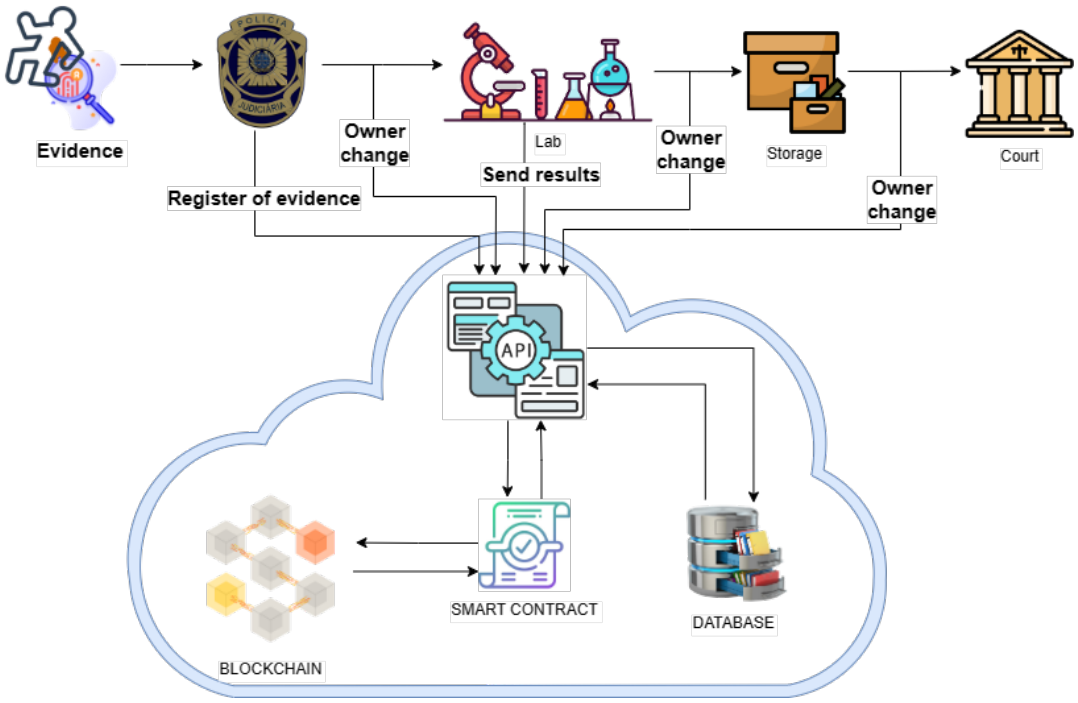


Figura 3.18: Arquitetura Funcional (V.1)

o *front-end*, e não existe outra forma de interação com o sistema. O *front-end* tem ligações diretas com o *smart contract* e com a *API (back-end)*. Essa conclusão pode ser alcançada com base nos diagramas de atividades apresentados na secção 3.3. O *back-end* só pode consultar informações no *smart contract*, como, por exemplo, a autenticação do utilizador. A base de dados só é acedida através da *API*, e a *Blockchain* é acedida apenas pelo *smart contract*.

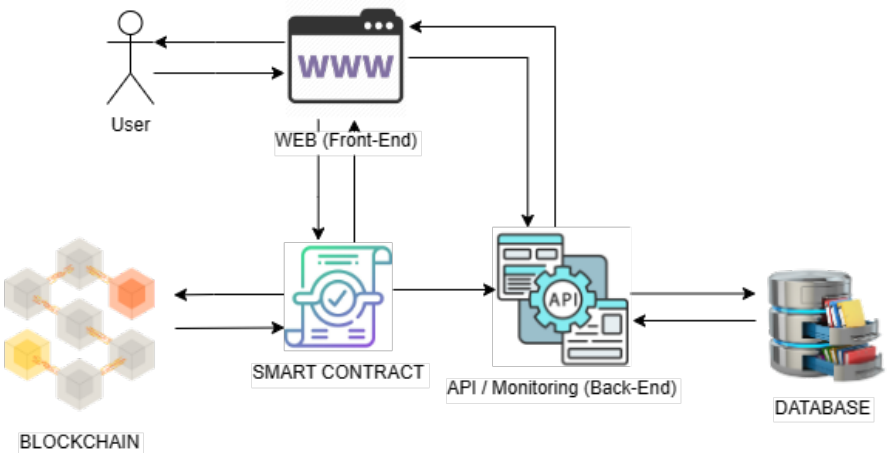


Figura 3.19: Arquitetura Funcional (V.2)

3.4.1 Arquitetura Tecnológica

Nesta secção, será explicada a arquitetura tecnológica da solução. A arquitetura tecnológica visa representar as tecnologias a serem utilizadas em cada parte do projeto. Será apresentada uma descrição breve de cada tecnologia presente na arquitetura tecnológica do projeto.

A Figura 3.20 representa a arquitetura tecnológica da solução.

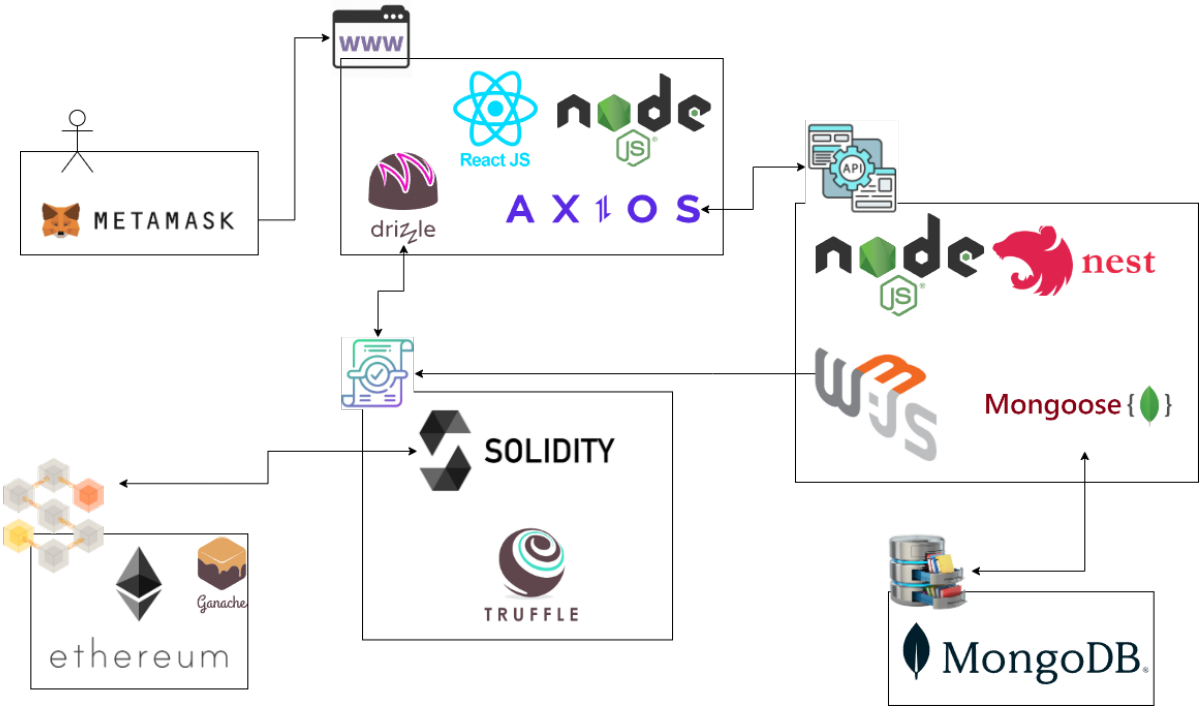


Figura 3.20: Arquitetura tecnológica

MetaMask

MetaMask é uma carteira de criptomoedas e extensão de navegador que permite aos utilizadores gerir e interagir com aplicações descentralizados *dApps* baseados na *Ethereum*. Ele serve como uma carteira digital para armazenar, enviar e receber *Ether (ETH)*, a criptomoeda nativa da rede *Ethereum*. A extensão está disponível para navegadores populares como Google Chrome, Firefox, Edge, etc. MetaMask oferece segurança na geração e gestão de chaves privadas e suporta diferentes redes *Ethereum*, incluindo *testnets* para fins de desenvolvimento (MetaMask, 2023).

Nesta arquitetura, o *MetaMask* desempenha um papel fundamental na interação com a solução. Através da extensão do *MetaMask*, os utilizadores têm um maior controlo sobre as transações, incluindo a capacidade de aprovar ou recusar. Além disso, o *MetaMask* é utilizado para realizar o *deployment* do *smart contract* em uma *testnet*.

Node.js

Node.js é um ambiente de execução de código *JavaScript* de código aberto e multiplataforma, bem como uma biblioteca para executar aplicações da web fora do navegador do cliente. Foi desenvolvido por Ryan Dahl em 2009. Desenvolvedores utilizam o *Node.js* para criar aplicações web no lado do servidor, sendo particularmente adequado para aplicações intensivas em dados, uma vez que utiliza um modelo assíncrono e orientado a eventos. *Node.js* é uma escolha popular para desenvolvimento *web* e *back-end* devido à sua escalabilidade, desempenho e facilidade de uso (Sufiyan, 2023). Neste projeto, o *Node.js* foi utilizado de uma forma implícita com a utilização do *ReactJS* e *NestJS*.

ReactJS

O *ReactJS* é uma biblioteca *JavaScript* de código aberto amplamente utilizada para o desenvolvimento de interfaces de utilizador. Uma das principais vantagens do *ReactJS* é o *JSX*, uma extensão do *JavaScript* que permite escrever elementos de interface de utilizador em uma sintaxe semelhante ao *HTML* diretamente no código *JavaScript*, tornando a criação de componentes mais fácil e legível. Além disso, o *ReactJS* é conhecido por sua reatividade, o que significa que quando os dados são alterados, a interface do utilizador é atualizada automaticamente (React, 2023).

Essa capacidade de atualização automática é particularmente importante ao lidar com a *blockchain*, pois as transações e execuções podem levar algum tempo para serem concluídas. Portanto, quando uma transação é finalizada, o *React*, juntamente com a biblioteca *Drizzle*, ajuda a manter a interface do utilizador atualizada com as informações mais recentes. Isso garante uma experiência de utilizador mais suave e eficiente.

Foi utilizado o *ReactJS* no desenvolvimento da parte *front-end* do sistema.

Drizzle

Conforme referido na secção 2.5.4, a biblioteca *Drizzle* faz parte do conjunto de ferramentas fornecido pelo *Truffle*. O *Drizzle* será utilizado em conjunto com o *ReactJS* para conectar o *smart contract* à interface *WEB*. Assim permitindo que as informações mais atualizadas sejam exibidas ao utilizador e proporciona uma experiência de utilização mais fluida.

Neste projeto o *Drizzle* foi utilizado para garantir a sincronização eficiente entre a *blockchain* e a interface, tornando a interação com o sistema mais eficaz e intuitiva.

Axios

O *Axios* é uma biblioteca *JavaScript promise-based* que simplifica a interação com serviços de *API*, oferecendo suporte para várias solicitações *HTTP*, como *GET, POST, PUT e DELETE*. Ele pode ser usado em diversas *frameworks JavaScript* e até mesmo em código *JavaScript* puro. O *Axios* economiza tempo, funciona bem com dados *JSON* por padrão (não requer configuração de cabeçalhos), possui nomes de funções compatíveis com os métodos *HTTP*, requer menos código para obter dados *JSON* e lida melhor com erros, fornecendo códigos de erro detalhados. É uma escolha versátil tanto para o cliente quanto para o servidor, sendo útil em desenvolvimento *Node.js* (Gospel, 2023).

O *Axios* será utilizado para interligar o *front-end* com os *endpoint* criados no *back-end*.

NestJS

O *Nest* (ou *NestJS*) é uma *framework* para construir aplicações no lado do servidor em *Node.js* que combina *JavaScript* progressivo com suporte total ao *TypeScript*. Ele une conceitos de programação orientada a objetos (OOP), programação funcional (FP) e programação reativa funcional (FRP). O *Nest* utiliza *frameworks HTTP* robustos, como o *Express* (padrão) ou o *Fastify* (opcional), proporcionando uma camada de abstração sobre eles, mas permitindo que os desenvolvedores acessem as suas *APIs* diretamente. Isso oferece flexibilidade para usar módulos de terceiros e o torna uma escolha versátil para a construção de aplicações escaláveis *server-side* em *Node.js* (NestJS, 2023).

Será utilizado o *NestJS* para o desenvolvimento do *back-end* do sistema.

Web3.js

O *Web3.js* é uma biblioteca *JavaScript* que oferece uma *API* para interagir com a *blockchain Ethereum*. Com ele, os desenvolvedores podem criar aplicações que se conectam a *smart contracts*, consultam informações da *blockchain*, enviam transações e rastreiam eventos. É amplamente utilizado em aplicações descentralizadas (dApps) e simplifica o desenvolvimento de soluções *blockchain* ao abstrair as complexidades da interação com a *Ethereum* (McCubbin, 2023).

O *Web3.js* será utilizado explicitamente do lado do servidor para executar pedidos ao *smart contract*, também é será utilizado o *Web3.js* do lado do *front-end* mas de uma forma implicita com o *Drizzle*.

MongoDB e Mongoose

O *MongoDB* é um banco de dados de código aberto orientado a documentos projetado para armazenar grandes volumes de dados de forma eficiente. Ele se enquadra na categoria *NoSQL* devido à sua estrutura de armazenamento não baseada em tabelas. Foi lançado em 2009 e é gerido pela *MongoDB, Inc.* O *MongoDB* oferece suporte oficial para drivers em várias linguagens populares, o que permite que os desenvolvedores criem aplicações em diversas plataformas (Saini, 2021).

Não seria viável armazenar na *blockchain* os arquivos digitais adicionados pelos utilizadores devido ao seu tamanho. Portanto, para o armazenamento de arquivos digitais, será utilizada a base de dados *MongoDB*. Para interligar a base de dados com o *back-end* será utilizada uma biblioteca *JavaScript* integrada no *NestJS*, o *Mongoose*.

Solidity

Na secção 2.5.1, foi realizada uma comparação entre as duas principais linguagens de programação para *smart contracts* na *blockchain Ethereum*. Após uma análise detalhada, foi decidido utilizar a linguagem *Solidity* para desenvolver os *smart contracts* da presente solução. A escolha do *Solidity* deve-se à sua ampla adoção na comunidade *Ethereum*, sua maturidade e robustez, bem como à disponibilidade de recursos e ferramentas de desenvolvimento que facilitam a criação de *contratos inteligentes* seguros e eficientes, como por exemplo *Truffle Framework*.

Truffle

Conforme referido na secção 2.5.2, o *Truffle* é uma *framework* poderosa para a construção, teste e implantação de *smart contracts* na plataforma *Ethereum*.

Será utilizado neste projeto para esse efeito, compilar os *smart contracts*, fazer *deploy* na *blockchain* local e na *testnet* e criar testes.

Ganache

Conforme referido na secção 2.5.3, o *Ganache* é uma *blockchain* privada local.

Foi usada neste projeto para o desenvolvimento e testes dos *smart contracts*.

3.5 Modelo de Domínio

A modelação de domínio é crucial na engenharia de software, mesmo em pequena escala, pois simplifica o desenvolvimento de sistemas, esclarece requisitos e reflete o entendimento atual das entidades e relacionamentos no domínio do problema (Agile, 2023). No âmbito da representação do modelo de domínio, foi criado o seguinte diagrama de classes (Fig. 3.21).

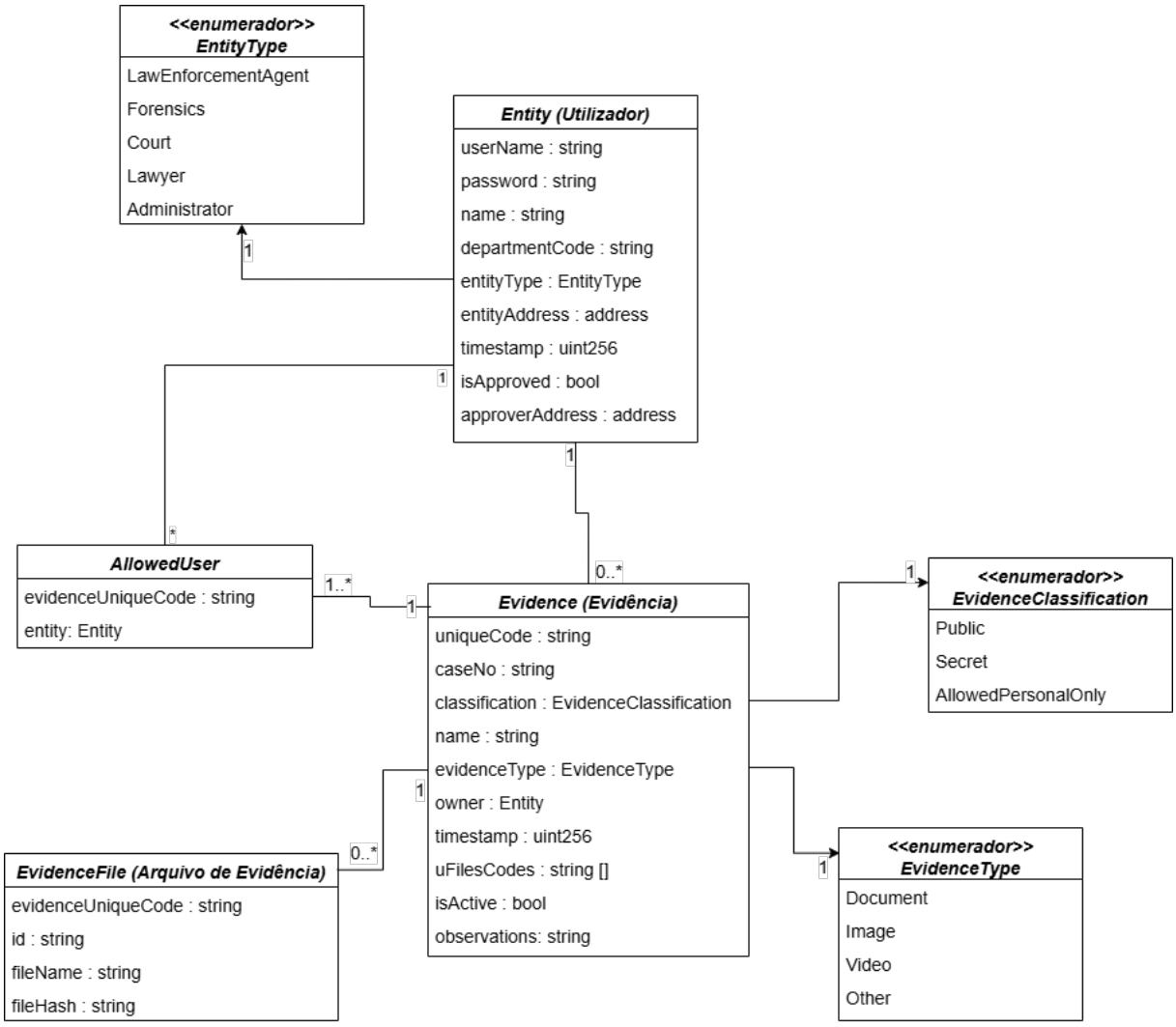


Figura 3.21: Diagrama de Classes

A classe ***Evidence*** é considerada a classe principal, e será a representação digital de uma evidencia física/real. Outras classes serão também a representação digital de entidades físicas como por exemplo ***Entity***, que será a representação de utilizador, ***EvidenceFile***, que será representação de um arquivo e ***AllowedUser*** que será representação da permissão perante um registo. De seguida será elaborada uma descrição para cada uma das classes da Figura 3.21:

- ***Evidence -*** Classe que simboliza um registo de uma evidência. Aqui, serão guardados os seguintes campos:
  - ***uniqueCode : string***: Código único da evidência definido pelo utilizador.
  - ***caseNo : string***: Número do caso jurídico ao qual a evidência pertence.
  - ***classification : EvidenceClassification***: Enumerado para classificar a evidência. Foram adicionados 3 tipos, mas posteriormente poderão ser alterados e/ou acrescentados outros.
  - ***name : string***: Breve descritivo da evidência.
  - ***evideceType : EvidenceType***: Enumerado para classificar o tipo de evidência. Foram adicionados 4 tipos, mas posteriormente poderão ser alterados e/ou acrescentados outros.
  - ***owner : Entity***: Responsável pelo registo da evidência.
  - ***timestamp : uint256***: Momento exato da criação daquele registo.
  - ***uFilesCodes : string[]***: Identificadores únicos dos arquivos adicionados naquele registo de evidência em específico.
  - ***isActive : bool***: Indica se a evidência está em estado ativo ou não.
  - ***observations : string***: Campo de texto para adicionar comentários sobre o registo.
- ***Entity -*** Classe que representa uma entidade jurídica. Aqui, serão guardados os seguintes campos:
  - ***userName : string***: Nome de utilizador escolhido pelo utilizador. Usado para efetuar a autorização.
  - ***password : string***: *Password* de utilizador escolhido pelo utilizador. Usado para efetuar a autorização.
  - ***name : string***: Nome do utilizador.
  - ***departmentCode : string***: Codigo do departamento ao qual pertence.
  - ***entityType : EntityType***: Enumerador para indicar o tipo de utilizador. Foram adicionados 5 tipos, mas posteriormente poderão ser alterados e/ou acrescentados outros.
  - ***entityAddress : address***: Endereço publico da entidade. Este endereço serve como identificador único pois o *address* nunca se repete dentro da *blockchain*.
  - ***timestamp : uint256***: Momento exato da criação daquele registo.
  - ***isApproved : bool***: Indica se o registo foi aprovado por algum administrador.
  - ***approverAddress : address***: Endereço único do responsável pela aprovação.
- ***EvidenceFile -*** Classe que representa os dados referentes a um arquivo digital adicionado a um registo. Aqui, serão guardados os seguintes campos:
  - ***evidenceUniqueCode : string***: O identificador único de uma evidência.
  - ***id : string***: Identificador único do arquivo na base de dados *MongoDB*.
  - ***fileName : string***: Nome do arquivo adicionado.
  - ***fileHash : string***: *Hash* do arquivo gerada no momento do *upload*.
- ***AllowedUser -*** Classe que representa permissão do utilizador do sistema perante uma evidência. Aqui, serão guardados os seguintes campos:
  - ***evidenceUniqueCode : string***: O identificador único de uma evidência.
  - ***entity : EntityType***: Entidade que terá permissão para a visualização dos registos da evidência.

No seguimento deste documento e após a análise e especificação dos requisitos, modelação e desenho da arquitetura, será apresentada a forma como a implementação foi realizada.

4. Implementação do sistema

Este capítulo aborda a implementação do sistema, que serve como prova de conceito para solucionar o problema da rastreabilidade de evidências em processos jurídicos. Serão apresentados segmentos de código da implementação explicado o processo de *deployment* de um *smart contract* para viabilizar sua utilização por meio de uma integração com o *front-end*. Além disso, será mostrada a visão de um utilizador e como ele interagirá com a solução.

É importante destacar que a implementação dos *smart contracts* exigiu o domínio de uma linguagem completamente nova, *Solidity*, que não tinha sido previamente abordada durante o processo de estudo e qualificação. Para atender a essa necessidade e no interesse do próprio mestrando, foi concluído um curso de 14 horas intitulado *"Become a Blockchain Developer"* no *LinkedIn Learning*. O certificado deste curso pode ser encontrado nos anexos desta dissertação.

Não será comentado todo o código desenvolvido, apenas os pontos mais importantes. No entanto, é possível aceder e visualizar todo o código desenvolvido no âmbito desta dissertação pelo link do *GitHub*: `https://github.com/OleksandrS23/evidences_dapp.git`.

4.1 *Framework Truffle*

A *framework Truffle* é o primeiro passo para o desenvolvimento do projeto. Para inicializar um projeto é preciso primeiro instalar a própria *framework* através do comando `npm install -g truffle` (aqui é importante ter o *Node.js* instalado). Após isso poderá ser executado o comando `truffle init`. Com o comando executado é possível verificar a criação de 3 pastas e um arquivo, as pastas: `.\contracts`; `.\migrations`; `.\test`; e o arquivo `.\truffle-config.js` (figura 4.1):

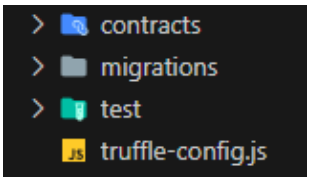


Figura 4.1: Projeto *Truffle*

Após o projeto inicializado é importante configurar o ficheiro *JSON* `.\truffle-config.js`. Aqui define-se a configuração da ligação com a *blockchain*, que, numa instância inicial, será utilizado o *Ganache*. No ponto inicial o ficheiro `.\truffle-config.js` estará maioritariamente comentado, pois o que nos interessa são as secções `networks` e `compilers` (Listagem 4.1)

```
1  (...)
2  networks: {
3      development: {          // Local Connection
4       host: "127.0.0.1",     // Localhost
5       port: 7545,            // Standard Ethereum port
6       network_id: "*",       // Any network
7      },
8      (...)
9   },
10  (...)
11  compilers: {
12     solc: {
13       version: "0.8.19",
14        settings: {          // Optimization and evmVersion
15         optimizer: {
16           enabled: false,
17           runs: 200
18         },
19         evmVersion: "byzantium"
20        }
21     }
22   },
```

Listagem 4.1: Ficheiro de configuração do Truffle

Na secção `networks` (linhas 2-7) foi acrescentado a secção `development` onde estão as configurações para a ligação com a *blockchain* privada, o *Ganache*. Na secção `compilers` (linhas 11-22) estão as configurações relacionada com o compilador de *solidity* nomeadamente a versão e a otimização na compilação. Depois de o ficheiro estar configurado, podemos prosseguir para a escrita dos *smart contracts*.

Os *smart contracts* são colocados na pasta `.\contracts` e são ficheiro terminados com a extensão `.sol` ex: `.\SmartContract.sol`. Aqui ainda não será apresentado o código dos *smart contracts* pois, inicialmente, irá ser explicado o processo de um *deploy* de um *smart contract* para uma *blockchain* privada.

Após o *smart contract* ter sido criado, é necessário criar um arquivo de migração. Este arquivo é um *script* em *JavaScript* e será responsável pelo processo de *deploy*. Na Listagem 4.2, é possível analisar o conteúdo de um ficheiro de migração.

```
1  const EvidenceChain = artifacts.require("./EvidenceChain.sol");
2
3  module.exports = async function(deployer) {
4      deployer.deploy(EvidenceChain);
5  };
```

Listagem 4.2: Ficheiro migrations do Truffle

Primeiramente, (linha 1) é feita a leitura do *smart contract*, de seguida, define-se a função para fazer o *deploy* do *smart contract*. Após o ficheiro ter sido criado será necessário executar o comando `truffle migrate` e o *smart contract* será compilado e migrado para a *blockchain* privada.

4.2 *Smart Contracts*

Para atender aos requisitos funcionais definidos na secção 3.2 e os casos de uso na secção 3.3, foram criados quatro *smart contracts* que se interligam entre si.

Um *smart contract* servirá como o ponto de ligação com o "exterior", ou seja, o contrato que funcionará como uma "ponte" para a comunicação com as *dApps*. Esse será o *smart contract* principal (`./EvidencesChain.sol`). De seguida, existem outros dois *smart contracts* que podem ser considerados como *"Managers"*, um *manager* para as entidades (`./EntitiesManager.sol`) e outro *manager* para as evidências
(`./EvidencesManager.sol`). Inicialmente, essa divisão seria suficiente, contudo, com o progresso do desenvolvimento e o aumento do código dos *smart contracts*, surgiu a necessidade de subdividir o *smart contract* responsável pelos registos de evidências devido a limitações no *deploy* dos *smart contracts*. Portanto, o quarto arquivo será equivalente a um *partial* do *manager* de evidências (`./EvidencesManager AllowedUsers.sol`).

De seguida, será apresentada a implementação dos *smart contracts* para cada caso de uso definido na secção 3.3.

4.2.1 *Manager* de evidências

Aqui serão apresentadas as implementações relativas ao diagrama de casos de uso de evidências (Secção 3.3.2).

```
1  struct Evidence {
2      string uniqueCode;
3      string caseNo;
4      EvidenceClassification classification;
5      string name;
6      EvidenceType eType;
7      EntitiesManager.Entity owner;
8      uint256 timestamp;
9      string [] uFilesCodes;
10     bool isActive;
11     string observations;
12 }
13
14 mapping(string => Evidence[]) public evidencesHistory;
```

Listagem 4.3: Estrutura de uma evidência

A Listagem 4.3 mostra o código referente a um registo de evidência no *smart contract*. Uma evidência (linhas 1-11) é formada por vários atributos, estes correspondem aos atributos definidos no modelo de domínio (Secção 3.5).

A instrução `mapping(string => Evidence[])` (linha 14) representa a matriz onde serão armazenados os registos das evidências. Na prática, pode ser lido como: `evidencesHistory["evidence1"]` correspondente a todos os registos da evidência 'evidence1' e `evidencesHistory["evidence1"][0]` correspondente ao primeiro registo de todos os registos da evidência 'evidence1'.

```
1  struct EvidenceFile {
2          string id;
3          string fileName;
4          string fileHash;
5      }
6
7      mapping(string => EvidenceFile[]) public evidencesFiles;
```

Listagem 4.4: Estrutura de arquivo de uma evidência

A Listagem 4.4 mostra o código referente a um registo de um arquivo de evidência no *smart contract*. A instrução `mapping(string => EvidenceFile[])` (linha 7) representa a matriz onde serão armazenados os registos sobre os arquivos das evidências, por exemplo `evidencesFiles["evidence1"][0]` corresponde ao primeiro arquivo da evidência 'evidence1'.

```
1      mapping(string => EntitiesManager.Entity[]) public evidencesAllowedUsers;
```

Listagem 4.5: Estrutura de registos de utilizadores permitidos

A Listagem 4.5 representa a matriz onde serão armazenados os utilizadores permitidos para a visualização de registos de uma determinada evidência, por exemplo `evidencesAllowedUsers[0]` corresponde a lista de todos os utilizadores que têm permissão para a visualização dos registos da evidência '0'.

```
function getLastUpdate (string memory _evidenceId)
public view returns (Evidence memory)
{
    uint totalVersions = evidencesHistory[_evidenceId].length;
    require(totalVersions > 0, "Evidence not found");
    return evidencesHistory[_evidenceId][totalVersions - 1];
}
```

Listagem 4.6: GET do ultimo registo de uma evidência

Na Listagem 4.6, está representada a função responsável por retornar o último registo de uma evidência específica. Ela também valida se a evidência existe; caso não exista, é retornado o erro `"Evidence not found"`.

Registar uma evidência

Nas Listagens 4.7 e 4.8, são apresentadas funções para inserir um registo de evidência. No contrato de entrada (Listagem 4.7), é feita a chamada (`entitiesManager.getLastUpdate(msg.sender)`) para obter os dados mais atualizados sobre a entidade que está a inserir a evidência e estabelecer a conexão com o contrato responsável pelas evidências (`EvidencesManager.sol`).

```
function addEvidence(
  string memory _evidenceUniqueCode,
  string memory _caseNo,
  EvidencesManager.EvidenceClassification _classification,
  string memory _evidenceName,
  EvidencesManager.EvidenceType _evidenceType
)
public
{
  EntitiesManager.Entity memory _entity = entitiesManager.getLastUpdate(msg.sender);

  evidencesManager.addEvidence(
      _evidenceUniqueCode,
      _caseNo,
      _classification,
      _evidenceName,
      _evidenceType,
      _entity
  );
}
```

Listagem 4.7: Registar uma evidência (*EvidencesChain.sol*)

```
1  function addEvidence(
2          string memory _evidenceId,
3          string memory _caseNo,
4          EvidenceClassification _classification,
5          string memory _evidenceName,
6          EvidenceType _evidenceType,
7          EntitiesManager.Entity memory _entity
8  )
9  public {
10   if (evidencesHistory[_evidenceId].length == 0)
11   {
12       evidencesHistory[_evidenceId].push(
13           Evidence(
14               _evidenceId,
15               _caseNo,
16               _classification,
17               _evidenceName,
18               _evidenceType,
19               _entity,
20               block.timestamp,
21               new string[](0),
22               true,
23               obsToString(ObservationsTipified.Created)
24           )
25       );
26       allEvidences.push(_evidenceId);
27   } else {
28       Evidence memory lastUpdateEvidence = getLastUpdate(_evidenceId);
29       require(lastUpdateEvidence.owner.entityAddress == _entity.entityAddress, "Not
             Owner");
30       require(lastUpdateEvidence.isActive == true, "Evidence inactive");
31       evidencesHistory[_evidenceId].push(
32           Evidence(
33               _evidenceId,
34               _caseNo,
35               _classification,
36               _evidenceName,
37               _evidenceType,
38               _entity,
39               block.timestamp,
40               new string[](0),
41               true,
42               obsToString(ObservationsTipified.Updated)
43           )
44       );
45   }
46
47      evidencesManagerAllowedUsers.addAllowedUser(_evidenceId, _entity);
48 }
```

Listagem 4.8: Registar uma evidência (*EvidencesManager.sol*)

No contrato do *manager* de evidências (Listagem 4.8), é realizada a validação para verificar se já existe um registo anterior com o mesmo identificador de evidência. Se não existir, uma nova instância de `Evidence` é criada e adicionada à matriz de todas as evidências (`evidencesHistory`). Se houver um registo anterior com o mesmo identificador de evidência, é realizada a validação do *owner* da evidência. Se o utilizador que está a inserir a evidência for o mesmo que o responsável do último registo da mesma evidência, o novo registo é adicionado. Caso contrário, é gerado um erro com a mensagem `"Not Owner"` . Além disso, é verificado o estado da evidência para garantir que ela se encontra ativa. Após a inserção do registo, é feita a adição do próprio utilizador à lista de entidades autorizadas a visualizar o registo (Listagem 4.8, l.47).

Adicionar um arquivo digital

Nas Listagens 4.9 e 4.10, são apresentadas as funções responsáveis pela inserção de informações sobre os arquivos adicionados pelos utilizadores. Nesse processo, apenas os dados relacionados aos arquivos são inseridos, não os arquivos em si, uma vez que esses arquivos serão armazenados numa base de dados *MongoDB*.

```
function addFile(
    string memory _evidenceUniqueCode,
    string memory _fileId,
    string memory _fileName,
    string memory _fileHash
)
public onlyOwner(_evidenceUniqueCode)
{
    EvidencesManager.EvidenceFile memory file = EvidencesManager.EvidenceFile(
        _fileId, _fileName, _fileHash);
    evidencesManager.addFile(_evidenceUniqueCode, file);
}
```

Listagem 4.9: Adicionar um arquivo digital (*EvidencesChain.sol*)

```
function addFile(string memory _evidenceId, EvidenceFile memory _file)
public
{
    Evidence memory evidence = getLastUpdate(_evidenceId);
    require(evidence.isActive == true, "Evidence inactive");

    string[] memory fileIds = new string[](1);
    fileIds[0] = _file.id;

    evidencesFiles[_evidenceId].push(EvidenceFile(_file.id, _file.fileName, _file.
        fileHash));
    evidencesHistory[_evidenceId].push(Evidence(_evidenceId, evidence.caseNo,
        evidence.classification, evidence.name, evidence.eType, evidence.owner,
        block.timestamp, fileIds, true, obsToString(ObservationsTipified.AddFile)));
}
```

Listagem 4.10: Adicionar um arquivo digital (*EvidencesManager.sol*)

```
1  modifier onlyOwner(string memory _evidenceUniqueCode)
2  {
3      EvidencesManager.Evidence memory _evidence = evidencesManager.getLastUpdate(
           _evidenceUniqueCode);
4      require(msg.sender == _evidence.owner.entityAddress, "Not Owner");
5      _;
6  }
```

Listagem 4.11: Modificador "*onlyOwner*"

Na Listagem 4.9, linha 7, é efetuada a validação `onlyOwner` com o uso do modificador `onlyOwner` (Listagem 4.11). Conforme o nome sugere, essa validação verifica se o utilizador que está a adicionar os arquivos a uma evidência é o responsável pela mesma. Caso não seja o responsável, é retornado o erro `"Not Owner"` . Se o utilizador for, de fato, o responsável, é criada uma instância do arquivo `EvidencesManager.EvidenceFile` e enviada para o *smart contract* seguinte.

Na Listagem 4.10 é implementada a função *addFile*. Nessa implementação ocorre:

- Verificação se a evidência ainda está ativa (linha 4-5);
- É criado um *array* de strings para armazenamento dos identificadores dos arquivos (linha 7);
- São adicionados os dados sobre o arquivo na matriz dos arquivos de evidência (`evidencesFiles`) (linha 10);
- É criado um novo registo de evidência no histórico, para registar que houve uma alteração (linha 11);

É importante notar que, embora a estrutura da evidência esteja preparada para receber múltiplos ficheiros para um registo apenas, optou-se por adicionar somente um arquivo por registo. Dessa forma, se forem adicionados dois ficheiros, cada ficheiro criará um novo registo de evidência.

Adicionar *"allowed users"*

Nas Listagens 4.12, 4.13 e 4.14 estão representadas funções referentes a adição de entidades permitidas à visualização de uma evidência.

Na Listagem 4.12, é possível observar a verificação com o modificador `onlyOwner` (Listagem 4.11). A função recebe uma lista de endereços, e após a verificação do utilizador, que deve ser confirmado como o responsável pela evidência, é feita a chamada `getLastUpdate` para cada endereço recebido na função. Isso é feito para validar se as entidades que estão a ser inseridas existem no sistema e são elegíveis para serem adicionadas à matriz de permissões de visualização.

```
1  function addAllowedUsers(
2          string memory _evidenceUniqueCode,
3          address[] memory _entities
4  )
5  public onlyOwner(_evidenceUniqueCode)
6  {
7      EntitiesManager.Entity[] memory entities = new EntitiesManager.Entity[](
8          _entities.length
9      );
10     for (uint32 i = 0; i < _entities.length; i++) {
11         entities[i] = entitiesManager.getLastUpdate(_entities[i]);
12     }
13     evidencesManager.addAllowedUsers(_evidenceUniqueCode, entities);
14 }
```

Listagem 4.12: Adicionar "allowed users" (*EvidencesChain.sol*)

```
1  function addAllowedUsers (string memory _evidenceId, EntitiesManager.Entity[] memory
       _entities)
2  public
3  {
4      Evidence memory evidence = getLastUpdate(_evidenceId);
5      require(evidence.isActive == true, "Evidence inactive");
6
7      for (uint32 i = 0; i < _entities.length; i++) {
8          evidencesManagerAllowedUsers.addAllowedUser(_evidenceId, _entities[i]);
9          string memory _obs= string(abi.encodePacked(obsToString(ObservationsTipified
               .AddAllowedUser), ' ', _entities[i].userName));
10         evidencesHistory[_evidenceId].push(Evidence(_evidenceId, evidence.caseNo,
               evidence.classification, evidence.name, evidence.eType, evidence.owner,
               block.timestamp, new string[](0), true, _obs));
11     }
12 }
```

Listagem 4.13: Adicionar "allowed users" (*EvidencesManager.sol*)

Na Listagem 4.13 é implementada a função *addAllowedUser*. Nessa implementação ocorre:

- Verificação se a evidência ainda está ativa (linha 4-5);
- Para cada entidade da lista de entidades recebida é executada a função *addAllowedUser* do *smart contract* *EvidencesManagerAllowedUsers.sol* (Listagem 4.14) onde a mesma será adicionada á matriz de entidades permitidas (`evidencesAllowedUsers`) mas antes é verificada se a entidade a ser adicionada já se encontra na matriz para evitar duplicações (linha 8);
- É Criada uma string de observações sobre a operação executada (linha 9);
- É criado um novo registo de evidência no histórico, para registar que houve uma alteração (linha 10);

```
1  function addAllowedUser(string memory _evidenceUniqueCode, EntitiesManager.Entity
       memory _entity)
2  public
3  {
4      bool alreadyAllowed = false;
5      for (uint32 i = 0; i < evidencesAllowedUsers[_evidenceUniqueCode].length; i++) {
6          EntitiesManager.Entity memory entity = evidencesAllowedUsers[
               _evidenceUniqueCode][i];
7
8          if (entity.entityAddress == _entity.entityAddress){
9              alreadyAllowed = true;
10         }
11     }
12     if(alreadyAllowed == false){
13         evidencesAllowedUsers[_evidenceUniqueCode].push(_entity);
14     }
15 }
```

Listagem 4.14: Adicionar "allowed users" (*EvidencesManagerAllowedUsers.sol*)

Transferir o *ownership* para outro utilizador

Nas Listagens 4.15 e 4.16 estão representadas as funções responsáveis por atualizar o responsável por uma evidência. Na Listagem 4.15, tal como nas anteriores, é possível observar que também esta a ser reutilizado o modificador `onlyOwner`. Na função é recebido o endereço do novo responsável pela evidência, se a verificação for concluída com sucesso, então é efetuado o *get* do último registo da entidade e enviado para a função seguinte.

```
function evidenceNewOwner(
        string memory _evidenceUniqueCode,
        address _newOwner
) public onlyOwner(_evidenceUniqueCode)
{
    EntitiesManager.Entity memory _entity = entitiesManager.getLastUpdate(
        _newOwner
    );
    evidencesManager.updateOwner(_evidenceUniqueCode, _entity);
}
```

Listagem 4.15: Transferir o "ownership" para outro utilizador (*EvidencesChain.sol*)

```
function updateOwner(string memory _evidenceId, EntitiesManager.Entity memory
    _entity)
public
{
    Evidence memory evidence = getLastUpdate(_evidenceId);
    require(evidence.isActive == true, "Evidence inactive");

    evidencesHistory[_evidenceId].push(Evidence(_evidenceId, evidence.caseNo,
        evidence.classification, evidence.name, evidence.eType, _entity, block.
        timestamp, new string[](0), true, obsToString(ObservationsTipified.
        UpdatedOwner)));

    if (evidence.classification == EvidenceClassification.AllowedPersonalOnly){
        evidencesManagerAllowedUsers.addAllowedUser(_evidenceId, _entity);
    }
}
```

Listagem 4.16: Transferir o "ownership" para outro utilizador (*EvidencesManager.sol*)

Na Listagem 4.16 é implementada a função *updateOwner*. Nessa implementação ocorre:

- Verificação se a evidência ainda está ativa (linha 4-5);
- É criado um novo registo de evidência no histórico, para registar o novo responsável pelo registo (linha 7);
- É verificada a necessidade de o novo responsável ser adicionado à matriz de permissões de visualização, caso seja necessário, é adicionado (linhas 9-10);

Visualizar a lista de registos de evidências permitidas

Nas Listagens 4.17 e 4.18 estão representadas as funções responsáveis por retornar os registos das evidências aos quais a entidade tem permissão de visualização. Na Listagem 4.17, é feito o *get* do último registo da entidade e enviado para a função da Listagem a seguir (linhas 5-6).

```
function getEvidences()
public view
returns (EvidencesManager.Evidence[] memory)
{
    EntitiesManager.Entity memory entity = entitiesManager.getLastUpdate(msg.sender)
        ;
    return evidencesManager.getEvidences(entity);
}
```

Listagem 4.17: Visualizar a lista de registos de evidências permitidas (*EvidencesChain.sol*)

```
1  function getEvidences(EntitiesManager.Entity memory entity)
2  public view returns (EvidencesManager.Evidence[] memory)
3  {
4      require(allEvidences.length > 0, "No Evidences");
5      EvidencesManager.Evidence[] memory evidences = new EvidencesManager.Evidence[](
           allEvidences.length);
6      uint256 count = 0;
7      for (uint32 i = 0; i < allEvidences.length; i++) {
8          Evidence memory evidence = getLastUpdate(allEvidences[i]);
9          if (evidencesManagerAllowedUsers.checkIfAllowed(evidence, entity))
10         {
11             evidences[count] = evidence;
12             count++;
13         }
14     }
15     // Resize the memory array to the actual number of filtered results
16     assembly {
17         mstore(evidences, count)
18     }
19     return evidences;
20 }
```

Listagem 4.18: Visualizar a lista de registos de evidências permitidas (*EvidencesManager.sol*)

Na Listagem 4.18, é feita a verificação da existência de evidências. Em caso de não existirem evidências, o contrato não continua a executar a função e é retornado *"No Evidences"*. Em caso de existirem evidências, é feita a verificação `checkIfAllowed` para cada último registo de cada evidência. A função `checkIfAllowed` está representada na Listagem a seguir 4.19. Em caso da função `checkIfAllowed` retornar *true* então o registo da evidencia será retornado. O `assembly { mstore }` (linhas 18-19) serve para fazer *resize* do *array* criado previamente para armazenadas os registos das evidências aos quais o utilizador tem permissão.

```
1  function checkIfAllowed (EvidencesManager.Evidence memory evidence, EntitiesManager.
       Entity memory entity)
2  public view returns (bool)
3  {
4      if (entity.entityType == EntitiesManager.EntityType.Court)
5      {
6          return true;
7      }
8
9      if (evidence.classification == EvidencesManager.EvidenceClassification.Public)
10     {
11         return true;
12     }
13
14     if (entity.entityType == EntitiesManager.EntityType.Forensics && evidence.
           classification != EvidencesManager.EvidenceClassification.
           AllowedPersonalOnly)
15     {
16         return true;
17     }
18
19     if (evidence.classification == EvidencesManager.EvidenceClassification.
           AllowedPersonalOnly || evidence.classification == EvidencesManager.
           EvidenceClassification.Secret){
20         EntitiesManager.Entity[] memory allowedEntities= evidencesAllowedUsers[
               evidence.uniqueCode];
21         for (uint32 i = 0; i < allowedEntities.length; i++){
22             EntitiesManager.Entity memory allowed = allowedEntities[i];
23             if (entity.entityAddress == allowed.entityAddress){
24                 return true;
25             }
26         }
27     }
28
29     return false;
30 }
```

Listagem 4.19: Verificação de permissão de visualização do utilizador

Na Listagem 4.19, está representada a função responsável por verificar se um determinado utilizador tem permissão para visualizar um registo de evidência. Se o utilizador for do tipo *Court* (linha 4), então a função sempre retorna *true*, o que significa que ele pode visualizar todas as evidências. Se a evidência for do tipo *Public* (linha 9), a função também retorna *true*, permitindo que todos visualizem as evidências públicas. Se o utilizador for do tipo *Forensics* e a evidência não for do tipo *Allowed Personal Only* (linha 14), a função retorna *true*, permitindo que o utilizador veja os registos. No entanto, se a evidência for do tipo *Allowed Personal Only* ou *Secret* (linha 19), a função verifica a matriz de entidades permitidas para visualização. Se o utilizador estiver na matriz, a função retorna *true*; caso contrário, retorna *false*, indicando que o utilizador não pode visualizar essa evidência (linhas 20-26).

Visualizar todo o histórico do registo de uma evidência

Nas Listagens 4.20 e 4.21, estão representadas as funções responsáveis por retornar todo o histórico de uma determinada evidência. Na Listagem 4.20, é feito o *get* do último registo da entidade e enviado para a função `getEvidenceHistory` do *manager*.

```
function getEvidenceHistory(string memory _evidenceUniqueCode )
public view returns (EvidencesManager.Evidence[] memory)
{
    EntitiesManager.Entity memory entity = entitiesManager.getLastUpdate(msg.sender)
        ;

    return evidencesManager.getEvidenceHistory(_evidenceUniqueCode, entity);
}
```

Listagem 4.20: Visualizar todo o histórico do registo de uma evidência (*EvidencesChain.sol*)

```
function getEvidenceHistory(string memory _evidenceId, EntitiesManager.Entity memory
     entity)
public view returns (Evidence[] memory)
{
    Evidence memory evidence = getLastUpdate(_evidenceId);
    require(evidencesManagerAllowedUsers.checkIfAllowed(evidence, entity), "Not
        Allowed");

    return evidencesHistory[_evidenceId];
}
```

Listagem 4.21: Visualizar todo o histórico do registo de uma evidência (*EvidencesManager.sol*)

Na Listagem 4.21, está representada a função `getEvidenceHistory` do *manager*, onde é feita a verificação se o utilizador tem permissão para visualizar os registos da evidência.

Verificar a integridade dos arquivos

Nas Listagens 4.22 e 4.23 estão representadas as funções que retornam os dados armazenados no momento da adição de arquivos numa evidência. Esses dados são cruciais para uma posterior verificação e comparação dos arquivos que se encontram armazenados na base de dados, a fim de garantir que eles estão intactos.

```
function getFileInfo(string memory _evidenceUniqueCode, string memory
    _fileUniqueCode)
external view returns (EvidencesManager.EvidenceFile memory)
{
    return evidencesManager.getFileInfo(_evidenceUniqueCode, _fileUniqueCode);
}
```

Listagem 4.22: Informações sobre o arquivo (*EvidencesChain.sol*)

```
function getFileInfo(string memory _evidenceId, string memory _fileId)
public view returns (EvidenceFile memory)
{
    EvidenceFile memory evidenceFile;

    for (uint256 i = 0; i < evidencesFiles[_evidenceId].length; i++) {
        if (keccak256(abi.encodePacked(evidencesFiles[_evidenceId][i].id)) ==
            keccak256(abi.encodePacked(_fileId))) {
            return evidencesFiles[_evidenceId][i];
        }
    }
    return evidenceFile;
}
```

Listagem 4.23: Informações sobre o arquivo (*EvidencesManager.sol*)

4.2.2 *Manager* de entidades

Aqui serão apresentadas as implementações relativas ao diagrama de casos de uso de entidades (Secção 3.3.2).

```
1  struct Entity {
2      string userName;
3      string password;
4      string name;
5      string departmentCode;
6      EntityType entityType;
7      address entityAddress;
8      uint256 timestamp;
9      bool isApproved;
10     address approverAddress;
11 }
12
13 mapping(address => Entity[]) public entitiesHistory;
```

Listagem 4.24: Estrutura de uma entidade

A Listagem 4.24 apresenta o código referente ao registo de entidade no *smart contract*. A instrução `mapping(address => Entity[])` (linha 13) representa a estrutura onde serão armazenados os registos das entidades. Na prática, pode ser lido como: `entitiesHistory[{address}]` correspondente a todos os registos da uma entidade '{address}' e `entitiesHistory[{address}][0]` correspondente ao primeiro registo de todos os registos da entidade '{address}'.

```
1 function getLastUpdate (address _entity) public view returns (Entity memory) {
2     uint totalVersions = entitiesHistory[_entity].length;
3     require(totalVersions > 0, "Entity Not Found");
4     return entitiesHistory[_entity][totalVersions - 1];
5 }
```

Listagem 4.25: GET do ultimo registo de uma entidade

Na Listagem 4.25, está representada a função responsável por retornar o último registo de uma entidade específica. Ela também valida se a entidade existe; caso não exista, é retornado o erro `"Entity not found"`.

Efetuar o registo no sistema

Nas Listagens 4.26 e 4.27 estão representadas as funções responsáveis por efetuar o registo de uma entidade no sistema.

```
1 function addEntity(address _entity, string memory _userName, string memory _password
      , string memory _name, string memory _departmentCode, EntitiesManager.EntityType
       _entityType)
2 public
3 {
4     entitiesManager.createEntity(_entity, _userName, _password, _name,
          _departmentCode, _entityType);
5 }
```

Listagem 4.26: Efetuar o registo do sistema (*EvidencesChain.sol*)

```
1  function createEntity(address _entity, string memory _userName, string memory
       _password, string memory _name, string  memory _departmentCode, EntityType
       _entityType)
2  public
3  {
4      require(entitiesHistory[_entity].length == 0, "Already Registred");
5
6      if (entitiesHistory[_entity].length == 0)
7      {
8          entitiesHistory[_entity].push(Entity(_userName, _password, _name,
               _departmentCode, _entityType, _entity,  block.timestamp, false, _entity)
               );
9          allEntities.push(_entity);
10     }
11 }
```

Listagem 4.27: Efetuar o registo do sistema (*EntitiesManager.sol*)

Na Listagem 4.26, é feita a chamada da função do *manager* (Listagem 4.27) (linha 4). Esta função simplesmente encaminha a chamada para o contrato do *manager* de entidades.

Na Listagem 4.27, a função verifica se já existe um registo de utilizador com o mesmo endereço. Se existir, a função retorna o erro `"Already Registered"` (linha 4). Caso contrário, a entidade é adicionada à matriz de entidades, efetuando o registo no sistema (linha 8).

Aprovar o registo do utilizador

Nas Listagens 4.28 e 4.29 estão representadas as funções responsáveis por efetuar a aprovação de um registo de uma entidade no sistema.

```
1  function approveEntity(address _entityToBeApproved)
2  public
3  {
4      entitiesManager.approveEntity(msg.sender, _entityToBeApproved);
5  }
```

Listagem 4.28: Aprovar o registo do utilizador (*EvidencesChain.sol*)

```
1  function approveEntity(address _entity, address _entityToBeApproved)
2  public
3  {
4      Entity memory entity = getLastUpdate(_entity);
5      require(entity.entityType == EntityType.Administrator, "Not Administrator");
6
7      Entity memory entityToBeApproved = getLastUpdate(_entityToBeApproved);
8      require(entityToBeApproved.isApproved == false, "Already Approved");
9
10     entitiesHistory[_entityToBeApproved].push(Entity(entityToBeApproved.userName,
           entityToBeApproved.password, entityToBeApproved.name, entityToBeApproved.
           departmentCode, entity.entityType, entityToBeApproved.entityAddress,  block.
           timestamp, true, _entity));
11 }
```

Listagem 4.29: Aprovar o registo do utilizador (*EntitiesManager.sol*)

Na Listagem 4.28, a função recebe o endereço da entidade que pretende ser aprovada e é feita a chamada da função do *manager* (Listagem 4.29) (linha 4). Esta função simplesmente encaminha a chamada para o contrato do *manager* de entidades.

Na Listagem 4.29, a função `approveEntity` faz o *get* do registo da entidade que irá aprovar para verificar se o mesmo é um *administrator*. Em seguida, é feito o *get* do último registo da entidade que será aprovada para verificar se a mesma precisa de aprovação. Se todas as verificações forem bem-sucedidas, um novo registo é adicionado à matriz de entidades, com o campo *isApproved* definido como *true* e o endereço da entidade que o aprovou.

Atualizar a informação do registo

Nas Listagens 4.30 e 4.31, estão representadas as funções responsáveis pela atualização das informações do registo da entidade.

```
1 function updateEntity(address _entity, string memory _userName, string memory
      _password, string memory _name, string memory _departmentCode)
2 public
3 {
4     entitiesManager.updateEntity(_entity, _userName, _password, _name,
          _departmentCode);
5 }
```

Listagem 4.30: Atualizar a informação do registo (*EvidencesChain.sol*)

```
1 function updateEntity(address _entity, string memory _userName, string memory
      _password, string memory _name, string  memory _departmentCode) public {
2         Entity memory entity = getLastUpdate(_entity);
3         require(keccak256(bytes(entity.userName)) == keccak256(bytes(_userName)), "
              User Name Didn't Match");
4         require(keccak256(bytes(entity.password)) == keccak256(bytes(_password)), "
              Password Didn't Match");
5
6         entitiesHistory[_entity].push(Entity(_userName, _password, _name,
              _departmentCode, entity.entityType, _entity,  block.timestamp, false,
              _entity));
7     }
```

Listagem 4.31: Atualizar a informação do registo (*EntitiesManager.sol*)

Na Listagem 4.30, a função `updateEntity` apenas reencaminha os dados para a função do *manager* da entidade.

Na Listagem 4.31, a função recebe os dados do utilizador a serem atualizados. A função começa por obter o último registo da entidade e, em seguida, verifica se o utilizador e a *password* correspondem à entidade. Se não corresponderem, a função retorna o erro correspondente. Se as informações estiverem corretas, o novo registo é adicionado à matriz de entidades com as informações atualizadas. Isso permite que os utilizadores atualizem as suas informações no sistema.

Visualizar todos os utilizadores registados

Nas Listagens 4.32 e 4.33, estão representadas as funções referentes à visualização de todos os utilizadores registados. Neste caso, queremos sempre mostrar todos os utilizadores existentes no sistema, portanto, não existe qualquer validação específica. Na linha 4 da Listagem 4.32, são recebidos todos os endereços dos utilizadores registados no sistema. Para cada endereço recebido, é feito o *get* do último registo de cada entidade (linhas 8-10).

```
1  function getEntities()
2  public view returns (EntitiesManager.Entity[] memory)
3  {
4      address[] memory entityAddresses = entitiesManager.getEntities();
5      require(entityAddresses.length > 0, "No Entities");
6
7      EntitiesManager.Entity[] memory entities = new EntitiesManager.Entity[](
           entityAddresses.length);
8      for (uint32 i = 0; i < entityAddresses.length; i++) {
9          entities[i] = entitiesManager.getLastUpdate(entityAddresses[i]);
10
11     return entities;
12 }
```

Listagem 4.32: Visualizar todos os utilizadores registados (*EvidencesChain.sol*)

```
function getEntities()
public view returns (address[] memory)
{
        return allEntities;
}
```

Listagem 4.33: Visualizar todos os utilizadores registados (*EntitiesManager.sol*)

Visualizar todo o histórico de atualizações

Nas Listagens 4.34 e 4.35, estão representadas as funções referentes à visualização de todo o histórico de atualizações de uma entidade. A função `getEntityHistory` (Listagem 4.34, l.3) somente faz a chamada da função do *manager* da entidade, e no *manager* (Listagem 4.35, l.4), são retornados todos os registos referentes a uma entidade, mostrando o histórico de atualizações ao longo do tempo.

```
function getEntityHistory() public view returns (EntitiesManager.Entity[] memory)
{
    return entitiesManager.getEntityHistory(msg.sender);
}
```

Listagem 4.34: Visualizar todo o histórico de atualizações (*EvidencesChain.sol*)

```
function getEntityHistory(address _entity)
public view returns (Entity[] memory)
{
    return entitiesHistory[_entity];
}
```

Listagem 4.35: Visualizar todo o histórico de atualizações (*EntitiesManager.sol*)

Efetuar *login* no sistema

Na Listagem 4.36, está representada a função de autorização de um utilizador no sistema via *username* e *password*. Na função, é feito o *get* do último registo da entidade e feita a comparação se o *username* e a *password* do registo coincidem com o *username* e a *password* fornecidos pelo utilizador. Isso garante que o utilizador é autorizado com base nas credenciais fornecidas.

```
function authenticateEntity(string memory _uname, string memory _pass)
public view returns (bool)
{
    EntitiesManager.Entity memory entity = entitiesManager.getLastUpdate(msg.sender)
        ;

    if (
        keccak256(abi.encodePacked(entity.userName)) ==
        keccak256(abi.encodePacked(_uname))
    ) {
        if (
            keccak256(abi.encodePacked(entity.password)) ==
            keccak256(abi.encodePacked(_pass))
        ) {
            return (true);
        }
    }
    return (false);
}
```

Listagem 4.36: Efetuar login no sistema (*EvidencesChain.sol*)

4.3 *Back-end*

O *back-end* foi desenvolvido com a *framework NestJS*, como referido na secção 3.4.1. A principal razão para a criação do *back-end* é a necessidade de conexão com uma base de dados *NoSQL MongoDB*

para o armazenamento de arquivos digitais. Também será realizada aqui a verificação da integridade dos arquivos digitais.

Para o desenvolvimento do *back-end*, utilizou-se o comando `nest new api` dentro da raiz do projeto, que foi previamente inicializado com a *framework Truffle*, conforme referido na secção 4.1. Isso inicializou um novo projeto do *NestJS* na pasta `.\api`. A partir desse ponto, inicializou-se a implementação dos *endpoints* necessários e das conexões com o *MongoDB*.

4.3.1 Ligação MongoDB

Antes de conectar o *Nest* com o *MongoDB*, é necessário criar uma base de dados *MongoDB*. Para fazer isso, foi necessário seguir as seguintes etapas:

1. Aceder ao site oficial do *MongoDB*.
2. Criar uma organização e um projeto dentro do *MongoDB*.
3. Dentro do projeto, criar um *cluster* que será o local onde estará a base de dados.
4. Abrir o *cluster* criado e aceder as *collections*.
5. Nas *collection*, criar a *database* com o nome `EvidencesChain`.

Após a base de dados estar criada, foi necessário copiar a *connection string* para ser possível a ligação do *NestJS* com o *MongoDB*. A *connection string* contém informações de autenticação, como nome de utilizador e *password*, e o endereço do *cluster MongoDB*. Isso permitirá que o *NestJS* se conecte a base de dados e realize operações de leitura e escrita.

dotenv

É importante não compartilhar publicamente a *connection string* do *MongoDB*, uma vez que ela contém informações sensíveis. Para proteger essas informações e separá-las do código, é uma boa prática utilizar o pacote *dotenv*. O *dotenv* permite definir variáveis de ambiente num arquivo de configuração separado (geralmente chamado de `.env`) acessível no código.

Para instalar o *dotenv* foi executado o comando `npm install dotenv`. De seguida, num arquivo `.env` foi adicionada a seguinte variável de ambiente (Listagem 4.37)

```
1 CONNECTION_STRING="mongodb+srv://{user}:{password}@cluster0.4xfwqs5.mongodb.net/
      EvidencesChain"
```

Listagem 4.37: Connection String MongoDB .env

O *user* e *password* foram removidos por motivos de segurança.

Após estar adicionada no ficheiro `.env` é possível a sua utilização da seguinte forma (Listagem 4.38)

```
1 import 'dotenv/config';
2
3 const mongoConnectionString = process.env.CONNECTION_STRING;
```

Listagem 4.38: Usar variavel de ambiente em app.module.ts

É preciso, inicialmente, importar o *dotenv config* através do `import 'dotenv/config';` e de seguida é possível a utilização das variáveis com o `process.env.{nome da variavel}`

Conectar o *Nest* com o *Mongo*

```
1  import { Module } from '@nestjs/common';
2  import { MongooseModule} from '@nestjs/mongoose';
3  import 'dotenv/config';
4  (...)
5
6  @Module({
7    imports: [
8      MongooseModule.forRoot(process.env.CONNECTION_STRING),
9      (...)
10   ],
11   exports: [MongooseModule],
12   (...)
13 })
14 export class AppModule {}
```

Listagem 4.39: Conectar Nest com Mongo (*app.module.ts*)

Na Listagem 4.39 está representada a ligação do *Nest* com o *MongoDB*. É possível compreender que está a ser utilizado o *package mongoose* do *nestjs* como descrito na secção 3.4.1. É usado o `MongooseModule.forRoot()` (linha 8) com a *connection string* definida no `.env` para estabelecer a conexão com o *MongoDB*.

4.3.2 Ligação com *smart contracts*

```
1  import { Injectable } from '@nestjs/common';
2  import Web3 from 'web3';
3  import * as contractJSON from '../contracts/EvidenceChain.json'; // Import the
       contract JSON file
4
5  @Injectable()
6  export class EthereumService {
7    private web3: Web3;
8    private contract: any;
9    constructor() {
10     this.web3 = new Web3(
11       new Web3.providers.HttpProvider(process.env.WEB3_PROVIDER),
12     );
13     this.contract = new this.web3.eth.Contract(
14       contractJSON.abi,
15       contractJSON.networks[process.env.NETWORK_ID].address,
16     );
17   }
18   (...)
19 }
```

Listagem 4.40: Serviço Ethereum Configurações (*ethereum.service.ts*)

Na Listagem 4.40, apresenta-se o segmento de código que configura o *Web3* no *back-end*. Foi criado o serviço *EthereumService*, responsável por realizar chamadas ao *smart contract*. Primeiramente, importou-se a biblioteca *Web3* no arquivo e, em seguida, no construtor, inicializou-se uma nova instância do *Web3* com o *HttpProvider* que foi definido no arquivo `.env`. O *HttpProvider* (linha 11) é o endereço usado para se conectar à blockchain. Após isso, criou-se uma instância do *smart contract* com *new this.web3.eth.Contract()* (linhas 13-16), fornecendo o *abi* e o endereço do *smart contract*.

```
1  (...)
2  async callContractTransaction(
3      from: string,
4      methodName: string,
5      ...args: any[]
6    )
7  {
8      var result = [];
9      try {
10       result = await this.contract.methods[methodName](
11         ...args
12       ).call({
```

```
13          from: from, // Specify the sender's Ethereum address
14          gas: 2000000, // Set an appropriate gas limit
15          gasPrice: '20000000000' // Set an appropriate gas price
16        });
17
18        return result;
19      } catch (error) {
20        console.error('Transaction failed:', error);
21      }
22  }
```

Listagem 4.41: Serviço Ethereum Call Contract (*ethereum.service.ts*)

Na Listagem 4.41, apresenta-se o segmento de código responsável por executar uma chamada ao *smart contract*. A função recebe três parâmetros: `from: string`, que é o endereço público do utilizador que está a realizar a chamada; `methodName: string`, que é o identificador do método no contrato; e `...args: any[]`, que é um *array* de dados a serem enviados para o método (linhas 3-5).

Para executar a chamada, é utilizada a instância do contrato criada no construtor do serviço e passamos todos os parâmetros, incluindo o nome do método e os argumentos, através de `this.contract.methods[methodName](...args)` (linhas 10-11). Em seguida, utilizamos o método `.call()`(linhas 12-16), fornecido pelo *Web3* na instância do contrato. Dentro do `.call()`, é possível fornecer parâmetros adicionais, como `from: from`, para especificar o endereço do *caller*. Também é passível explicitamente definir o limite de gás (*gas limit*) e o preço do gás (*gas price*) a ser utilizado.

Após a execução do pedido, a resposta é retornada. Caso a transação falhe, um erro é apresentado, e o retorno será vazio.

4.3.3 Autenticação e *token*

Na Listagem 4.42, apresenta-se o segmento de código correspondente ao serviço de autenticação.

```
1  import { JwtService } from '@nestjs/jwt';
2  @Injectable()
3  export class AuthService {
4    constructor(private jwtService: JwtService, private readonly ethereumService:
         EthereumService) {}
5
6    async generateToken(payload: any): Promise<string> {
7      return this.jwtService.sign(payload);
8    }
9
10   async validateUser(address: string, username: string, password: string): Promise<
         any> {
11     return await this.ethereumService.callContractTransaction(address, "
           authenticateEntity", ...[username, password]);
12   }
13
14   async verifyUser(address: string, username: string): Promise<any> {
15     const user = await this.ethereumService.callContractTransaction(address, "
           getEntity");
16
17     if (user["userName"] === username){
18       return user;
19     }
20
21     return false
22   }
23 }
```

Listagem 4.42: Serviço Autenticação (*auth.service.ts*)

Para realizar a autenticação, utilizou-se o pacote `@nestjs/jwt` do *Nest*, tornando a implementação da autenticação nos *endpoints* mais benéfica e simples. O serviço de autenticação contém três métodos importantes:

- `generateToken`: Este método recebe um *payload* e assina um *token* com o serviço *JWT*, usando o *payload* como parâmetro.
- `validateUser`: Realiza uma chamada ao método `authenticateEntity` do *smart contract* com os parâmetros de autenticação do utilizador e retorna a responsta (*true or false*).
- `verifyUser`: Efetua uma chamada ao método `getEntity` para receber as informações do utilizador e validar que o *token* pertence a pessoa que está a efetuar um pedido. Este método é chamado sempre que uma solicitação é feita a um *endpoint* do *back-end* que esteja protegido por autenticação.

Na Listagem 4.43, apresenta-se o segmento de código responsável pelo *login* de um utilizador. O método recebe três parâmetros: o endereço público do utilizador, o nome de utilizador (*user*) e a palavra-passe (*password*). De seguida, é chamado o método `validateUser` (linhas 8-12) do serviço de autenticação, conforme apresentado na Listagem 4.42. Após receber a resposta, se esta for *true*, significa que a autenticação foi bem-sucedida e pode prosseguir para a geração do *token* (linha 14), como também mostrado na Listagem 4.42. Se o *token* for obtido com sucesso, ele é retornado, indicando que a autenticação foi realizada com sucesso. Caso contrário, uma resposta de `'Invalid credentials.'` é retornada, ou uma mensagem de `'Authentication failed.'` em caso de algum erro inesperado.

```
1  @Post('login')
2  async login(@Body() credentials: any) {
3    console.log(credentials)
4    if (!credentials.password) {
5      throw new BadRequestException('Password is required.');
6    }
7    try {
8      const user = await this.authService.validateUser(
9        credentials.address,
10       credentials.user,
11       credentials.password
12     );
13     if (user) {
14         const token = await this.authService.generateToken({ address: credentials.
               address , user:  credentials.user});
15         return { token };
16     }
17
18     throw new UnauthorizedException('Invalid credentials.');
19   } catch (error) {
20     throw new InternalServerErrorException('Authentication failed.');
21   }
22 }
```

Listagem 4.43: Endpoint de login (*auth.controller.ts*)

4.3.4 Adicionar um arquivo digital

Na Listagem 4.44, apresenta-se o segmento de código responsável por adicionar um arquivo digital à base de dados *MongoDB*. Para efetuar o upload do ficheiro no *MongoDB*, optou-se por utilizar o *GridFSBucket* do pacote *mongoose*. Portanto, no construtor do serviço, o *GridFSBucket* está a ser instanciado globalmente para ser utilizado pelos métodos existentes nele (linhas 7-11).

```
1  import { InjectConnection } from '@nestjs/mongoose';
2  import { Connection } from 'mongoose';
3  import * as mongoose from 'mongoose';
4
5  @Injectable()
6  export class UploadService {
7    private gridFs: any;
8
9    constructor(@InjectConnection() private readonly connection: Connection) {
10     this.gridFs = new mongoose.mongo.GridFSBucket(this.connection.db);
11   }
12
13   async uploadFile(stream: NodeJS.ReadableStream, filename: string): Promise<string>
         {
14     const uploadStream = this.gridFs.openUploadStream(filename);
15     stream.pipe(uploadStream);
16     return new Promise<string>((resolve, reject) => {
17       uploadStream.once('finish', () => resolve(uploadStream.id.toString()));
18       uploadStream.once('error', (err) => reject(err));
19     });
20   }
21 (...)
22 }
```

Listagem 4.44: Upload de arquivos (*upload.service.ts*)

O método `uploadFile` (Listagem 4.44, l.13) é responsável por adicionar o arquivo à base de dados. A linha `this.gridFs.openUploadStream(filename);` (linha 14) abre um fluxo de gravação para fazer o upload de um arquivo com o nome *filename*. Em seguida, os dados do fluxo são redirecionados para *uploadStream* (linha 15), o que permite a gravação dos dados do arquivo no *GridFS*. Após essa operação, um *event listener* é adicionado aos eventos `finish` e `error`, o que possibilita o retorno de `uploadStream.id` em caso de sucesso ou `reject(err)` em caso de erro (linhas 16-19).

```
1  (...)
2  @UseGuards(AuthGuard('eth-jwt'))
3  @Post('upload')
4  @UseInterceptors(FilesInterceptor('files'))
5  async uploadFiles(
6    @UploadedFiles() files: Array<Express.Multer.File>,
7  ) {
8    if (!files || files.length === 0) {
9      return { message: 'No files to be uploaded' };
10   }
11   const uploadedFileIds: any = [];
12   for (const file of files) {
13     console.log(file);
14     const readableStream = stream.Readable.from(file.buffer);
15     const buffer = file.buffer;
16
17     const fileHash = crypto.createHash('sha256').update(buffer.toString()).digest('
         hex');
18     await this.uploadService
19       .uploadFile(readableStream, file.originalname)
20       .then((result) => {
21         uploadedFileIds.push({ id: result, filename: file.originalname, filehash:
             fileHash});
22         return result;
23       });
24   }
25   return { uploadedFileIds };
26 }
27 (...)
```

Listagem 4.45: Endpoint para upload de arquivos (*upload.controller.ts*)

Na Listagem 4.45, apresenta-se o segmento de código do *endpoint* responsável pelo *upload* de arquivos digitais. Este *endpoint* é protegido pela autenticação (linha 2), conforme mencionado na secção 4.3.3, obrigando o utilizador estar autenticado via *login* no sistema. Utiliza um *interceptor* de arquivos para receber os arquivos no lado do servidor (linha 4).

Após a invocação do *endpoint* e a aprovação da autorização, é realizada a validação da existência de arquivos a serem adicionados. Caso não existam arquivos, a mensagem `'No files to be uploaded'` é retornada (linhas 8-10). Se houver arquivos, para cada um deles é gerada uma *hash* do conteúdo do arquivo (linha 17). Essa *hash* é gerada com o pacote *crypto* do *Node*, considerando todo o conteúdo do arquivo. Isso permite a deteção de qualquer alteração no conteúdo original por meio da comparação de *hashs*.

Depois de criar a *hash*, o arquivo é enviado para o método `uploadFile` do *UploadService* (linhas 18-23), mencionado na Listagem 4.44. Em seguida, os dados do upload, incluindo o identificador do arquivo, o nome do arquivo e a *hash* do arquivo, são armazenados e retornados pelo método.

4.3.5 Verificar e *download* de um arquivo digital

Na Listagem 4.46, apresenta-se o segmento de código responsável por recuperar um arquivo da base de dados *MongoDB* para ser posteriormente baixado ou comparado com a *hash* original.

```
1  (...)
2  async getFileStream(fileId: string): Promise<NodeJS.ReadableStream>
3  {
4    const cursor = this.gridFs.find({});
5    for await (const doc of cursor) {
6      if (doc._id == fileId)
7      {
8        const downloadStream = this.gridFs.openDownloadStream(new mongoose.Types.
             ObjectId(fileId));
9        return downloadStream;
10     }
11   }
12   throw new NotFoundException('File not found');
13 }
14 (...)
```

Listagem 4.46: Download de arquivos (*upload.service.ts*)

Inicialmente, é feita uma pesquisa pelo arquivo na base de dados (linha 4). Se o arquivo for encontrado, é realizado o *download* utilizando o método `openDownloadStream` do `GridFs` (`this.gridFs .openDownloadStream()`, linhas 8-9) e fornecendo o identificador único do documento, que foi gerado durante o processo de *upload*. No caso em que o arquivo não exista na base de dados, é retornada a mensagem `'File not found'` (linha 12).

Na Listagem 4.47, encontramos dois *endpoints*. O primeiro, chamado `getFile` (linhas 2-7), é responsável por recuperar o arquivo e enviá-lo de volta ao utilizador para visualização. Esse *endpoint* é protegido por autenticação para garantir que apenas utilizadores autorizados possam recuperar os arquivos. O segundo *endpoint*, chamado `verifyFile` (linha 9-28), tem a função de verificar a integridade do arquivo em relação à sua *hash* original. Diferentemente do primeiro *endpoint*, o `verifyFile` não requer autenticação, permitindo que qualquer pessoa verifique a autenticidade de um arquivo, desde que possua as informações necessárias, ou seja, o identificador único do arquivo na base de dados e a *hash* contra a qual deseja verificar.

```
1  (...)
2  @UseGuards(AuthGuard('eth-jwt'))
3  @Get(':fileId')
4  async getFile(@Param('fileId') fileId: string, @Res() res: Response) {
5    const fileStream = await this.uploadService.getFileStream(fileId);
6    fileStream.pipe(res);
7  }
8
9  @Post(':fileId')
10 async verifyFile(
11   @Param('fileId') fileId: string,
12   @Res() res: Response,
13   @Request() req,
14   @Body() body: any,
15 ) {
16   try {
17     const fileStream = await this.uploadService.getFileStream(fileId);
18     const fileHash = await calculateSHA256Hash(fileStream);
19     console.log(fileHash)
20     if (fileHash === body.fileHash) {
21       return res.status(200).json({ isVerified: true });
22     }
23     return res.status(200).json({ isVerified: false });
24   } catch (ex) {
25     console.error('Error:', ex);
26     return res.status(200).json({ isVerified: false });
27   }
28 }
29
30 async function calculateSHA256Hash(stream): Promise<string> {
31   return new Promise<string>((resolve, reject) => {
32     const hash = crypto.createHash('sha256');
33     let data = '';
34     stream.on('data', (chunk) => {
35       data += chunk;
36     });
37
38     stream.on('end', () => {
39       const fileHash = hash.update(data).digest('hex');
40       resolve(fileHash);
41     });
42
43     stream.on('error', (error) => {
44       reject(error);
45     });
46   });
47 }
```

Listagem 4.47: Endpoints download e verificação de arquivos (*upload.controller.ts*)

No *endpoint* de verificação, o procedimento começa com a recuperação do arquivo (linha 17). Em seguida, é utilizado o método `calculateSHA256Hash` (Listagem 4.47, linha 30) para calcular novamente a *hash* a partir dos dados obtidos durante a recuperação do arquivo. Após a geração da *hash*, ela é comparada com a *hash* fornecida no corpo da solicitação do *endpoint* (linhas 20-27). Se ambas forem iguais, o *endpoint* retornará `{ isVerified: true }`. Caso contrário, o retorno será `{ isVerified: false }`.

4.4 *Monitoring Trigger*

Apesar de estar incluído no *back-end*, optou-se por dedicar uma secção separada ao assunto da monitorização da base de dados. Como o armazenamento de arquivos desempenha um papel crucial nesse desenvolvimento, e a *blockchain Ethereum* não suporta o armazenamento de arquivos de grande tamanho, chegamos à conclusão de que é fundamental ter um meio de lidar com eventos indesejados na base de dados *MongoDB*.

Para alcançar esse objetivo, foi decidido utilizar *triggers* para monitorar os eventos que ocorrem na coleção do *MongoDB* usada no sistema. Embora a abordagem ideal fosse criar uma aplicação separada

do *back-end*, devido a considerações de custo de hospedagem, optamos por integrar o *trigger* no mesmo projeto do *back-end*. A seguir, será apresentada a implementação realizada.

Na Listagem 4.48, apresenta-se o código do *trigger* responsável por monitorizar a base de dados e a coleção onde os arquivos armazenados estão guardados. Este serviço é chamado de *ChangeStreamService* devido à utilização do *change stream* fornecido pelo pacote *Mongoose*.

```
1  import { Injectable } from '@nestjs/common';
2  import { InjectConnection , InjectModel } from '@nestjs/mongoose';
3  import { Connection, Model } from 'mongoose';
4  import { ChangeData } from './Schema/changes.schema';
5
6  @Injectable()
7  export class ChangeStreamService {
8    private collection: any;
9
10   constructor(@InjectModel(ChangeData.name)
11   private model: Model<ChangeData>, @InjectConnection() private readonly connection:
          Connection) {
12     this.collection = this.connection.db.collection('fs.chunks')
13     this.initChangeStream();
14   }
15
16   private async initChangeStream() {
17
18     const changeStream = this.collection.watch();
19
20     changeStream.on('change', (change) => {
21       const createdData = new this.model(change);
22       createdData.save();
23     });
24   }
25 }
```

Listagem 4.48: Monitoring Trigger

No construtor, é utilizado o decorador `@InjectModel(ChangeData.name)`, que é empregado com injeção de dependência. Ele injeta o modelo *Mongoose* correspondente ao esquema *ChangeData* na classe, tornando-o disponível como a propriedade *model*.

A linha 12 , `this.collection = this.connection.db.collection('fs.chunks')` atribui a coleção do *MongoDB* chamada `'fs.chunks'` (onde os arquivos são armazenados) à propriedade *collection* da classe. Por fim, a linha `this.initChangeStream();` chama o método *initChangeStream*.

No método `initChangeStream`, a operação `this.collection.watch();` (linha 18), cria um fluxo de alterações para a coleção, permitindo que a aplicação escute as alterações que ocorrem nessa coleção. A linha `changeStream.on('change', (change) => { ... })` configura um *event handler* para o evento `'change'` emitido pelo fluxo de alterações. Quando uma alteração ocorre na coleção, a função de retorno de chamada fornecida é executada. Nessa função de retorno, uma instância do modelo *Mongoose ChangeData* é criada e, em seguida, é gravada na base de dados.

4.5 *Front-End*

O desenvolvimento do *front-end* foi realizado com a *framework ReactJS* (Secção 3.4.1). Para iniciar o projeto do *front-end*, utilizou-se o comando `npx create-react-app client` na raiz do projeto, que já havia sido configurado anteriormente com a *framework Truffle*, conforme detalhado anteriormente. Esse comando criou um novo projeto *ReactJS* na pasta `.\client`, a partir da qual se iniciou a implementação do *front-end*.

4.5.1 Ligação com *smart contracts* e *MetaMask*

Para a ligação do *front-end* com o *smart contract*, foi utilizada a biblioteca *Drizzle*. O *Drizzle* simplifica a interação com *Ethereum smart contracts* no contexto de uma aplicação *React*. Inicialmente, foi

necessário configurar a biblioteca *Drizzle* para que fosse possível utilizá-la.

Drizzle Options

```
1  import Web3 from 'web3';
2  import EvidenceChain from "../contracts/EvidenceChain.json";
3
4  const web3 = new Web3(window.ethereum);
5
6  const options = {
7     web3:{
8       customProvider: web3,
9     },
10   contracts: [EvidenceChain],
11   events: {
12     EvidenceChain: ["TransferOwnership", "ParticipantAdded", "EvidenceAdded" ],
13   },
14 };
15
16 export default options;
```

Listagem 4.49: Drizzle Options

Na Listagem 4.49, temos o código das opções que serão utilizadas na biblioteca *Drizzle*, assim como no servidor, onde usamos o *web3*. No cliente também é necessário utilizá-lo, mas apenas para instanciar o *Drizzle*. Na linha 4, está a ser instanciado o *Web3* com o `window.ethereum`. O `window.ethereum` faz parte da conexão do nosso *front-end* com o *MetaMask*. O *MetaMask*, sendo uma extensão, injeta o *Web3* na janela (*window*), permitindo uma conexão simples entre o *MetaMask* e o cliente *front-end*.

Após a instanciação do *web3*, a constante `options` foi criada, necessitando dos seguintes parâmetros: `web3: { customProvider: }`, `contracts: []`, e `events: {}` caso existam. O `import EvidenceChain` da linha 2, é o arquivo gerado no momento do *deploy* do *smart contract* na *blockchain*, fornecendo todas as informações sobre o contrato.

Drizzle Config

```
1  (...)
2  import { DrizzleContext } from "@drizzle/react-plugin";
3  import { Drizzle } from "@drizzle/store";
4  import drizzleOptions from "./configs/drizzleOptions";
5
6  const drizzle = new Drizzle(drizzleOptions);
7
8  function App() {
9    const [drizzleReadinessState, setDrizzleReadinessState] = useState({ drizzleState:
         null, loading: true });
10   const [tokenData, setTokenState] = useState();
11     (...)
12   return (
13     <DrizzleContext.Provider drizzle={drizzle}>
14       <DrizzleContext.Consumer>
15         {(drizzleContext) => {
16           const { drizzle, drizzleState } = drizzleContext;
17
18           if (drizzleReadinessState.loading) {
19             return <Loader />;
20           }
21
22           return (
23             <Router>
24               <MyNavBar drizzleContext={drizzleContext} token = {tokenData}/>
25               <Routes>
26                 <Route exact path="/" element={<Home drizzleContext={drizzleContext}
                       />} />
27                 <Route path="/entities" element={<Entities drizzleContext={
                       drizzleContext} />} />
28                 <Route path="/evidences" element={<Evidences drizzleContext={
                       drizzleContext} />} />
29               </Routes>
30             </Router>
31           );
32         }}
33       </DrizzleContext.Consumer>
34     </DrizzleContext.Provider>
35   );
36 }
```

Listagem 4.50: Drizzle Config

Na Listagem 4.50, apresenta-se o segmento de código responsável por instanciar o *Drizzle* para uso posterior na aplicação. Na linha 4, encontra-se o *import* das *options* mencionadas na Listagem 4.49. Após o *import* das *options*, na linha 6, é realizado o *instanciamento* do *Drizzle* com `new Drizzle (drizzleOptions)`.

Em seguida, foi definido o contexto do *Drizzle* com `<DrizzleContext.Provider drizzle={drizzle}`, o que permite compartilhar o *Drizzle* com todos os componentes que estão dentro deste contexto. É por isso que foi adicionado na raiz da aplicação. O `<DrizzleContext.Consumer>` é um componente *React* que permite consumir o contexto do *Drizzle* que foi definido anteriormente. Também é possível aceder o contexto do *Drizzle* se o mesmo for passado para os componentes subsequentes (por exemplo, linha 27).

Obtenção de registos com o *Drizzle*

```
1  import React, { useState } from 'react';
2  import { newContextComponents } from "@drizzle/react-components";
3  import EvidencesTable from '../components/Evidences/EvidencesTable';
4  (...)
5  const {ContractData } = newContextComponents;
6
7  function Evidences(props) {
8    const { drizzleContext } = props;
9    const { drizzle, drizzleState } = drizzleContext;
10
11 (...)
12   return (
```

```
13      <Container fluid className="p-3">
14       (...)
15        <ContractData
16          drizzle={drizzle}
17          drizzleState={drizzleState}
18          contract= {drizzle.contracts.EvidenceChain.contractName}
19          method="getEvidences"
20          render={(evidencesData, loading) => {
21            if (!loading && evidencesData) {
22              return <EvidencesTable EvidencesData = {evidencesData} drizzleContext =
                    {drizzleContext} />;
23            }
24          }}
25        />
26      </Container>
27    );
28  }
29  (...)
```

Listagem 4.51: Obter a lista de evidências com Drizzle

Na Listagem 4.51, apresenta-se o segmento de código responsável por obter a lista de evidências do *Ethereum smart contract* usando um componente da biblioteca *Drizzle*, o `ContractData`. Esse componente recebe o *drizzle* e o *drizzleState*, que são fornecidos pelo contexto do *Drizzle*. Além disso, ele recebe o parâmetro *contract*, que é o nome do contrato a ser chamado, o *method*, que é o método implementado no contrato (neste caso, está a ser passado `getEvidences` conforme descrito na secção 4.2.1, Listagem 4.17), e finalmente, ele recebe a função *render*, que será executada após a obtenção do resultado da chamada ao contrato.

Neste caso, após o contrato retornar a lista de evidências, a função *render* monta uma tabela para apresentar essas evidências (como ilustrado na secção 4.5.4).

Enviar uma transação com o *Drizzle*

Na Listagem 4.52, apresenta-se o segmento de código responsável por submeter um registo de evidência para o *Ethereum smart contract*, ou seja, enviar uma transação para o método do *smart contract* responsável por validar e adicionar um registo de evidência.

```
1   const handleSubmit = async (event) => {
2       (...)
3       if (form.checkValidity()) {
4         const contractInstance = drizzle.contracts.EvidenceChain;
5         const methodArgs = [
6           formData.evidenceUniqueCode,
7           formData.evidenceCaseNo,
8           selectedClassification,
9           formData.evidenceName,
10          selectedType
11        ];
12        const stackId = contractInstance.methods.addEvidence.cacheSend(
13          ...methodArgs
14        );
15        (...)
16      }
17      (...)
18    };
```

Listagem 4.52: Enviar uma transação com Drizzle

Primeiro, é realizada a validação do formulário com base nas regras de validação HTML5 (linha 3). Após a validação bem-sucedida do formulário, foi efetuada a obtenção da referência à instância do contrato inteligente (linha 4). Em seguida, definiu-se um *array* contendo os argumentos a serem enviados para o método. Após isso, é invocado o método `addEvidence` do *smart contract*, armazenando em *cache* a transação a ser enviada (linha 6). O *"stackId"* retornado pode ser usado para acompanhar o status da transação.

Este código mostra como foi implementado o envio de uma transação para o *ethereum smart contract* utilizado o *Drizzle*.

4.5.2 Ligação com *back-end*

Com finalidade de mostrar a forma como foi elaborada a ligação entre o *front-end* e o *back-end*, será apresentado na Listagem 4.53 o segmento de código responsável pelo upload de um arquivo digital.

Adicionar um arquivo digital

Na Listagem 4.53, encontra-se o segmento de código responsável por fazer a ligação ao *back-end*, enviar arquivos para serem armazenados e enviar uma transação para o *smart contract* para cada arquivo que está a ser enviado para armazenamento.

```
1  (...)
2  const url = consts.API_URL + `files/upload`;
3  try {
4      const response = await axios.post(url, formDataReq, {
5          headers: {
6              'Authorization': `Bearer ${localStorage.getItem("token")}`,
7              'Content-Type': 'multipart/form-data',
8          }
9      });
10
11     if (response.data.uploadedFileIds){
12       response.data.uploadedFileIds.forEach(element => {
13       console.log(element)
14        const contractInstance = drizzle.contracts.EvidenceChain;
15        const methodArgs = [formData.evidenceUniqueCode, element.id, element.filename
             , element.filehash];
16        const stackId = contractInstance.methods.addFile.cacheSend(...methodArgs)
17     });
18   }
19 } catch (error) {
20   (...)
21 }
```

Listagem 4.53: Upload de arquivos digitais

Na linha 2, esta a ser definido o *URL* do *endpoint* a ser invocado. Como mencionado na secção 3.4.1, a conexão com o *back-end* foi realizada utilizando a biblioteca *Axios*. Na linha 4, é feita o pedido *POST*, passando o URL, os arquivos (*formDataReq*) e as opções de cabeçalho (*headers*) da solicitação, que incluem o *token* de autorização e o tipo de conteúdo (*Content-Type*).

Após o pedido ser feito e a resposta ter sido obtida, verifica-se se a resposta contém os dados sobre os arquivos adicionados. Se eles existirem, para cada arquivo, é enviada uma transação para o *smart contract* *EvidenceChain*, para o método `addFile`. Todo o processo é realizado dentro de um bloco *try-catch* para que, em caso de erro em qualquer um dos processos, ele possa ser tratado no bloco *catch*.

4.5.3 Utilização do *MetaMask*

Nesta secção será apresentado o funcionamento do *MetaMask* com o *front-end* desenvolvido. Como mencionado na secção 3.4.1, o *MetaMask* é uma carteira de criptomoedas e uma extensão para o navegador que permite uma ligação e interação simples e fluida com aplicações descentralizadas. Na Figura 4.2 está representado a forma como funciona o *MetaMask* ao utilizar duas contas diferentes.

Contexto da conta na aplicação

Nas Figuras 4.2 e 4.3, está representada uma das páginas desenvolvidas na solução, que é a lista de utilizadores registados no sistema. Quando um utilizador registado acede a aplicação e visita a lista de entidades, o seu registo é destacado em verde para indicar que esse é o seu registo. A extensão que se sobrepõe à pagina *web* é a extensão do *MetaMask*. É possível analisar, na Figura 4.2, que a conta utilizada na extensão começa com 0xFE2 e termina com C3c2, sendo exatamente o mesmo endereço que está representado no canto superior direito da visualização.

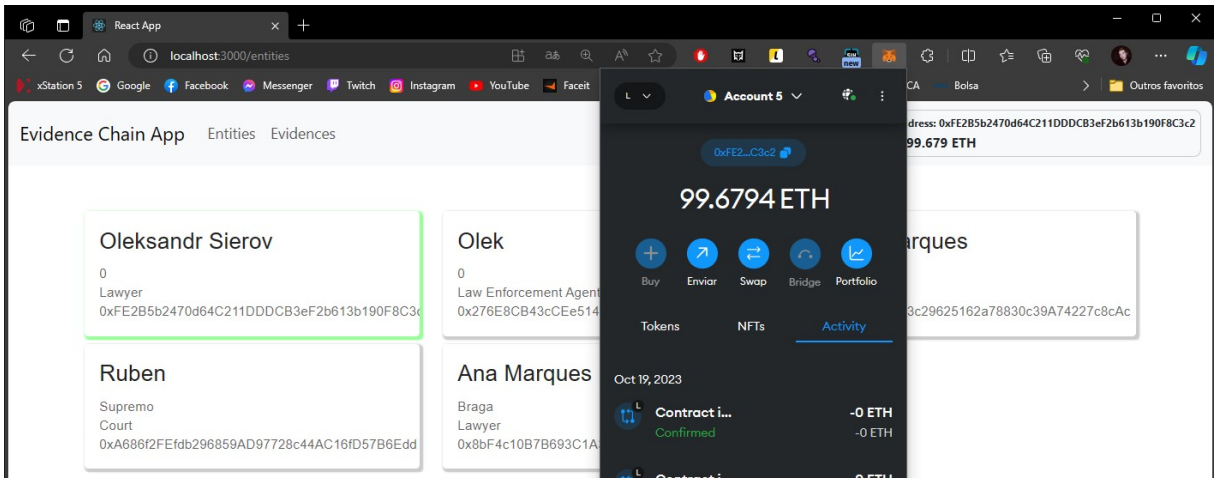


Figura 4.2: a) *MetaMask*: Contexto da conta na aplicação

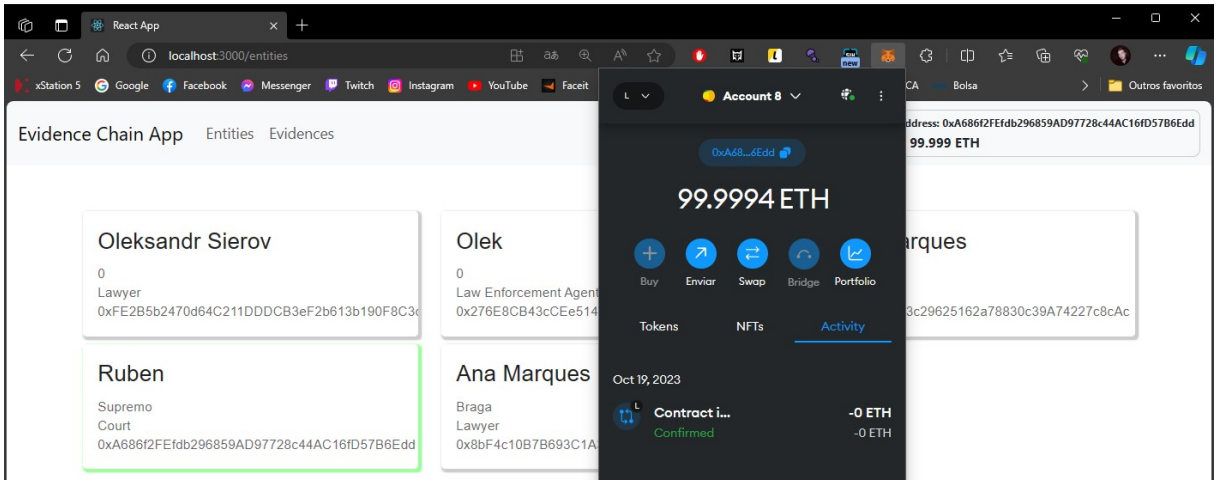


Figura 4.3: b) *MetaMask*: Contexto da conta na aplicação

Ao observar a Figura 4.3, é notável que a conta foi alterada (de *"Account 5"* para *"Account 8"*), indicando que outra conta de utilizador está a ser utilizada agora. Assim que a conta do *MetaMask* muda, a pagina da aplicação é atualizada. Na lista de entidades, a conta logada (destacada em verde) é diferente da Figura 4.2. Além disso, o endereço da conta usada no *MetaMask* corresponde ao que é exibido no canto superior direito da Figura 4.3 e ao endereço do registo destacado em verde.

Aprovar transações

Na Figura 4.4, está representada uma notificação do *MetaMask* solicitando a aprovação de uma transação pelo utilizador. Isso adiciona uma camada adicional de segurança à *dApp*, garantindo que nenhuma transação seja executada sem a aprovação explícita do utilizador. É possível ainda visualizar o custo estimado da transação em termos de *gas*, bem como o valor total em *ETH* necessário para a conclusão da transação. Na parte superior da notificação, é possível ver a direção da transação, ou seja, a transferência de fundos da conta do utilizador para um endereço que não está explicitamente identificado. Esse endereço, na verdade, é o endereço do *smart contract* na *blockchain*.

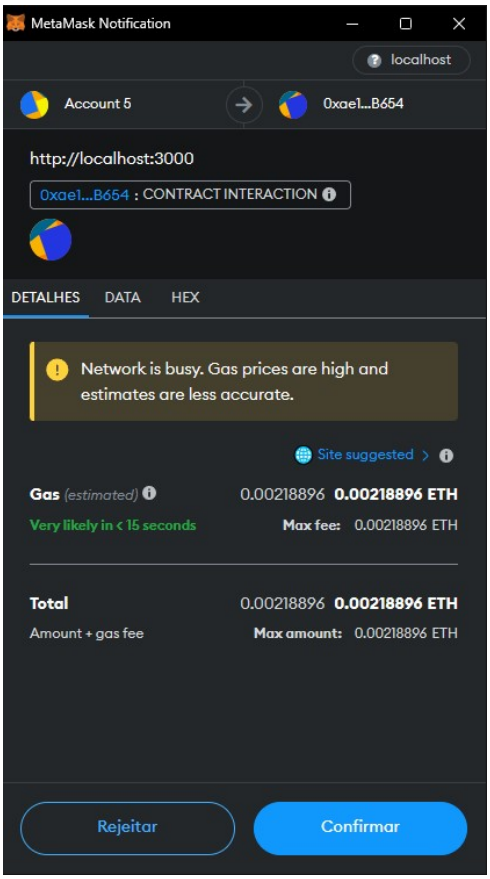


Figura 4.4: *MetaMask*: Notificação com pedido de aprovação

4.5.4 Aplicação *Web*

Para tornar a leitura mais intuitiva e facilitar a compreensão, nesta secção, optou-se por apresentar algumas das páginas *web* criadas durante o desenvolvimento. Essas páginas servem para interagir com os métodos implementados nos *smart contracts*.

Lista de registos de evidências

Na Figura 4.5, é possível visualizar uma lista de registos de evidência que o utilizador que tenha efetuado o *login* pode ver. Esses registos são apresentados em forma de tabela, e as colunas incluem o identificador do registo na blockchain (*ID*), o identificador do caso associado ao registo (*Case No*), a classificação do registo de evidência (*Classification*), o nome do registo (*Name*), o tipo de registo (*Type*), o responsável pelo registo (*Owner*) e as ações (*Actions*) possíveis de serem realizadas em relação a um determinado registo.

Além disso, é de notar que o utilizador tem permissão para executar várias ações em relação aos registos pelos quais ele é responsável. No entanto, em relação aos demais registos, o utilizador apenas pode visualizar o histórico, não podendo executar outras ações.

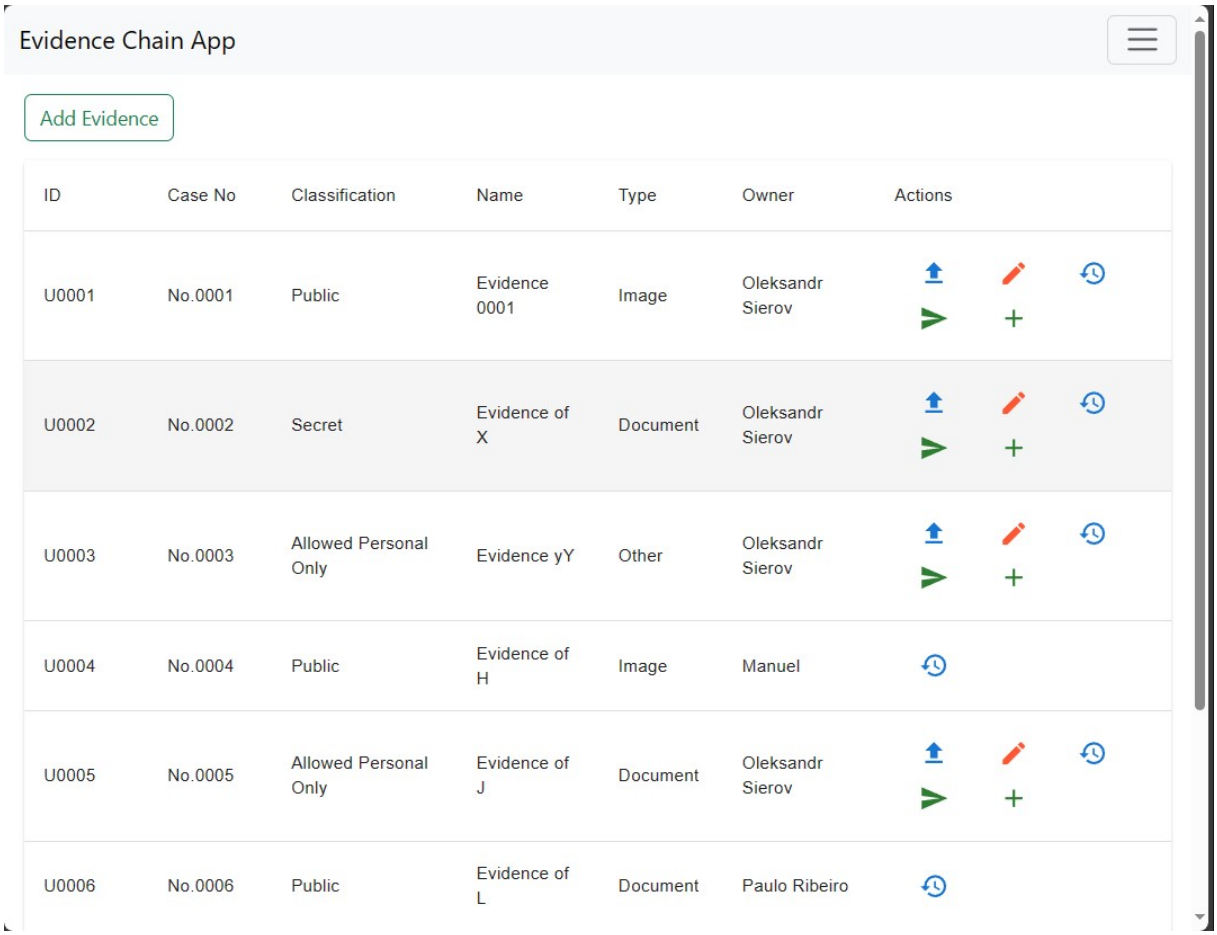


Figura 4.5: *Web*: Lista de registos de evidências

Adicionar um registo de evidência

Na Figura 4.6, é possível visualizar o *popup* que aparece quando o utilizador clica em *"Add Evidence"* na *view* anterior. Este *popup* é o formulário para adicionar um registo de evidência. Todos os campos são obrigatórios, e cabe ao utilizador preenchê-los corretamente antes de enviar. Os campos obrigatórios incluem o identificador do registo na blockchain (*ID*), o identificador do caso associado ao registo (*Case No*), a classificação do registo de evidência (*Classification*), o nome do registo (*Name*) e o tipo de registo (*Type*). Após preencher os campos e clicar em *"Submit"*, uma notificação do *MetaMask* aparecerá para que o utilizador confirme a transação, como ilustrado na Figura 4.4. Depois de confirmar a transação, o registo será adicionado à *blockchain*, e o utilizador que aprovou a transação será automaticamente definido o responsável (*owner*) do registo.

Adicionar um arquivo digital

Na Figura 4.7, é possível visualizar o *popup* que aparece quando o utilizador clica na ação de fazer *upload* de arquivos digitais num registo de evidência caso o mesmo tiver efetuado o *login* previamente, caso o utilizador não esteja *loggado*, irá inicialmente aparecer um popup para efetuar o *login*. O utilizador pode selecionar um ou vários arquivos digitais para *upload*, independentemente de sua extensão. Após selecionar os arquivos, o utilizador clica em *"Submit"*. Nesse momento, os arquivos selecionados são enviados para o servidor para serem armazenados na base de dados e, em seguida, são enviados para a blockchain com informações sobre eles, as informações são: o identificador no arquivo na base de dados, o nome do arquivo e a hash do arquivo. Durante o processo de envio para a blockchain, o utilizador receberá uma notificação do *MetaMask* para aprovar a transação. Após a aprovação, um novo registo será adicionado à *blockchain* no registo de evidência em questão.

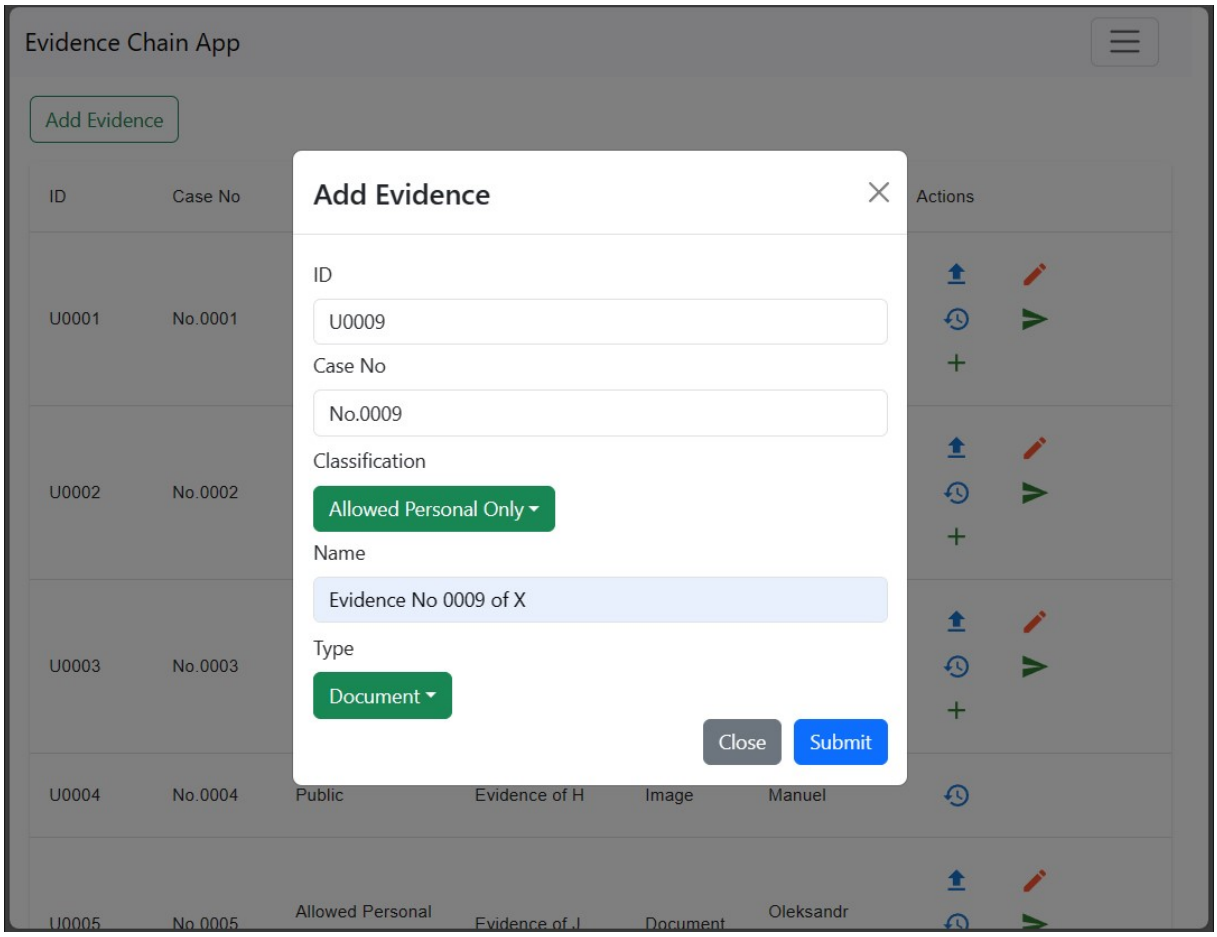


Figura 4.6: *Web*: Adicionar um registo de evidência

Adicionar um utilizador a um registo de evidência

Na Figura 4.8, é possível visualizar o *popup* que aparece quando o utilizador clica na ação de adicionar usuários com permissão para visualizar um registo de evidência. Neste *popup*, o utilizador pode adicionar um ou vários endereços públicos de entidades registadas no sistema para permitir que, as mesmas, visualizem um registo específico na lista de registos de evidências. Ao enviar a lista de endereços, o utilizador é novamente notificado pelo *MetaMask* devido à necessidade de confirmar a transação. Após a conclusão da transação, os utilizadores adicionados agora poderão visualizar o registo na lista de registos de evidências.

Transferir o *ownership* de um registo

Na Figura 4.9, é possível visualizar o *popup* que aparece quando o utilizador clica na ação de "enviar" uma evidência para outro utilizador, o que na prática significa transferir a responsabilidade (*ownership*) para outro utilizador. É possível ver que há apenas um campo a ser preenchido, que é o endereço público do utilizador para o qual será transferida a responsabilidade por um registo de evidência. Ao clicar em *"Submit"*, o utilizador receberá outra notificação do *MetaMask* para confirmar a transação. Após a transação ter sido concluída com sucesso, o utilizador agora somente terá a possibilidade de aceder ao histórico do registo da evidência que "enviou", o resto das ações não estarão mais disponíveis.

Histórico de uma evidência

Na Figura 4.10, podemos ver um *popup* com o histórico de um determinado registo de evidência. Cada etapa do processo pode ser relacionada às operações demonstradas nas figuras apresentadas anteriormente.

A Figura 4.6 (adicionar uma evidência) corresponde ao primeiro *card*. À esquerda, a data e a hora da

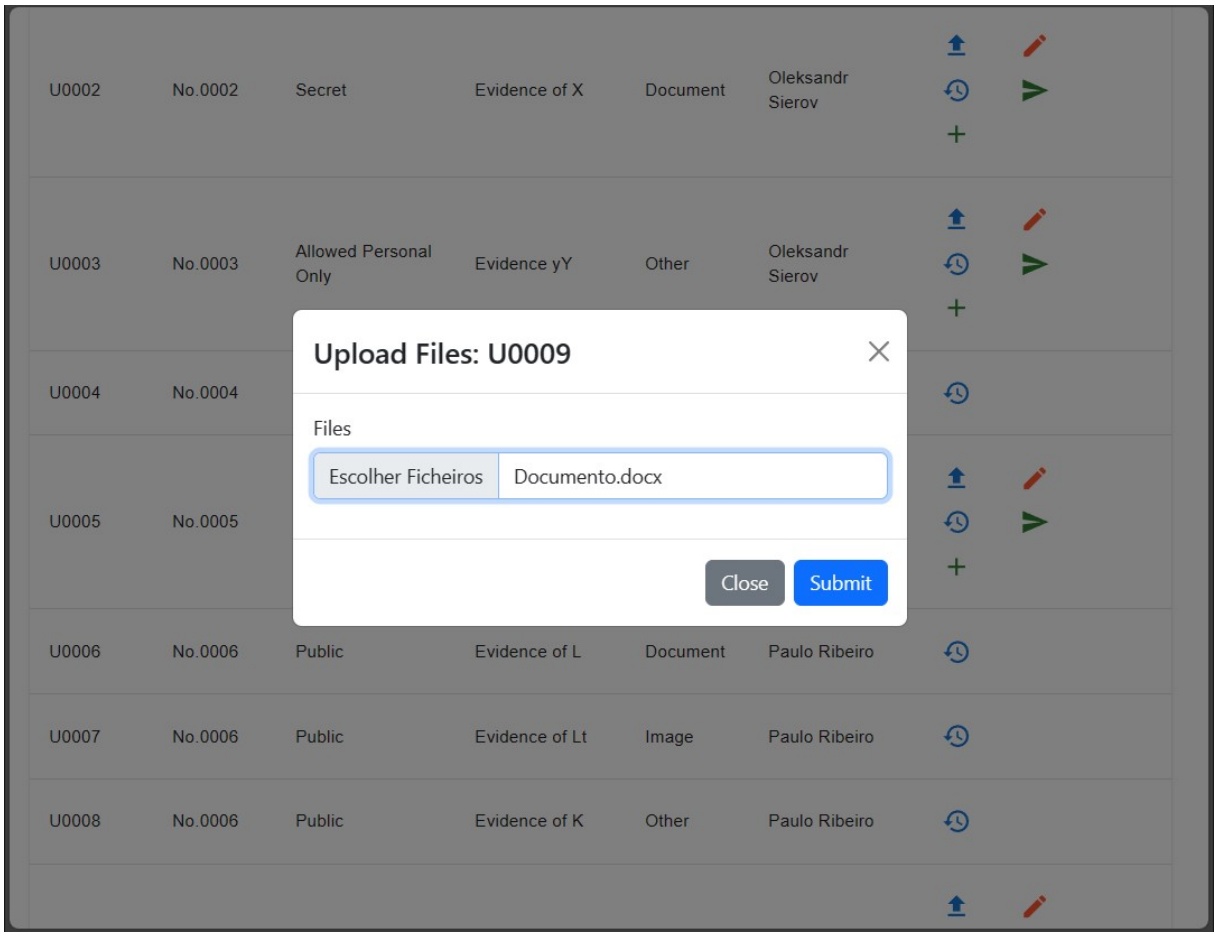


Figura 4.7: *Web*: Adicionar um arquivo digital

adição do registo à *blockchain* são exibidas, juntamente com uma observação, indicando que o registo foi criado. À direita, encontram-se as informações do registo, como nome, *Case No*, classificação, responsável pelo registo e tipo de registo de evidência.

A Figura 4.7 (adicionar um arquivo digital) corresponde ao segundo *card*. Neste *card*, podemos ver a data e hora exatas em que o arquivo digital foi adicionado, bem como a observação *"Added File"*. No lado direito, as informações são semelhantes às do *card* anterior, com a adição de uma linha que exibe detalhes do documento adicionado, incluindo a ação de download e um identificador de integridade. Sempre que o histórico é acedido, um pedido ao servidor é feito para validar a integridade do documento, o símbolo a verde apresenta que o ficheiro foi validado com sucesso, e a vermelho aparecerá caso o arquivo tenha sido manipulado.

A Figura 4.8 (adicionar um utilizador a um registo de evidência) corresponde ao terceiro *card*. A diferença em relação aos *cards* anteriores está no lado direito, onde podemos ver a data e hora do registo e as observações indicando a adição de um utilizador autorizado (*"Added Allowed User"*) com os dados do utilizador.

A Figura 4.9 (transferir o *ownership* de um registo) é o quarto e último *card* na Figura 4.10. Nas observações deste *card*, está indicado *"Updated Owner"*, o que significa que o utilizador responsável foi alterado. No *card*, à direita, em vez do *owner* anterior, agora vemos um utilizador diferente, que é o novo responsável pelo registo.

4.6 *Deploy* da *dApp*

Na secção 4.1, explorou-se a *Framework Truffle* e foram fornecidas as instruções detalhadas sobre como fazer o *deploy* de um *smart contract* numa *blockchain* privada. Foram discutidos os comandos

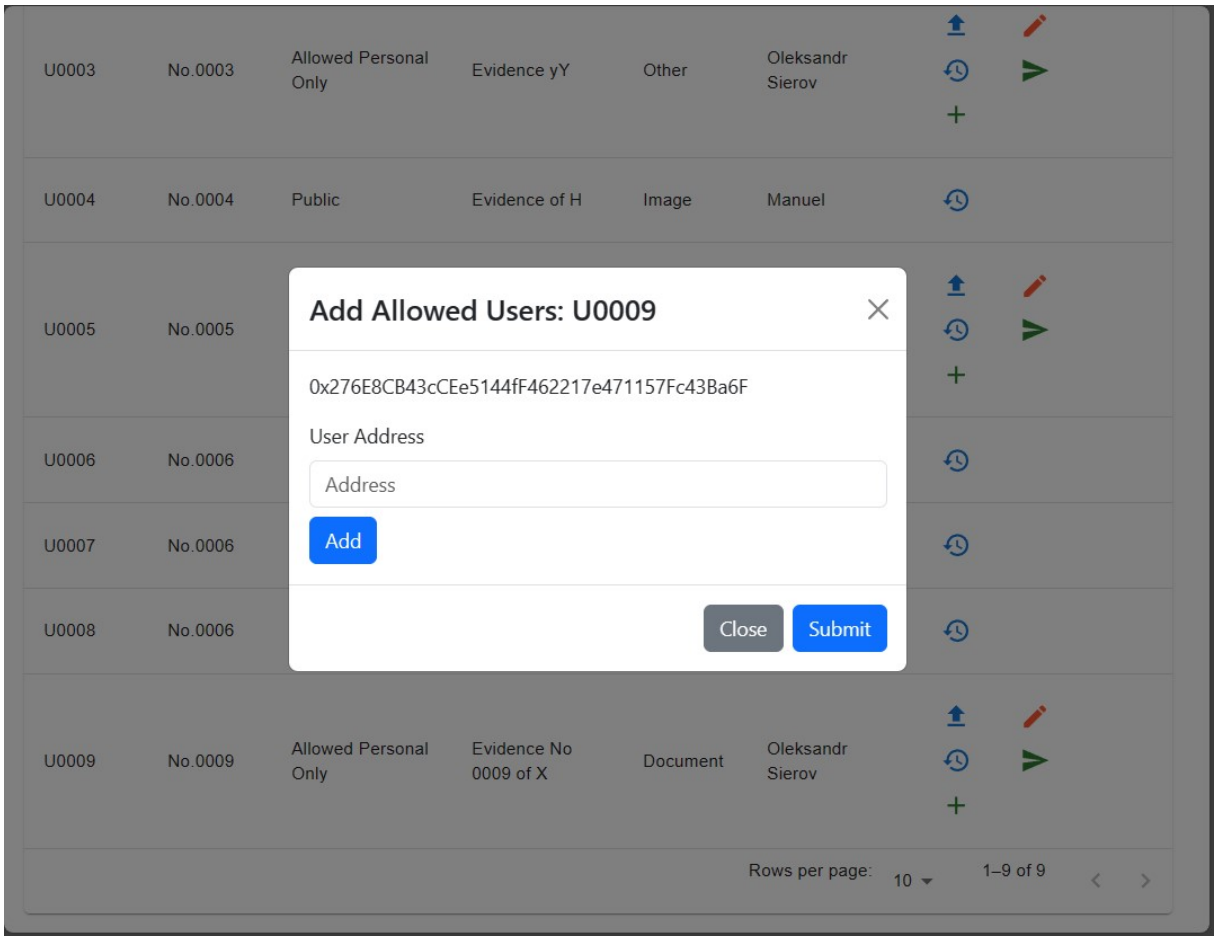


Figura 4.8: *Web*: Adicionar um utilizador a um registo de evidência

necessários e as configurações para uma conexão bem-sucedida com o *Ganache*, uma *blockchain* privada. Neste secção, será mostrado o *deploy*, via diagrama, do sistema para um ambiente não local.

4.6.1 Configuração com uma *blockchain testnet*

Antes de prosseguir com o *deploy* em si, é necessário configurar o *Truffle* para que este saiba em qual rede de teste o contrato será implantado. Até agora, apenas havíamos configurado a blockchain local, como discutido anteriormente (Secção 4.1).

Como ambiente de testes, será utilizado o *Sepolia*, o *Sepolia* é uma *blockchain ethereum* de testes, criada para os *developers* testarem os seus *smart contracts* antes de serem publicados para a *MAINNET*. Para conseguirmos, de uma forma simplista, conectar e efetuar o *deploy* dos *smart contracts* na *TESTNET Sepolia*, foi utilizado o *Infura*.

Infura

Infura é um serviço de infraestrutura e API que permite aceder a rede *Ethereum* sem precisar executar um nó completo da *Ethereum*. Ele atua como um *gateway* para a rede *Ethereum*, fornecendo uma maneira simples e confiável de se conectar à blockchain *Ethereum*. O *Infura* gere uma rede de nós *Ethereum* que lida com as tarefas de comunicação com a rede, validação de transações e coleta de dados, tornando mais fácil para os desenvolvedores interagirem com a blockchain *Ethereum* por meio de uma API simplificada (Infura, 2023).

Após aceder ao site oficial do *Infura*, procedeu se a criação de uma conta nova. Após a conta criada, foi necessário criar uma *"API KEY"*, isto será a nossa ponte entre a *TESTNET* e o *Truffle* para efetuar o *deploy* dos *smart contracts*. Após a *"API KEY"* ter sido criada, podemos listar as *Networks* disponíveis,

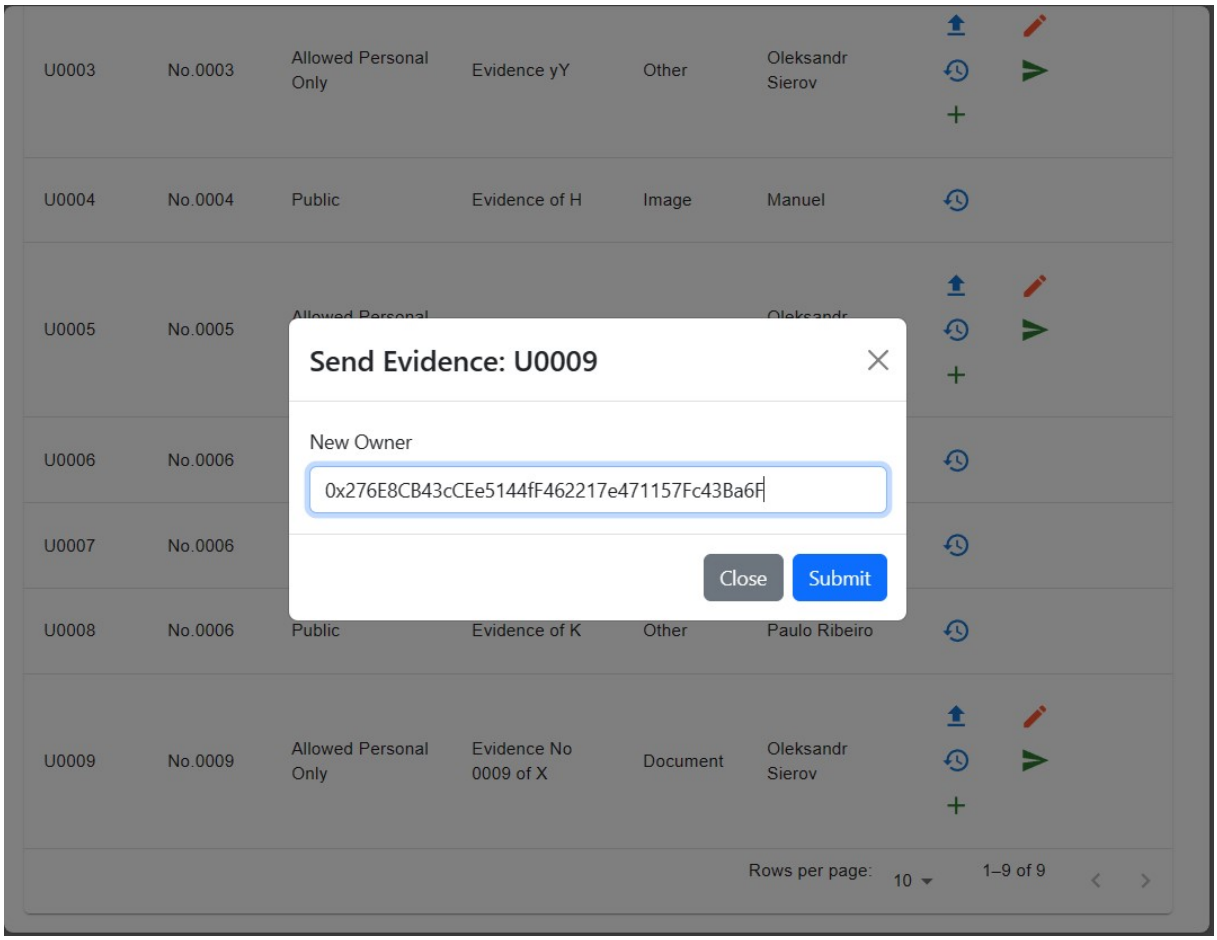


Figura 4.9: *Web*: Transferir o *ownership* de um registo

a que nos interessa é a *Network Ethereum*, nessa linha podemos selecionar o ambiente, por defeito estará selecionada a *MAINNET*, porem, nesta fase o que nos interessa é uma *TESTNET* e ai selecionamos a *Sepolia*. De seguida, copiamos o endereço fornecido e vamos até a raiz da solução.

No arquivo `truffle-config.js`, podemos configurar a *network* para o *sepolia*. Na Listagem 4.54, encontra se então a nova versão do `truffle-config.js` com a *network*:

```
require("dotenv").config();
const HDWalletProvider = require("@truffle/hdwallet-provider");
const { INFURA_API_KEY, MNEMONIC } = process.env;

networks: {
    development: {          // Local Connection
     host: "127.0.0.1",     // Localhost
     port: 7545,            // Standard Ethereum port
     network_id: "*",       // Any network
    },
    sepolia: {
      provider: () => new HDWalletProvider(MNEMONIC, INFURA_API_KEY),
      network_id: "11155111", // Sepolia's network ID
      gas: 6219800,
      from: "0x810bEFaf873c9109b6c42396f93E5E7F96369155",
     }
 },
 (...)
```

Listagem 4.54: Ficheiro de configuração do Truffle

Na Listagem 4.54 mostra a secção *"sepolia"* do *JavaScript* de configuração para a configuração do ambiente de testes de contratos inteligentes *Ethereum* na *testnet Sepolia*. Essa secção possibilita o *deployment* dos *smart contracts* na *testnet* e a utilização das funções implementadas, juntamente com a

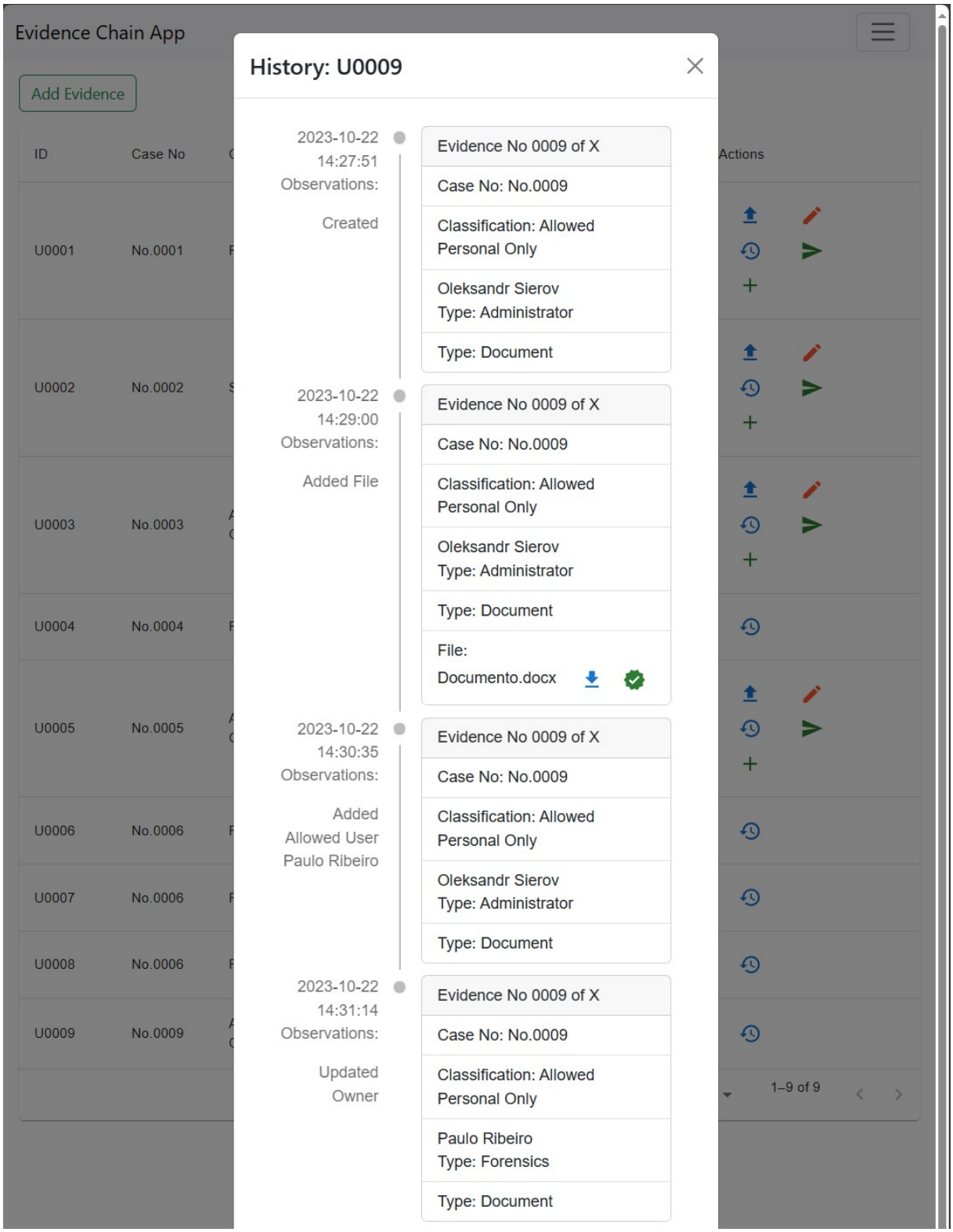


Figura 4.10: *Web*: Histórico de uma evidência

conexão do *MetaMask* à mesma rede. É importante notar que a secção `sepolia` difere consideravelmente da secção de `development`, pois requer novos dados para o *deployment*.

O `provider` é provedor que será utilizado para interagir com a *blockchain*. *HDWalletProvider* é uma classe frequentemente utilizada para conectar a conta *Ethereum* a uma rede (nesse caso, a rede *Sepolia*) com o uso de uma frase de recuperação (mnemónica) e a chave de *API* do *Infura*. O `network_id` é o

identificador da rede à qual queremos conectar, no caso o identificador da rede *Sepolia* é `"11155111"`. O `gas` é o limite de *gas* que será utilizado para as transações na rede. O `from` é o endereço publico *ethereum* do qual as transações serão enviadas.

4.6.2 *Deployment*

Na Figura 4.11, encontra se representado, de uma forma simplificada, o passo a passo efetuado para se colocar a solução desenvolvida num ambiente não local.

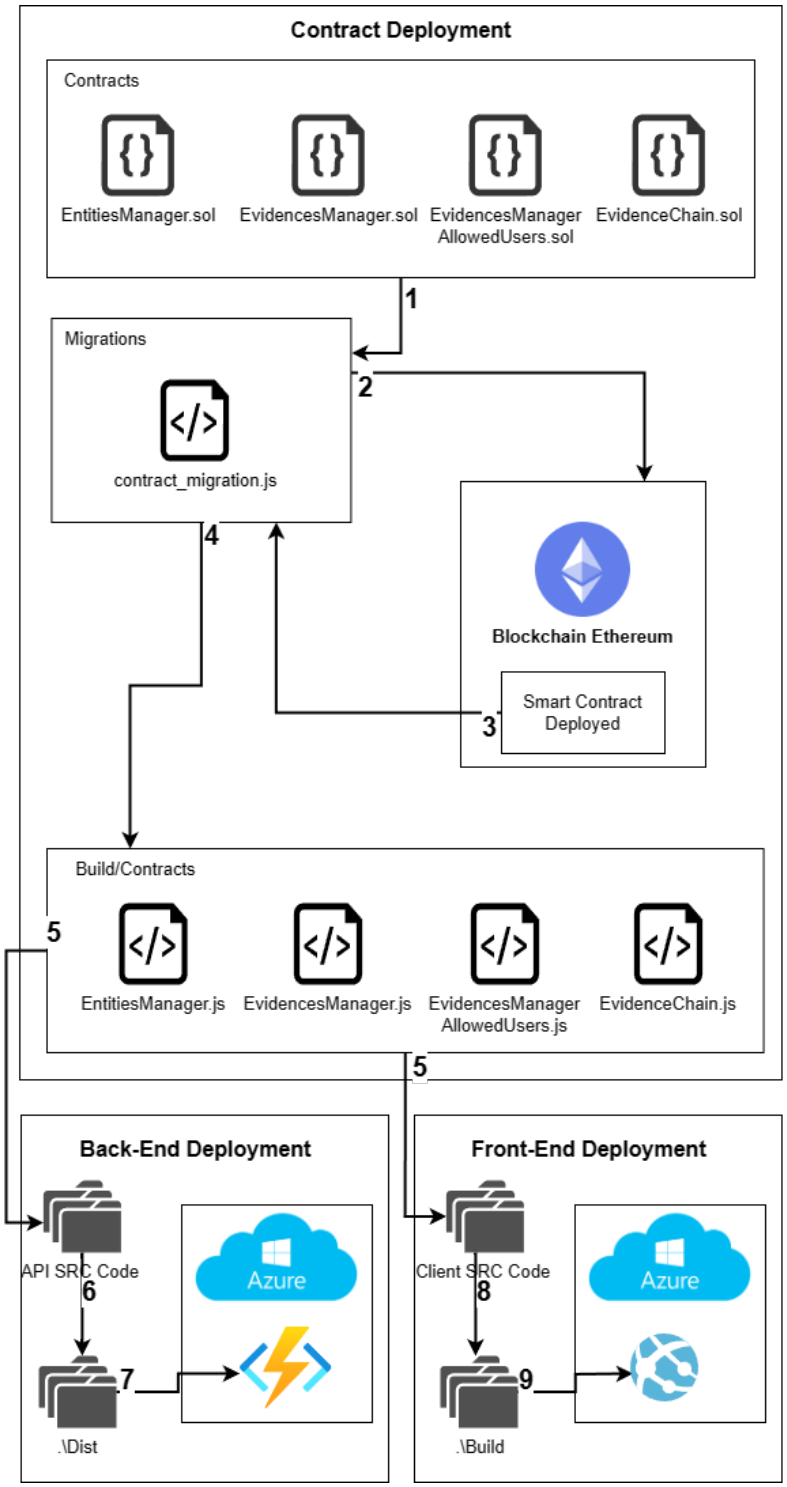


Figura 4.11: Step-by-Step: *Deployment* da *dApp*

De seguida, serão discutidos os passos de uma forma mais detalhada:

- Passo 1: Executar o comando `truffle migrate` na consola dentro do projeto. O código dos contratos inteligentes desenvolvidos é lido pelo arquivo *contract_ migrations.js*, que é responsável por implantar o código na *blockchain Ethereum*. Dado que alguns dos contratos inteligentes dependem uns dos outros, a ordem de implantação é crucial, uma vez que alguns contratos precisam fazer referência (ou seja, usar o endereço) a outros contratos. Por exemplo, o contrato *EvidenceChain* deve ser o último a ser implantado, uma vez que depende de todos os contratos anteriores.

```
1  const EvidenceChain = artifacts.require("./EvidenceChain.sol");
2  const EntitiesManager = artifacts.require("./EntitiesManager.sol")
3  const EvidencesManager = artifacts.require("./EvidencesManager.sol")
4  const EvidencesManagerAllowedUsers = artifacts.require("./
       EvidencesManagerAllowedUsers.sol")
5
6  module.exports = async function(deployer) {
7      await deployer.deploy(EntitiesManager);
8      const entitiesManagerInstance = await EntitiesManager.deployed();
9      await deployer.deploy(EvidencesManagerAllowedUsers);
10     const evidencesManagerAllowedUsersInstance = await
           EvidencesManagerAllowedUsers.deployed();
11     await deployer.deploy(EvidencesManager,
           evidencesManagerAllowedUsersInstance.address);
12     const evidencesManagerInstance = await EvidencesManager.deployed();
13     await deployer.deploy(EvidenceChain, entitiesManagerInstance.address,
           evidencesManagerInstance.address);
14 };
```

Listagem 4.55: Contract Migrations

- Passo 2 e 3: Na Listagem 4.55, é possível ver o conteúdo do arquivo *contract_ migrations.js*. Da linha 1 à linha 4, são adicionados os arquivos *.sol* dos contratos inteligentes implementados. Em seguida, para cada contrato, é feito o processo de *deploy*. A ordem de *deploy* é determinada pelas dependências entre os contratos. O *EntitiesManager.sol* não possui dependências em relação a nenhum dos contratos inteligentes, então ele é o primeiro a ser implantado. Em seguida, o *EvidencesManagerAllowedUsers.sol* é implantado, também não tendo dependências. Em terceiro lugar, é implantado o *EvidencesManager.sol*. Conforme pode ser observado na linha 11, ele requer o endereço do contrato *EvidencesManagerAllowedUsers.sol*, que foi implantado anteriormente, para funcionar corretamente (o endereço do contrato *EvidencesManagerAllowedUsers.sol* é necessário no construtor do contrato *EvidencesManager.sol*). Por fim, o *EvidenceChain.sol* é implantado, que requer os endereços dos contratos inteligentes *EntitiesManager.sol* e *EvidencesManager.sol*, pois somente dessa forma o contrato *EvidenceChain.sol* teria a possibilidade de comunicar com os contractos *EntitiesManager.sol* e *EvidencesManager.sol*.

- Passo 4: Após a conclusão do processo de migração, os arquivos *.js* relacionados aos contratos implementados estarão na pasta `.\build\contracts`. Esses arquivos contêm as informações do `ABI` dos contratos e os endereços correspondentes na rede.

- Passo 5: Com os arquivos *.js* obtidos para cada contrato, eles devem ser copiados para a raiz de cada projeto, tanto o *front-end* quanto o *back-end*. Isso é feito para que o *Drizzle* e o *MetaMask* tenham acesso aos dados necessários sobre os *smart contracts* no *front-end*, e para que o *back-end* possa estabelecer a conexão com os *smart contracts* por meio do *Web3*.

- Passo 6: No *back-end*, foi necessário instalar o pacote *Azure Functions* do *Nest* (*azure-func-nest*) para permitir o *deploy* da aplicação *Nest*. Após a instalação do pacote, o comando `npm run build` foi executado para gerar a pasta `.dist` com todos os arquivos necessários para o *deploy* do *backend*.

- Passo 7: Foi instalada a extensão do *Azure* no *Visual Studio Code*, e com essa extensão, realizou-se o *deploy* da pasta `.dist` no *Azure* como uma *Azure Function* (a extensão torna o processo de *deploy* de funções ou aplicações web simples e rápido).

- Passo 8: Foi utilizado o comando `npm run build` para gerar a pasta `.build` com todos os arquivos necessários para o *deploy* do *front-end*.

- Passo 9: Utilizando a mesma extensão instalada no Passo 7, foi realizado o *deploy* da pasta `.build` do *front-end* como um nova aplicação web no *Azure*.

Após a conclusão de todos esses passos, foi possível testar a implementação num ambiente completamente independente do ambiente local, tornando-a acessível a partir de qualquer local.

Como conclusão deste capítulo é possível resumir todo o processo da Implementação do Sistema da seguinte maneira:

- Foi criado o projeto *Truffle* juntamente com a ligação a uma blockchain privada *Ganache*.
- Foram desenvolvidos os *smart contracts* necessários para atenderem as necessidades definidas pelos casos de uso no capítulo anterior
- Foi implementado um *back-end* com ligação a uma base de dados *MongoDB* para armazenamento de arquivos digitais.
- Foi implementado um *Trigger* no *back-end* para uma maior segurança e controlo da base de dados *MongoDB*.
- Foi desenvolvida uma aplicação *WEB* para ser possivel a visualização e interação com todas as funcionalidades anteriormente mencionadas.
- Foi efetuada uma publicação de todos os componentes desenvolvidos para um ambiente não local.

5. Experimentação e Análise de Resultados

Para demonstrar a usabilidade e a rastreabilidade fornecida pela solução, optou-se por realizar uma experimentação comparativa. Nesta experimentação procurou-se comparar: i) a solução desenvolvida nesta investigação, registo e armazenamento de registos de evidências utilizando a plataforma *blockchain Ethereum*, com ii) a abordagem convencional em *Linux* utilizada por entidades jurídicas na recolha de evidências digitais.

A solução desenvolvida para esta dissertação reúne os seguintes pontos chave:

- Criar registos de evidências na *blockchain* de forma a garantir a sua integridade e rastreabilidade perante as alterações que fossem ocorrendo horizontalmente.
- Armazenar os documentos jurídicos/relatórios. O armazenamento dos documentos jurídicos e relatórios preenchidos pelas entidades jurídicas é essencial ao longo de todo o processo de tratamento de evidências, desde a recolha até a análise e armazenamento final. Conforme descrito no guia *INTERPOL Global guidelines for digital forensics laboratories* (INTERPOL, 2019), é imperativo documentar e armazenar cuidadosamente cada etapa do processo.
- Armazenar outros arquivos relevantes sobre uma evidência, como por exemplo fotos e/ou vídeos tirados no processo da recolha.
- Gerir o acesso aos registos de evidências, mantendo o registo dos utilizadores na *blockchain* e vinculando-os às atividades que realizam. Por exemplo, ao adicionar um registo, o utilizador que o cria será vinculado ao registo recém-criado.

A experimentação foi planeada da seguinte forma na plataforma *Linux*: i) levantamento de uma cópia de evidência digital usando ferramentas específicas fornecidas pelo sistema operacional *Linux*; ii) a cópia da evidência foi submetida a testes que envolveram tentativas de adulteração dos dados; iii) analisar o estado da cópia e aferir se as tentativas de adulteração podem ser rastreadas na sua plenitude. Para a plataforma *blockchain* desenvolvida neste trabalho o processo foi semelhante porém o ponto i) foi armazenar um arquivo digital no sistema.

5.1 Experimentação em *Linux*

A experimentação foi realizada utilizando o método que entendemos ser convencional empregado por cientistas forenses na recolha de evidências digitais. Para viabilizar essa abordagem, foram realizados preparativos prévios. O processo iniciou-se com a recolha de uma imagem de um dispositivo para preservar o seu conteúdo para as análises subsequentes, analise das hashes geradas, tentativas de adulteração do arquivo de imagem extraído, avaliação do sucesso ou fracasso dessas tentativas de adulteração, validação da hash final e, por fim, observação como o sistema convencional lida com possíveis tentativas de adulteração.

5.1.1 Plataforma tecnológica de suporte

Para a realização da experimentação, foi utilizado o sistema operativo *Linux*, especificamente a versão *Kali GNU/Linux Rolling*, com o *Kernel Linux 6.3.0-kali1-amd64*. O *Linux* foi executado num computador com recurso a máquina virtual *VMWare Workstation Player*, na versão 17.5.0, com o *Build Number: 22583795 & Release Date: 2023-10-19*.

Para obter as informações do sistema operativo que foi utilizado para a experimentação, foi utilizado o comando do *Linux* `hostnamectl` apresentado na Figura 5.1.

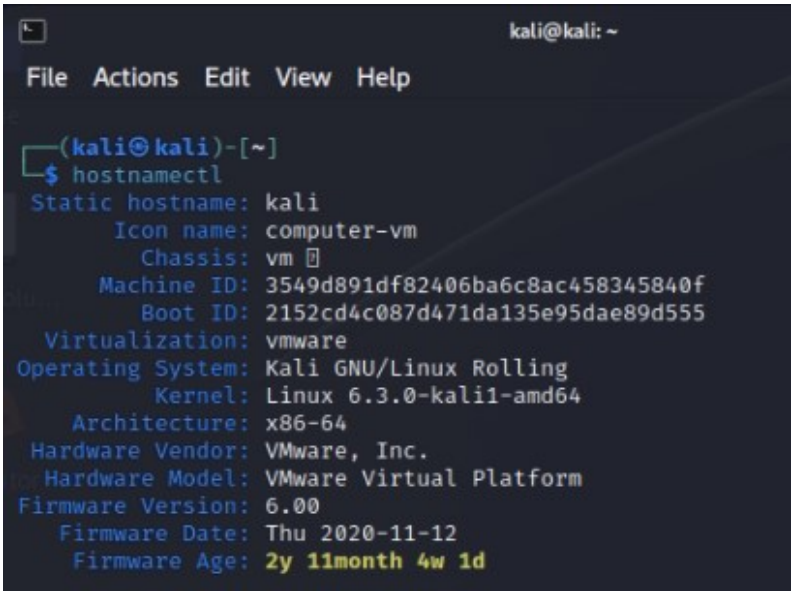


Figura 5.1: *Kali Linux Setup*

5.1.2 Ferramentas

Na plataforma *Linux* foram utilizadas um conjunto de ferramentas, nomeadamente:

- *Guymager*: é um gerador de imagens forenses para a aquisição de suportes de dados (gvoncken, 2023).
- *GtkHash*: é um utilitário de ambiente de trabalho concebido para calcular resumos de mensagens ou somas de verificação (Heaven et al., 2023).
- *losetup*: Ao contrario das ferramentas anteriores que contêm uma interface de utilizador, o *losetup*, é um comando do sistema operativo *Linux* que foi utilizado para montar a imagem recolhida com a ferramenta *Guymager*.

5.1.3 Processo

Nesta secção, descreve-se passo a passo a experimentação no ambiente *Linux*.

A) Instalação e preparação da máquina

A maquina que suportou a experimentação foi preparada com as seguinte fases:

1. Foi efetuada a instalação do *VMWare Workstation Player 17.5.0*;
2. Adicionou-se o sistema operativo *Kali* ao *VMWare*
3. Conforme referido, foi necessário preparar um dispositivo para possibilitar a recolha da sua imagem. O dispositivo escolhido foi um *USB PenDrive* de 4GB, contendo alguns arquivos utilizados na experiência.

B) Recolha da imagem

Para recolher a imagem do dispositivo *USB*, foi utilizada a ferramenta *Guymager*.

Na Figura 5.2, apresenta-se a interface do *Guymager*, juntamente com a indicação dos dispositivos disponíveis para a recolha da imagem. A primeira linha representa a máquina virtual, e a segunda linha corresponde ao dispositivo *USB*, com a capacidade de 4GB.

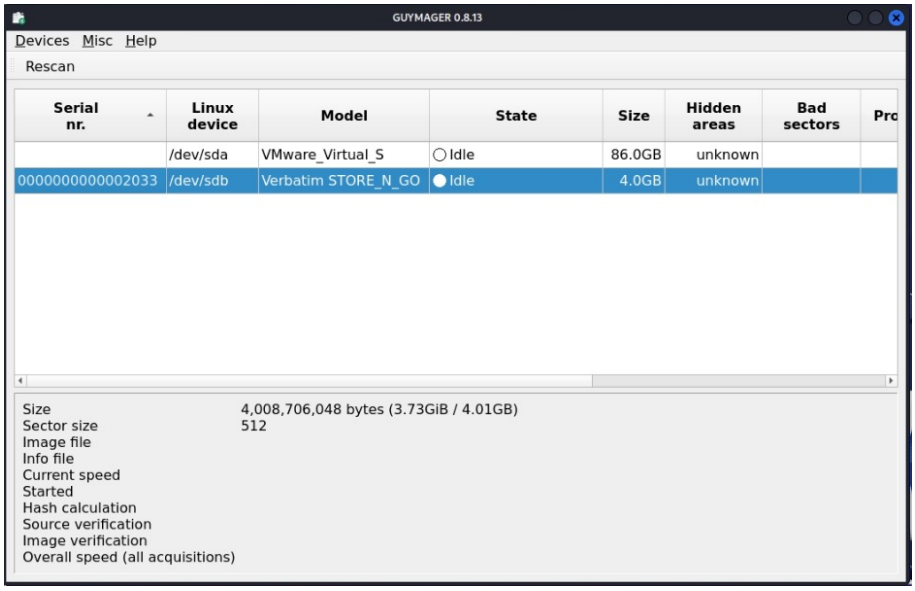


Figura 5.2: Interface *Guymager*

Optou-se por fazer o *"Acquire image"* de dispositivo *USB* (Figura 5.3).

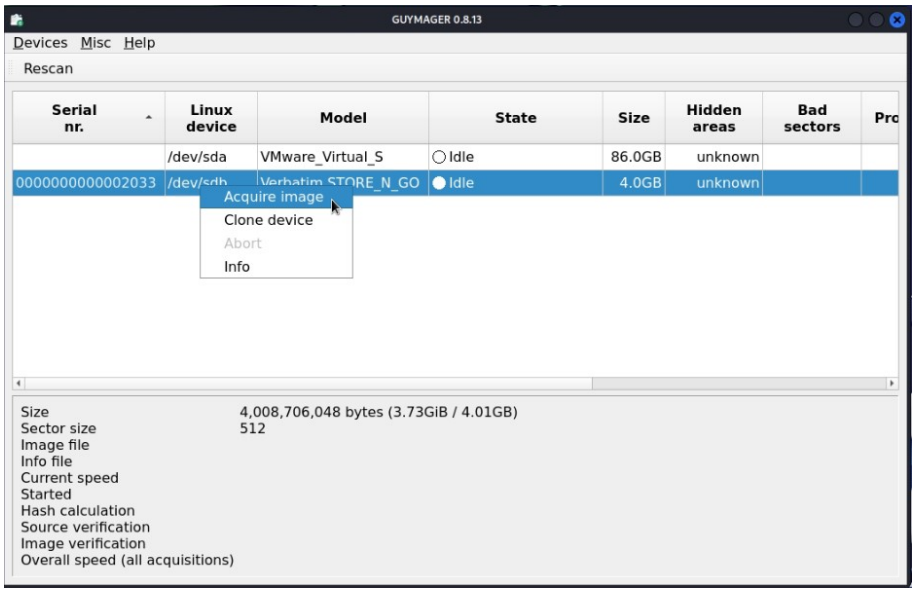


Figura 5.3: *Guymager acquire image*

Na Figura 5.4, pode-se visualizar a janela que é apresentada após selecionar a opção *"Acquire image"*. Aqui, temos a capacidade de escolher algumas opções para a recolha da imagem, tais como:

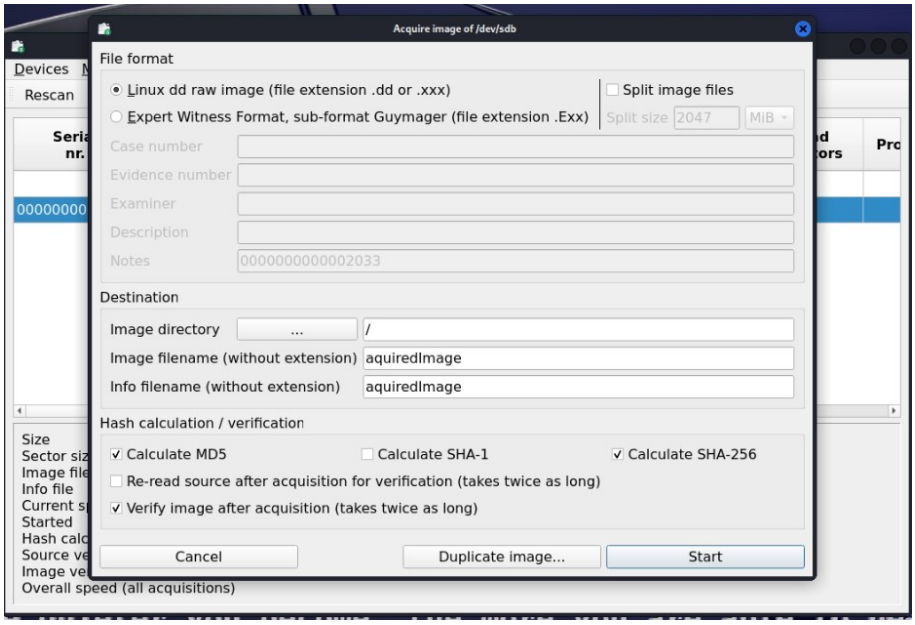


Figura 5.4: *Guymager acquire image options*

- *File format*: formato do arquivo a ser criado. O selecionado foi o formado *.dd raw file.*
- *Split image files*: permite definir se a imagem será dividida em partes de tamanho X ou não. Neste caso, optou-se por uma cópia inteira.
- *Destination*: indica o local onde a imagem será armazenada, e o nome do arquivo resultante. Escolheu-se armazenar o arquivo na raiz do sistema operativo com o nome *"acquiredImage"*.
- *Hash calculation/verification*: possibilita a seleção das hashes a serem calculadas. Optou-se por *MD5* e *SHA-256*, e foi marcada a opção para verificar as hashes ao final da recolha da imagem.

Com todas as opções e configurações realizadas, realizou-se o *"Start"* do processo da recolha da imagem (Figura 5.4).

Assim que na coluna do estado aparecer o estado *"Finished"* e o progresso indicar *100%* significa que o processo de recolha e verificação da imagem foi concluído com sucesso tal como demonstrado na Figura 5.5.

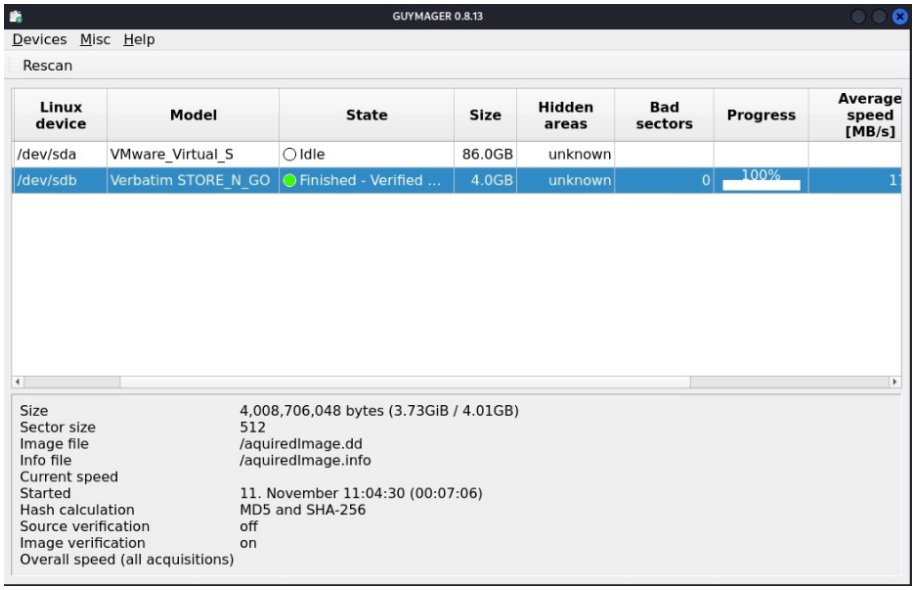


Figura 5.5: *Guymager acquire image* terminado

Na Figura 5.6, apresentam-se os arquivos na raiz do sistema operativo, onde são visíveis dois novos arquivos: o arquivo com a extensão *.dd*, que corresponde à imagem recolhida do nosso dispositivo físico, e o arquivo *.info*, que contém informações sobre a aquisição da imagem. Este último arquivo inclui informações detalhadas sobre o dispositivo do qual a imagem foi recolhida, bem como as *hashes* geradas no momento da recolha e as *hashes* geradas e validadas no final do processo de recolha, na listagem 5.1 é possível ver um excerto do conteúdo do arquivo *.info* gerado. Neste é possível ver os valores das *hashes* geradas no momento da criação (linhas 16 e 22) e as *hashes* gerada no momento da verificação da cópia criada (linhas 18 e 24), também é possível ver o tamanho do dispositivo e as opções que foram selecionadas no momento do *"Acquire image"* (linhas 5-11). Na linha 25, é visível o *output log* da verificação e por fim as linhas 27-28 são os instantes de quando ocorreram cada um dos processos.

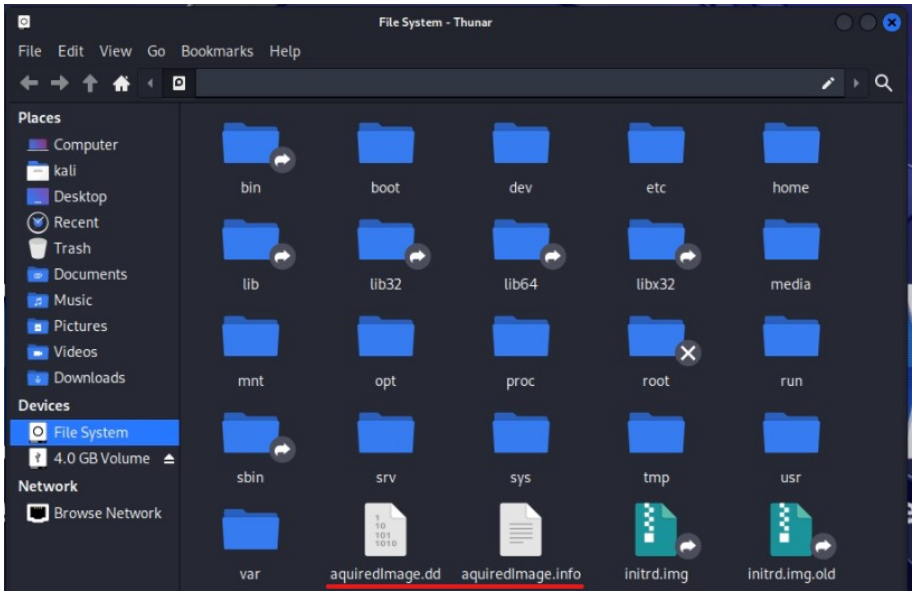


Figura 5.6: Ficheiros Criados pelo *Guymager*

```
Acquisition
===========

Linux device            : /dev/sdb
Device size             : 4008706048 (4.0GB)
Format                  : Linux dd raw image - file extension is .dd
Image path and file name: /aquiredImage.dd
Info  path and file name: /aquiredImage.info
Hash calculation        : MD5 and SHA-256
Source verification     : off
Image verification      : on

No bad sectors encountered during acquisition.
State: Finished successfully

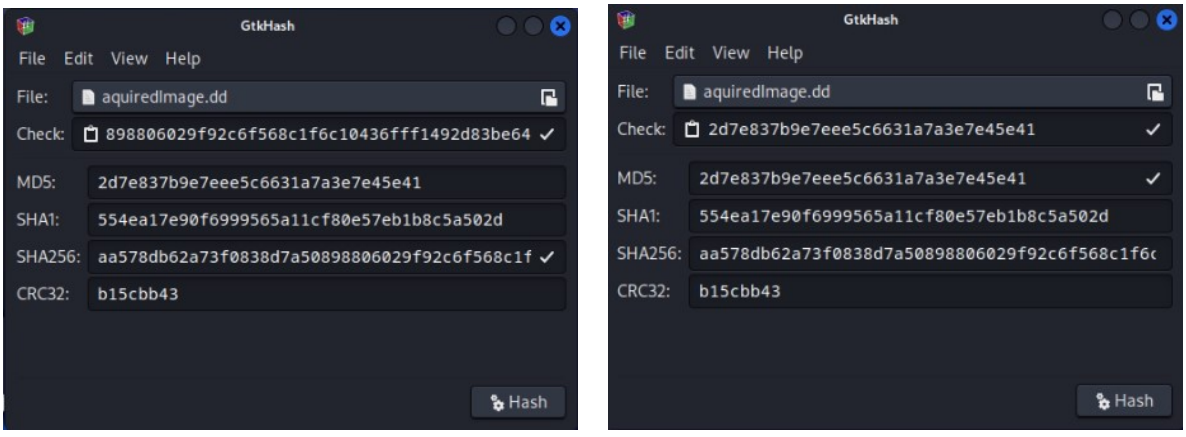
MD5 hash                    : 2d7e837b9e7eee5c6631a7a3e7e45e41
MD5 hash verified source    : --
MD5 hash verified image     : 2d7e837b9e7eee5c6631a7a3e7e45e41
SHA1 hash                   : --
SHA1 hash verified source   : --
SHA1 hash verified image    : --
SHA256 hash                 :
    aa578db62a73f0838d7a50898806029f92c6f568c1f6c10436fff1492d83be64
SHA256 hash verified source: --
SHA256 hash verified image :
    aa578db62a73f0838d7a50898806029f92c6f568c1f6c10436fff1492d83be64
Image verification OK. The image contains exactly the data that was written.

Acquisition started : 2023-11-11 11:04:30 (ISO format YYYY-MM-DD HH:MM:SS)
Verification started: 2023-11-11 11:11:07
Ended               : 2023-11-11 11:11:37 (0 hours, 7 minutes and 6 seconds)
Acquisition speed   : 9.65 MByte/s (0 hours, 6 minutes and 36 seconds)
Verification speed  : 131.83 MByte/s (0 hours, 0 minutes and 29 seconds)
```

Listagem 5.1: Excerto do ficheiro .info

C) Verificação da imagem recolhida

Embora tenha sido realizada a verificação pós-recolha da hash gerada durante o processo de recolha, optamos por validar a hash utilizando outra ferramenta para assegurar que tudo está em conformidade com os padrões estabelecidos para a aquisição de imagens. Para tal, utilizou se a ferramenta *GtkHash*.



(a) Validação hash SHA-256 (b) Validação hash MD5

Figura 5.7: *GtkHash*

Na Figura 5.7, são apresentadas as duas imagens dos resultados da ferramenta *GtkHash* correspondentes às validações das hashes *SHA-256* (também utilizada pelo sistema implementado) e *MD5* (para confirmação de resultados), geradas no momento da recolha da imagem do dispositivo *USB* utilizando as *hashes* geradas com o arquivo *.dd* criado. Dessa forma, podemos confirmar que as hashes correspondem,

isto é, a *hash* do *"Check"* (*hash* gerada no momento da criação) com a *hash* gerada no mesmo instante da validação, garantindo assim que o processo foi realizado corretamente e que o arquivo está íntegro.

D) Manipulação do arquivo *.dd*

Para manipular o arquivo, temos duas opções. A primeira opção seria manipular os bytes do arquivo, o que provavelmente resultaria num arquivo corrompido. A outra opção seria montar o dispositivo de forma virtual para acedermos ao conteúdo e, por exemplo, apagar alguns dos arquivos contidos. Optamos pela segunda abordagem.

Para montar o arquivo *.dd* e aceder ao conteúdo, é necessário seguir os seguintes passos:

Em primeiro lugar, utilizamos o comando *losetup* para associar o arquivo *.dd* a um dispositivo *loop*. O comando executado foi ***sudo losetup -fP acquiredImage.dd***.

Após a execução do comando, uma unidade de volume correspondente ao dispositivo *loop* criado, apareceu no *desktop* do *Linux*. Fez se então o *mount* do dispositivo *loop* (Figura 5.8).

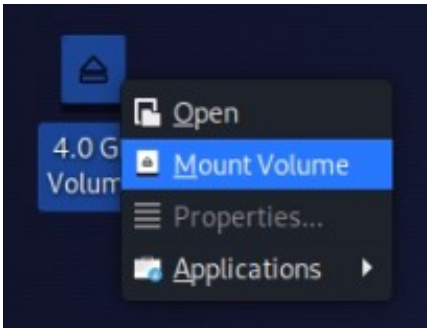


Figura 5.8: *Mount* do volume do arquivo *.dd*

Na Figura 5.9, é apresentado conteúdo do arquivo *.dd* montado. Procedemos à remoção de alguns dos arquivos contidos no dispositivo *loop*, que foi criado a partir do *.dd* recolhido no sentido de simular uma operação de adulteração.

Após realizar as alterações, é necessário seguir o processo inverso para desmontar o volume criado a partir do dispositivo *loop* e remover o próprio dispositivo *loop* do sistema (Figura 5.10).

Após desmontar o volume, foi necessário remover o dispositivo *loop*. Para isso, executamos os seguintes comandos:

losetup -a: Para verificar o nome correto atribuído ao dispositivo *loop* correspondente ao nosso arquivo *.dd*, executamos o comando para obtenção do nome. Com o nome obtido, procedemos à execução do comando ***sudo losetup -d /dev/loop0*** para a remoção do dispositivo *loop*.

Após a remoção do dispositivo *loop* retornamos ao ponto em que temos apenas o arquivo *.dd* criado durante a recolha da imagem do dispositivo *USB*.

E) Verificação da imagem recolhida após manipulação

Após a manipulação da imagem recolhida (*.dd*), foi necessário confirmar se as operações foram realmente gravadas na imagem recolhida. Existia a possibilidade de todas as alterações terem sido efetuadas apenas na memória e nada ter sido guardado no arquivo imagem criado com a ferramenta *Guymager*. Para realizar essa verificação, utilizamos novamente a ferramenta *GtkHash* para gerar a hash do arquivo *.dd* para comparar com a hash criada no momento da recolha da imagem. Em caso de sucesso na verificação das alterações, a hash obrigatoriamente deverá ser diferente da hash original.

Na Figura 5.11, são apresentadas as duas imagens da ferramenta *GtkHash*, correspondentes às validações das hashes *SHA-256* e *MD5*. Os valores originais das hashes *SHA-256* e *MD5* foram inseridos no campo *"Check"*. Após a geração das novas hashes, podemos confirmar que são completamente diferentes, indicando que a imagem recolhida (*.dd*) não está igual àquela que foi recolhida inicialmente, logo as manipulações realizadas foram guardadas com sucesso.

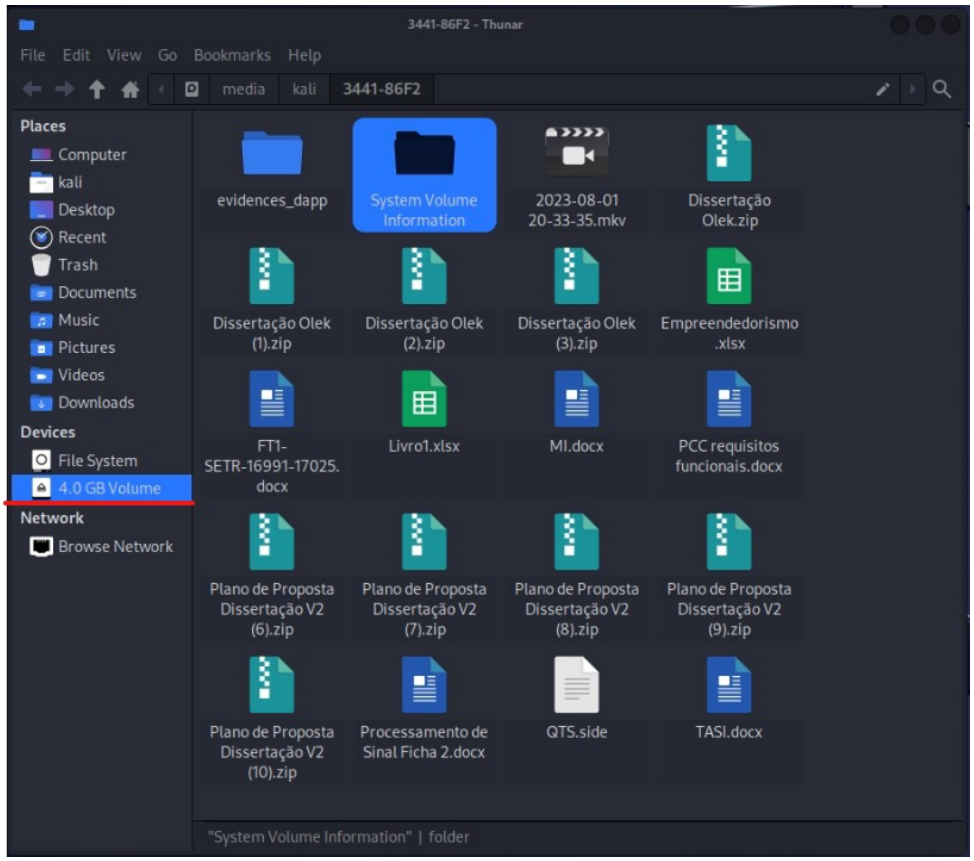


Figura 5.9: Conteúdo do arquivo *.dd mounted*

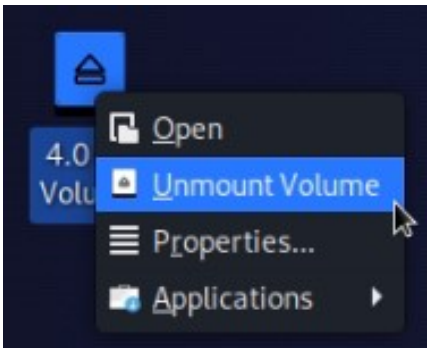


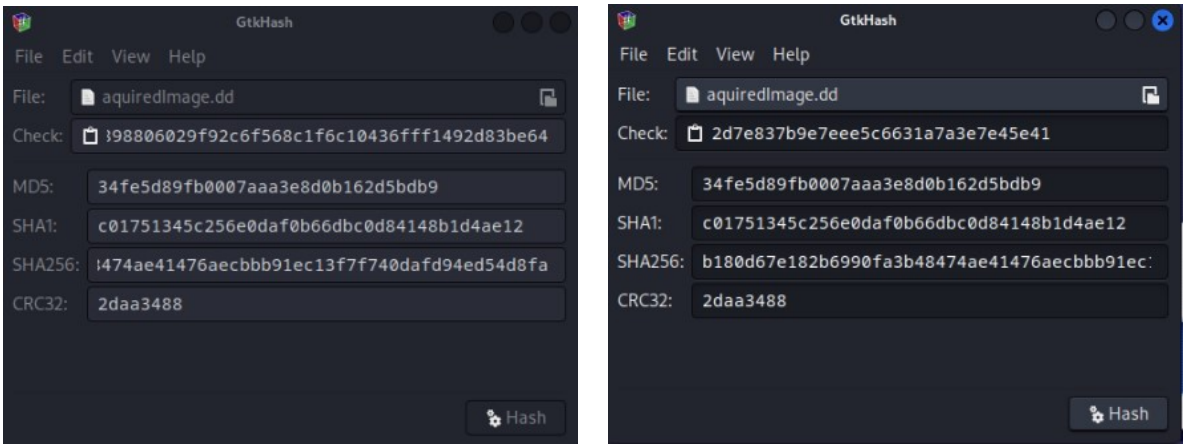
Figura 5.10: *Unmount* do volume do arquivo *.dd*

5.1.4 Observações

Após a realização da experimentação, foi possível obter alguns resultados em relação aos procedimentos realizados. Ao realizar uma manipulação forçada numa imagem recolhida, com o objetivo de ser posteriormente analisada por um cientista forense, não há nenhum aviso ou registo para que o utilizador saiba que ocorreu uma manipulação no arquivo. As formas que existem para um utilizador perceber que um arquivo foi manipulado são:

i) Através do comando

O comando ***stat {nome do arquivo}*** mostra ao utilizador informações sobre o arquivo que ele deseja consultar. Na Figura 5.12, é apresentada a informação do arquivo, indicando a hora e a data do último acesso, da ultima modificação e da ultima alteração ocorrida, juntamente com a data de criação do mesmo. Também aparece o utilizador *Linux* que acedeu por ultimo ao arquivo. Apesar de percebermos



(a) Validação hash SHA-256 (b) Validação hash MD5

Figura 5.11: *GtkHash* depois da manipulação da imagem recolhida

```
┌──(kali㉿kali)-[/]
└─$ stat aquiredImage.dd
  File: aquiredImage.dd
  Size: 4008706048      Blocks: 7829512    IO Block: 4096   regular file
Device: 8,1     Inode: 27          Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2023-11-11 11:41:14.217939856 -0500
Modify: 2023-11-11 11:41:14.213939866 -0500
Change: 2023-11-11 11:41:14.213939866 -0500
 Birth: 2023-11-11 11:04:30.789142287 -0500
```

Figura 5.12: Informações do ficheiro

que houve uma alteração no arquivo, apenas temos a data da última modificação. Não há registos sobre o que aconteceu e o que foi alterado.

ii) Criação constante de *hashes*

Conforme referido na secção 5.1.3, a forma de garantir a integridade do arquivo seria gerar a hash do arquivo desejado e compará-la com a *hash* do momento da criação do arquivo. Para assegurar a integridade de um arquivo, seria necessário gerar *hashes* constantemente e compará-las continuamente com a original. No entanto, realizar essa tarefa manualmente não é uma tarefa simples.

Concluindo, é possível dizer que o processo de rastreabilidade das manipulações do contudo de um ficheiro só permite aferir se e quando ocorreram. O processo não é simples e não é fácil aferir quem de facto alterou.

Em seguida, será realizada a abordagem com a *blockchain*. Posteriormente, os resultados obtidos com a abordagem *Linux* serão comparados com os obtidos na experimentação em *blockchain*.

5.2 Experimentação em *blockchain*

Esta secção, descreve a experimentação realizada utilizando a solução implementada no âmbito desta dissertação. O objetivo da experimentação é o seguinte: criar um documento *PDF*, e armazená-lo no sistema. Em seguida, tentaremos de alguma forma adulterar o arquivo armazenado e analisar como o sistema reagirá, bem como, em caso de sucesso na adulteração, avaliar o quão explícita será a indicação para o utilizador de que um dado arquivo "X" foi adulterado. Como a solução implementada foi publicada num ambiente *Azure*, as configurações sobre a plataforma serão obtidas no Portal do *Azure*.

5.2.1 Plataforma tecnológica de suporte

Como todo o sistema (back-end e front-end) foi hospedado na plataforma *Cloud Azure*, acedeu-se o Portal do Azure para extrair informações sobre as configurações existentes. Os detalhes recolhidos do portal sobre o ambiente *back-end* e *front-end* estão apresentados na Tabela 5.2.

	Back-End	***Front-End***
Stack	Node.js	Node
Runtime version	4.27.5.21549	Must
Node.js Version	Node.js 18 LTS	Node.js 18 LTS
Platform	32 Bit	—
FTP state	FTPS Only	All allowed
HTTP version	1.1	1.1
HTTPS Only	OFF	ON
Server Location	West Europe	West Europe
OS	Windows	Linux
ASP	Application Service Plan Y1:0	Application Service Plan B1:1

Tabela 5.2: Configurações dos ambientes na plataforma *Azure*

Base de dados

A base de dados encontra se hospedada no *MongoDB Cloud*. Acedemos ao site do MongoDB Cloud para recolhermos as informações sobre a base de dados. As informações recolhidas estão representadas na Tabela 5.4.

	MongoDB Cloud
Version	6.0.11
Region	AWS / Paris (eu-west-3)
Cluster Tier	M0 Sandbox (General)
Type	Replica Set - 3 nodes
Data Size	512MB

Tabela 5.4: Configurações da base de dados no *MongoDB Cloud*

5.2.2 Processo

Nesta secção, é descrito o procedimento realizado para a experimentação com a *dApp* desenvolvida com *blockchain Ethereum*.

A) Aceder a aplicação

Para aceder a aplicação, é necessário utilizar o *URL* fornecido pelo portal *Azure*. Para conseguirmos utilizar a aplicação, é necessário instalar a extensão do *MetaMask* no navegador. A Figura 5.13, mostra o momento em que se acede à aplicação, sem *login* no *MetaMask*, nesta etapa, não há nenhuma conta conectada à aplicação.

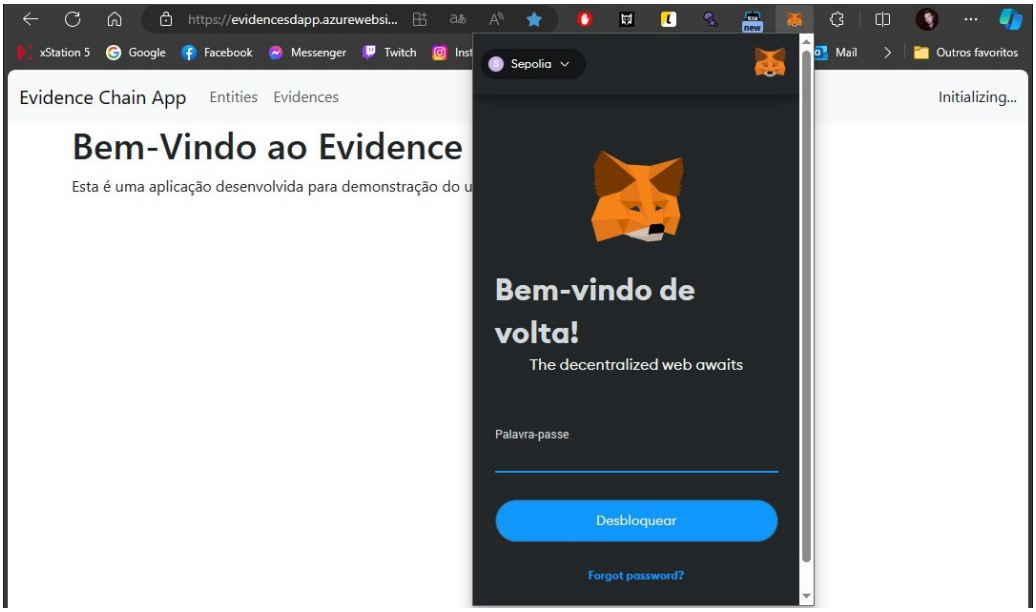


Figura 5.13: Aceder a aplicação

Após acedermos à conta no *MetaMask* (Figuras 5.14), a página da aplicação atualiza e os dados da conta, como o endereço público e o saldo, são apresentados no canto superior direito da página (Figura 5.14 e 5.15, A).

Após o registo, acede-se à lista de evidências, pois o objetivo é adicionar uma nova evidência ao sistema.

A Figura 5.15, mostra a lista de evidências. Neste instante é visível a existência das evidencias com os identificadores U001 e 1.

B) Inserir um registo

Adicionou-se uma nova evidência no sistema (*"Add Evidence"* na Figura 5.15), que corresponde a um novo registo na *blockchain*.

A Figura 5.16, representa o formulário necessário preencher para adicionar um registo de evidência juntamente com os dados que foram preenchido para a criação do registo da evidência.

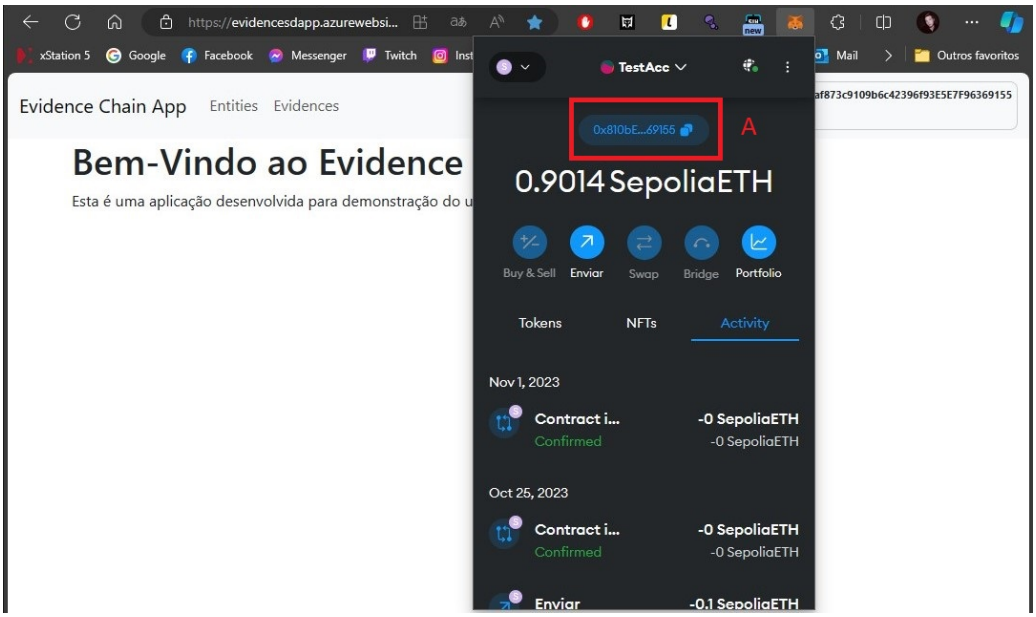


Figura 5.14: *Login* no *MetaMask* concluído

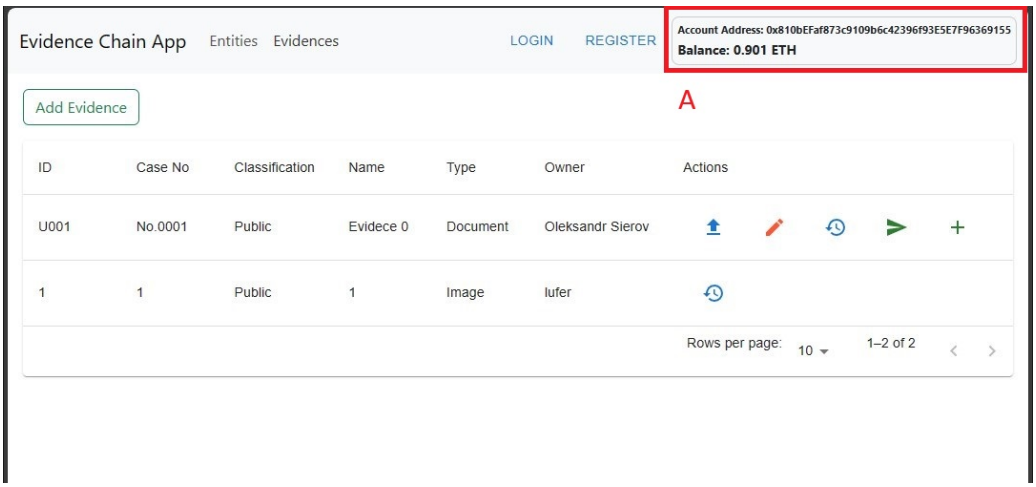


Figura 5.15: Lista de evidências no ponto inicial

Submetida a evidência (*"Submit"* na Figura 5.16), aparece a notificação no *MetaMask* para a aprovação da transação, como referido na secção 4.5.3, uma vez que estamos a executar uma chamada ao *smart contract* para executar uma ação de escrita na *blockchain* (Figura 5.17).

Depois de aprovada a transação, foi necessário aguardar alguns segundos até que a transação fosse concluída. Após a conclusão, uma nova linha apareceu na lista de registos (Figura 5.18), indicando o novo registo.

C) Adicionar um arquivo

Foi feito o *upload* de um novo arquivo (Figura 5.18, A).

Como apenas estamos autenticados na aplicação por meio do *MetaMask*, foi solicitado que fizéssemos *login* usando o nome de utilizador e a *password* (Figura 5.19) para conseguirmos realizar as chamadas à *API* (é através da *API* que se realiza o processo de *upload* do arquivo na base de dados *MongoDB*).

Assim que o processo de *login* é concluído com sucesso, é imediatamente apresentada a janela para efetuar o *upload* do arquivo. A Figura 5.20 mostra essa janela.

Em seguida, selecionamos o arquivo *PDF* e enviamos para o sistema (*"Submit"* Figura 5.21). Quando

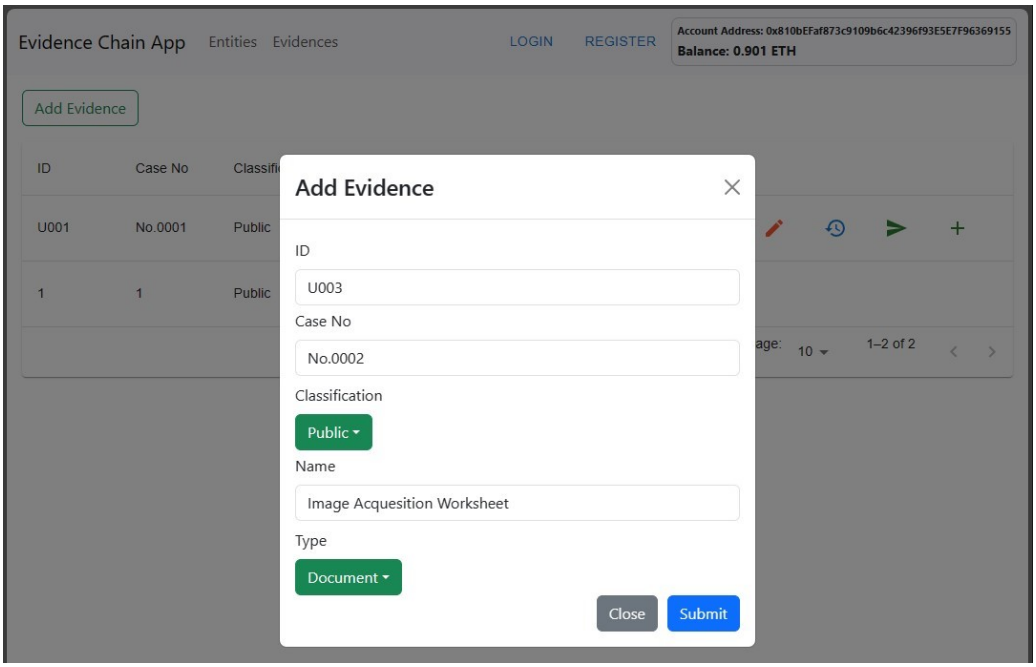


Figura 5.16: Adicionar um registo de evidência

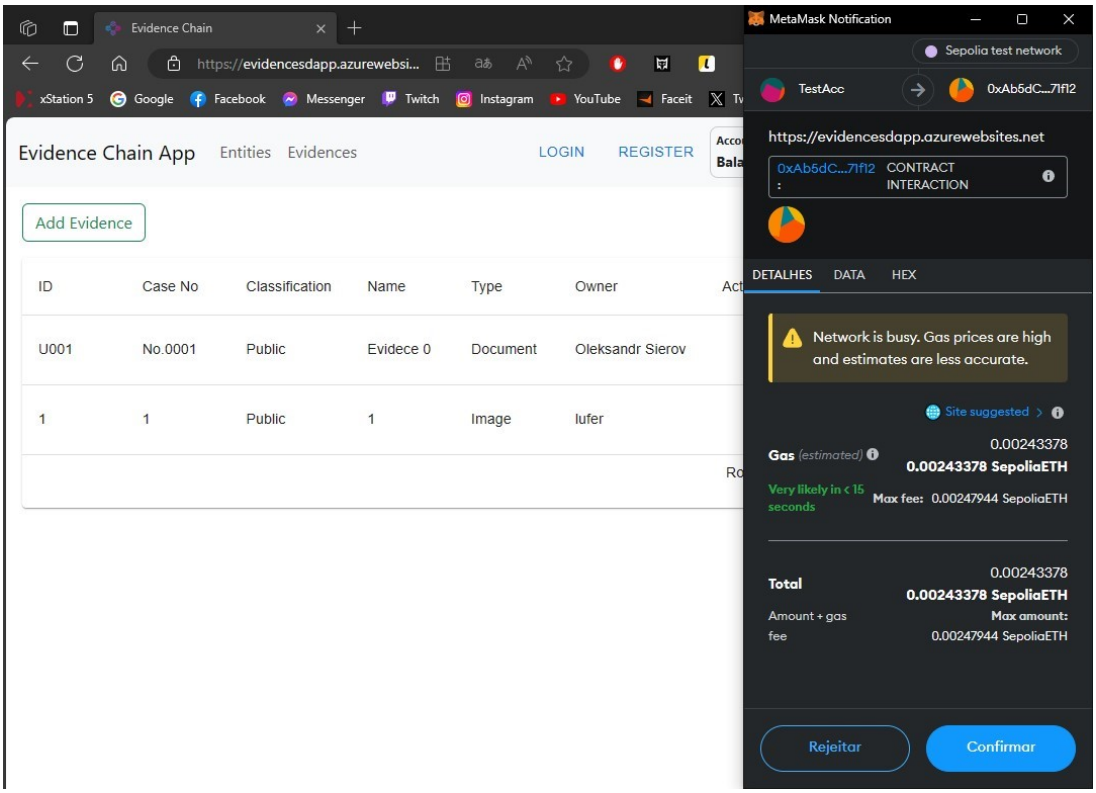


Figura 5.17: Notificação *MetaMask* para aprovação da transação

o *"Submit"* é efetuado, o arquivo selecionado é enviado para a *API*, na *API* é gerada a *hash* sobre o arquivo e, de seguida, o mesmo é armazenado na base de dados.

Novamente, da mesma forma que ocorreu na inserção do registo, recebemos a notificação do *MetaMask* solicitando a aprovação da transação (Figura 5.22). Apesar do documento em si ser armazenado na base de dados do *MongoDB*, os meta-dados do arquivo e a sua hash são armazenados na blockchain.

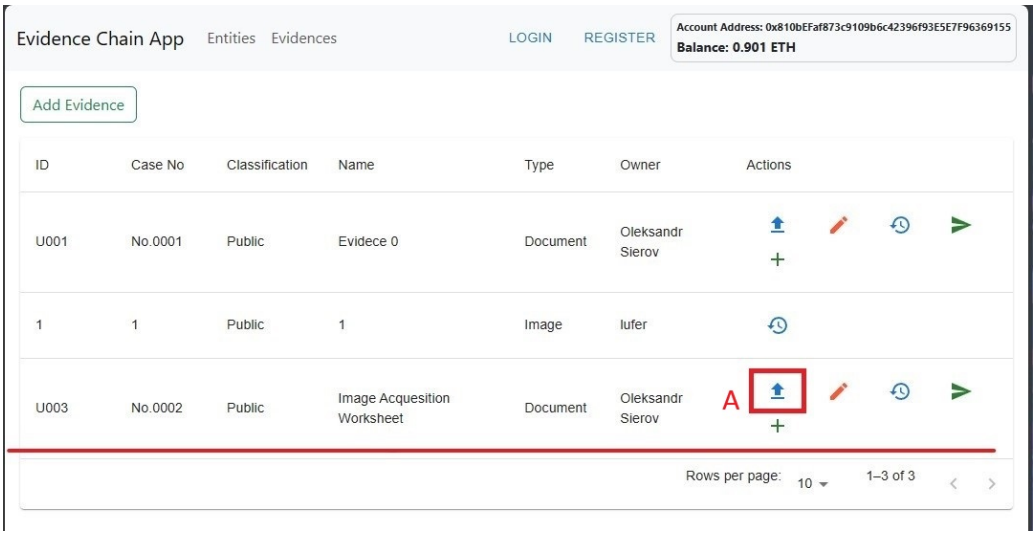


Figura 5.18: Lista de evidências depois de adicionar uma evidência

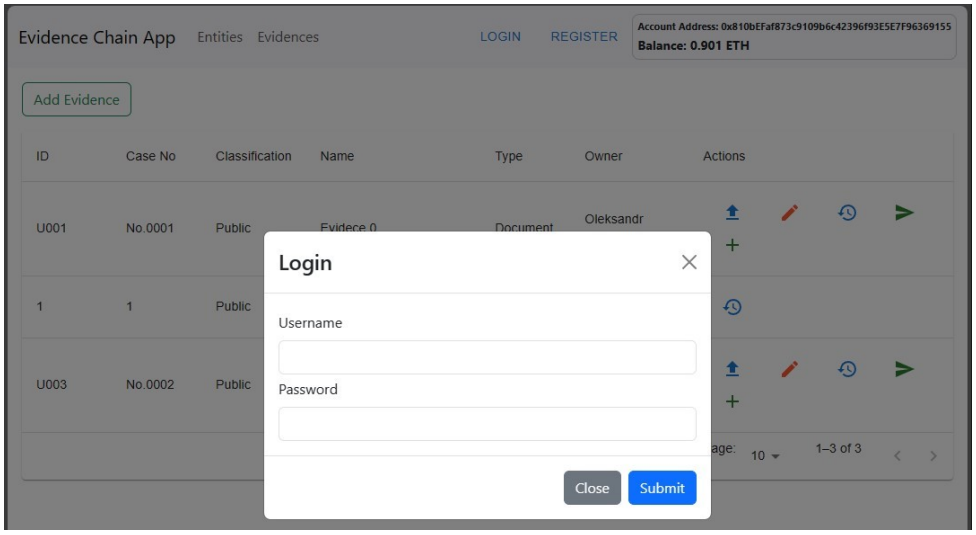


Figura 5.19: *Login* via utilizador e *password*

Após a validação das informações da transação, aprovamos o processo. Novamente, foi necessário aguardar alguns momentos até a conclusão da transação. Depois que a transação foi concluída, verificamos o histórico desse registo para garantir que as informações sobre a adição do arquivo estavam presentes nos registos. Para aceder ao histórico do registo, basta usar o ícone correspondente à ação de visualização do histórico (Figura 5.23, A).

Na Figura 5.24, está representado o histórico do registo de evidências. É claramente notável que existem dois registos: o primeiro corresponde à ação da criação de um registo de evidência, enquanto o segundo corresponde à ação realizada, ou seja, a adição de um arquivo digital. Além desses dois registos, é notável que o segundo registo contém o nome do arquivo que foi adicionado, junto com dois ícones ao lado do seu nome. O ícone A da Figura 5.24 permite-nos efetuar o *download* do arquivo. O ícone B da Figura 5.24 representa o estado de validação, onde o ícone verde simboliza que a validação ocorreu com sucesso, indicando que a comparação da *hash* gerada sobre o arquivo armazenado com a *hash* original resultou em valores estritamente iguais.

Todo este processo é do conhecimento do utilizador, contudo é necessário perceber o acontece no *background* do sistema, aquilo que o utilizador não tem acesso nem conhecimento.

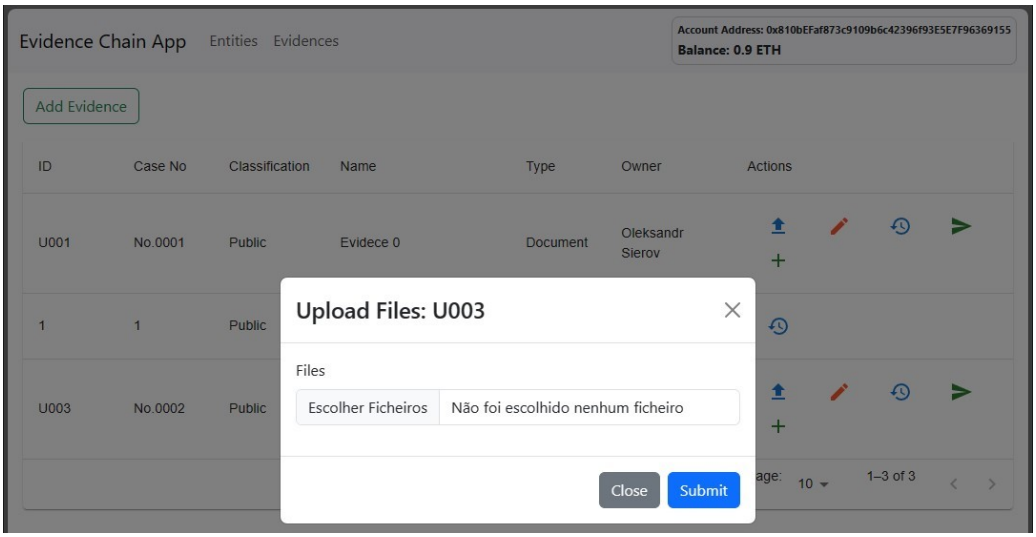


Figura 5.20: View de *Upload* de arquivos

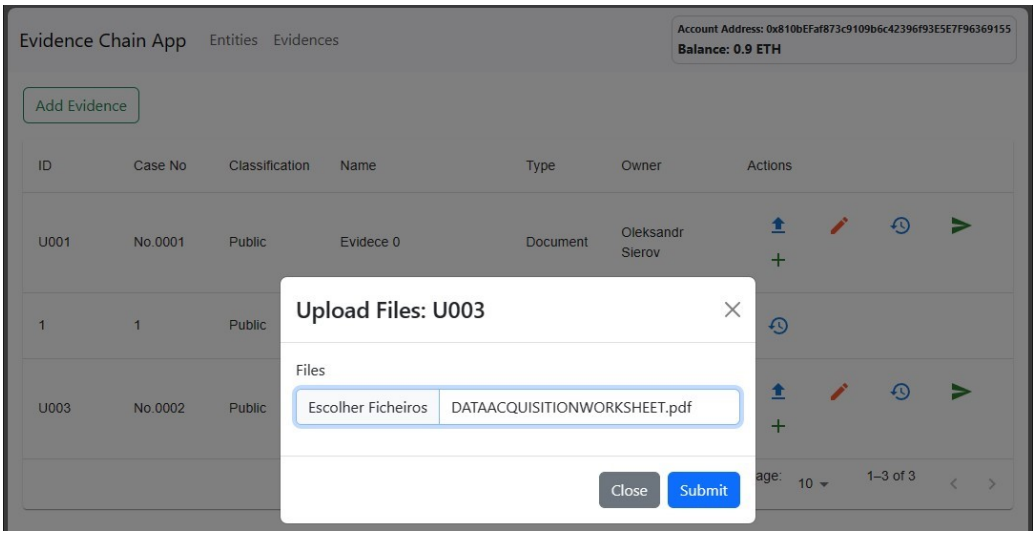


Figura 5.21: View de *Upload* de arquivos com o arquivo selecionado

D) Adicionar um arquivo

Descreve-se o que acontece no lado da base de dados. Além da importância da geração da *hash* do arquivo, também importante tentar ao máximo garantir a integridade do mesmo dentro da base de dados.

Ao adicionar um arquivo novo num registo de evidência o que é armazenado na base de dados está representado pelas figuras seguintes.

Na Figura 5.25, estão representados os meta-dados armazenados na base de dados *MongoDB* para o arquivo adicionado. Podemos observar informações como o identificador do arquivo na base de dados, seu tamanho, a data de *upload* e o nome original do arquivo.

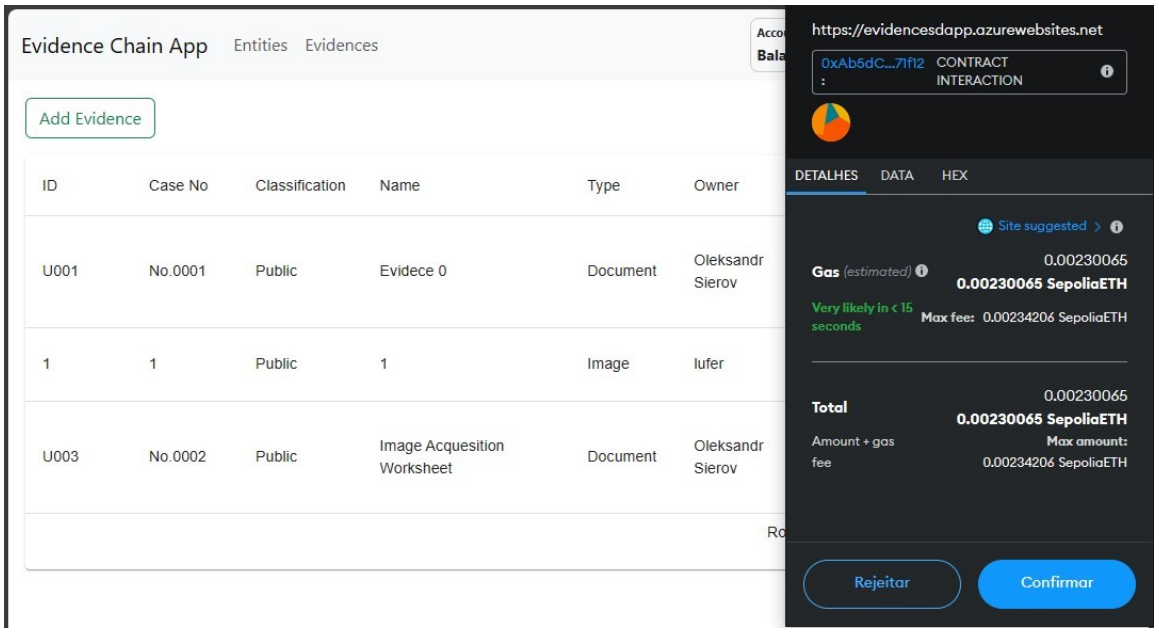


Figura 5.22: Notificação *MetaMask* aprovação de adição de arquivo

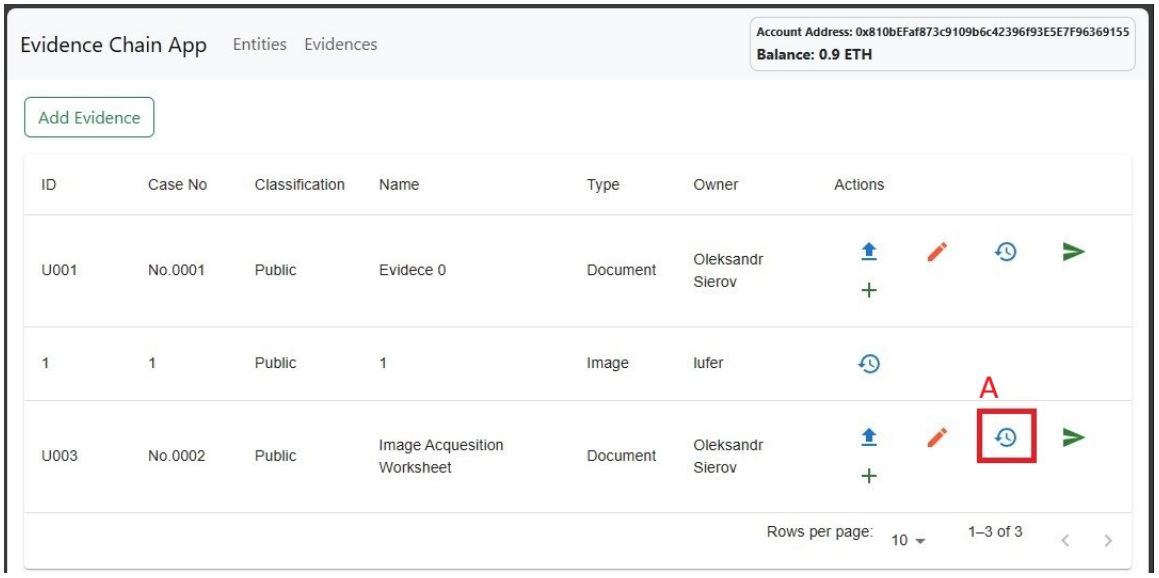


Figura 5.23: Ícone para aceder ao histórico

Na Figura 5.26, está representado o fragmento de dados correspondente ao arquivo armazenado anteriormente na base de dados *MongoDB*. Podemos observar informações como o identificador do fragmento de dados, o identificador do arquivo ao qual os dados binários pertencem, o índice do fragmento (caso o arquivo tenha sido dividido em várias partes) e, eventualmente, os próprios dados binários.

Na Figura 5.27, está representado o registo guardado pelo *trigger* mencionado e implementado na secção 4.4. Esse *trigger* fica à escuta de qualquer alteração que ocorra na base de dados e armazena as informações recolhidas sobre o evento. É possível observar o tipo de operação executada, que no caso é uma inserção (*"insert"*) do arquivo, o horário em que o evento ocorreu, além de detalhes completos do documento adicionado (informação relevante para backup e correções), e o *ID* correspondente a esse documento.

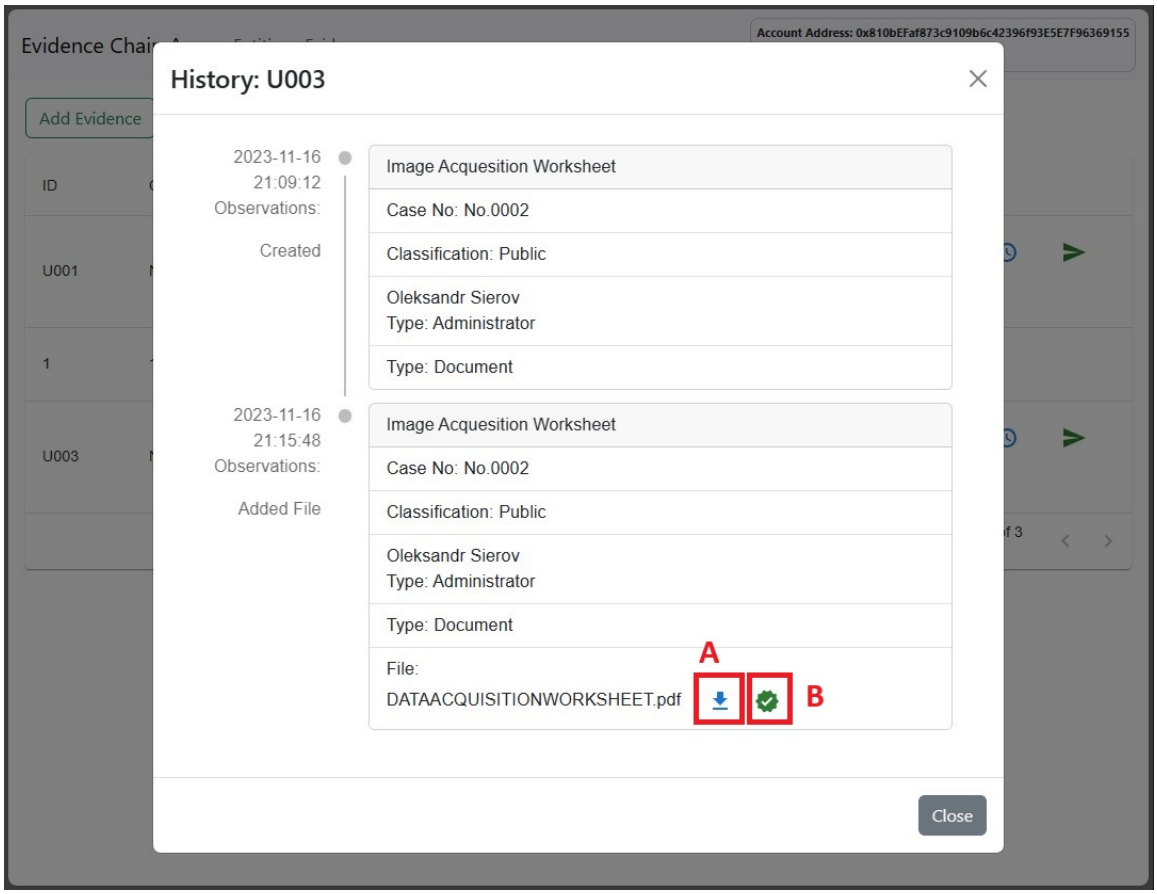


Figura 5.24: Histórico do registo

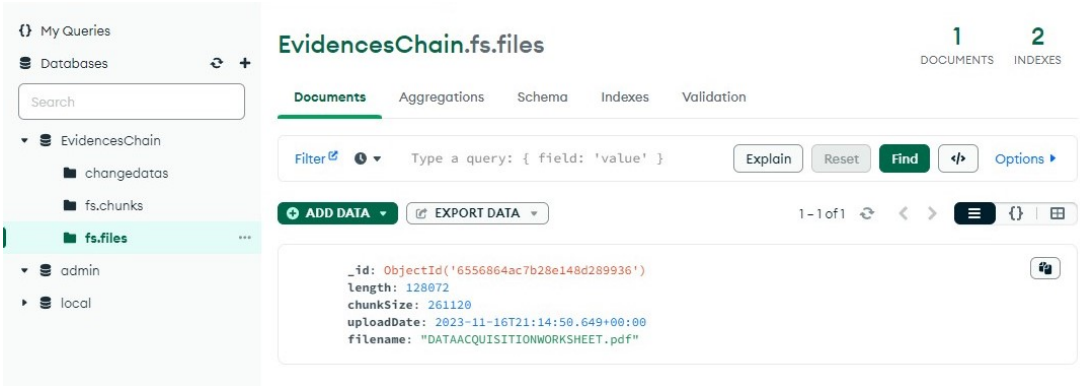


Figura 5.25: Meta-dados do arquivo adicionado

E) Adulterar os dados do arquivo

Sabendo o que está armazenado na base de dados, vamos simular que, de alguma forma, foi possível aceder a nossa base de dados *MongoDB* e que uma adulteração propositada foi feita no arquivo.

A Figura 5.28, mostra a adulteração realizada ao arquivo; parte dos dados binários do arquivo foram apagados propositadamente.

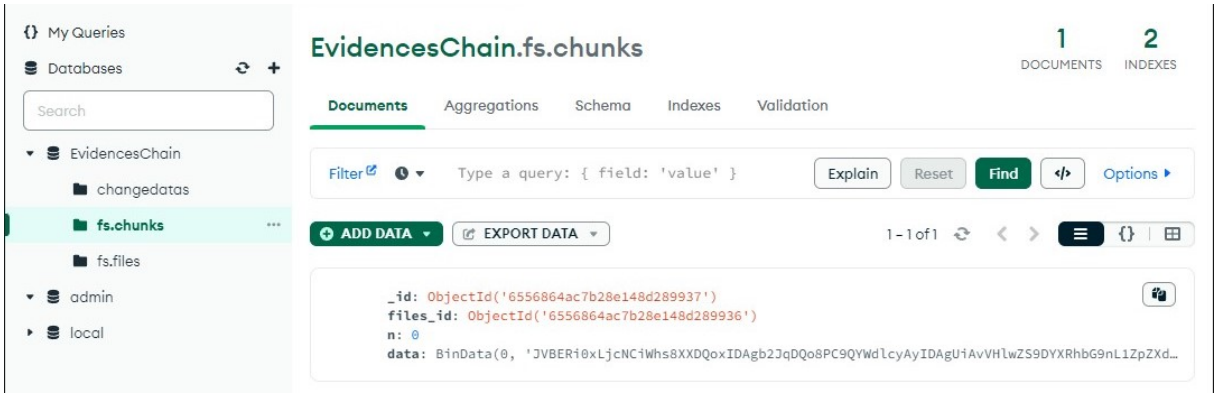


Figura 5.26: Dados do arquivo adicionado

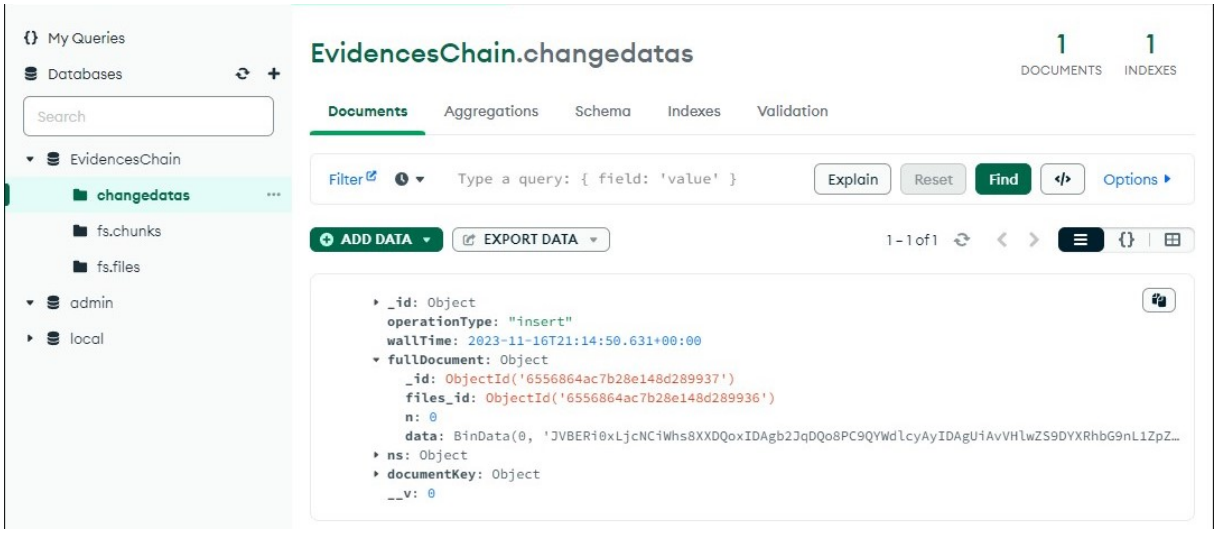


Figura 5.27: Dados guardados pelo *Trigger*

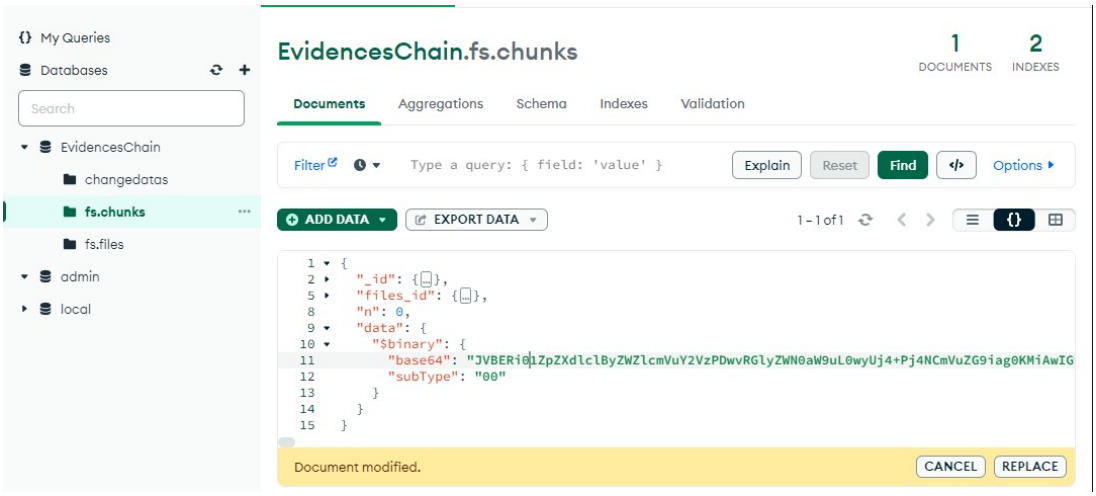


Figura 5.28: Adulteração dos dados binários do arquivo armazenado

Na Figura 5.29, encontram-se os registos armazenados pelo *trigger*. É visível a adição de um novo registo, resultado da adulteração. Podemos validar que o *operation type* agora é *"replace"*, indicando a substituição no registo do arquivo. Além disso, temos o registo dos dados do documento que foi alterado juntamente com a hora da alteração.

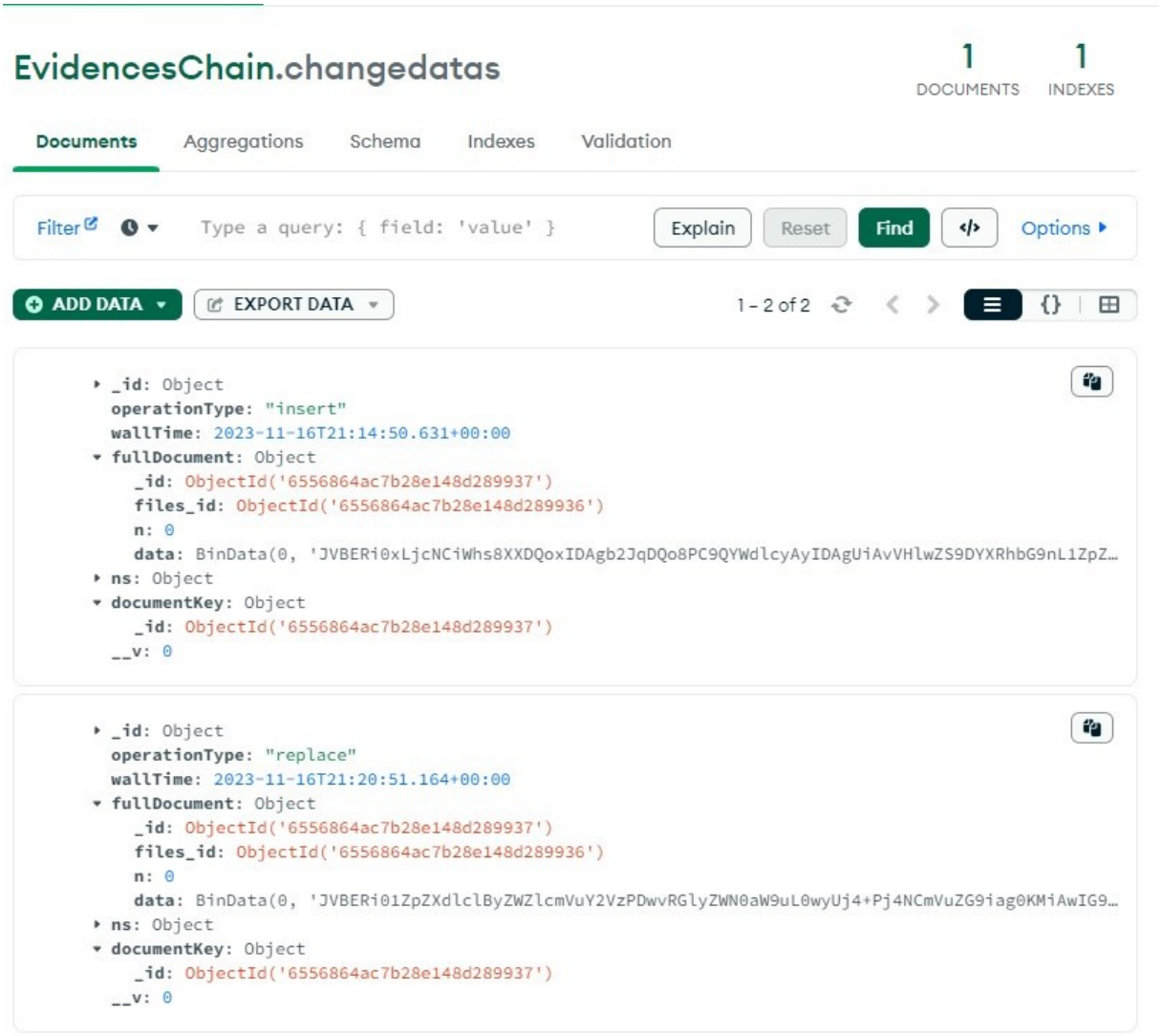


Figura 5.29: Adulteramento dos dados binários capturado pelo *trigger*

Agora, ao voltar à nossa *dApp*, vamos aceder novamente ao histórico do registo para verificar se houve alguma modificação no registo.

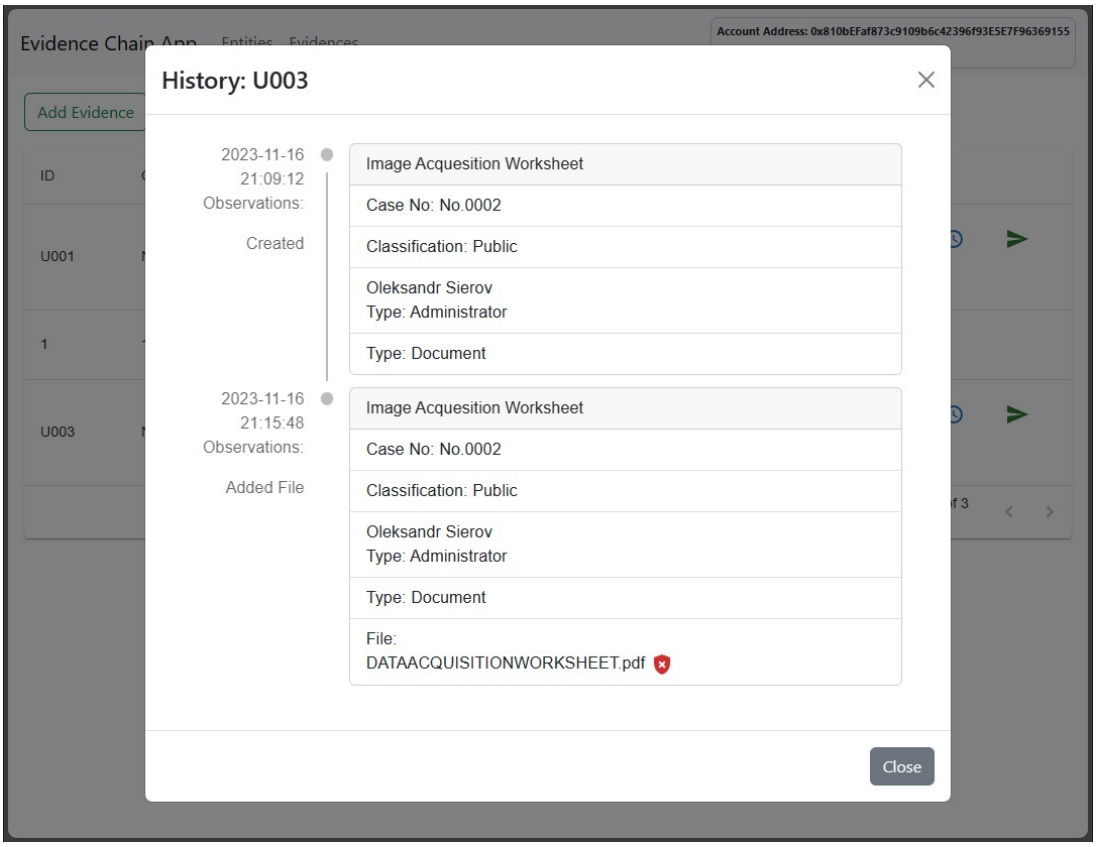


Figura 5.30: Histórico do registo de evidências após a adulteração da base de dados

A Figura 5.30 mostra o histórico do nosso registo de evidências. É percetível que houve uma alteração: ao lado do nome do arquivo adicionado, agora há um ícone vermelho indicando que o arquivo não conseguiu ser validado com sucesso em relação à *hash* original gerada no momento do upload, armazenada na *blockchain*. Isso indica que o arquivo foi adulterado e sua integridade está comprometida.

5.2.3 Observações

Após a realização da experimentação, obtiveram-se os seguintes resultados. Logo após a adulteração do arquivo armazenado na base de dados, o sistema reagiu imediatamente ao registo deste evento de adulteração, preservando todas as informações sobre o que ocorreu, quando ocorreu e as suas consequências.

Verificou-se que a capacidade de detetar essas mudanças permitiu proteger o sistema contra possíveis adulterações, possibilitando ainda a recuperação do conteúdo original através das informações registadas. Esta capacidade sugere uma estratégia eficaz para assegurar a integridade dos dados, permitindo não apenas a deteção, mas também a restauração do conteúdo original, já que são mantidas informações detalhadas sobre os eventos de criação de registos.

Além disso, para o utilizador, é facilmente percetível que ocorreu alguma adulteração. O sistema compara a hash armazenada na *blockchain*, gerada no momento do upload do arquivo, com a hash gerada sobre os dados do arquivo armazenado no momento da visualização do histórico. Isso permite ao utilizador validar o arquivo, de forma automática, sempre que acede ao registo. Quando as hashes não são compatíveis, o sistema apresenta ao utilizador que o arquivo não foi validado com sucesso.

5.3 Comparação dos resultados

Nesta secção, faremos uma comparação de resultados entre a abordagem realizada no *Linux* e a abordagem utilizada com *blockchain*. A Tabela 5.5 contém os principais pontos da experimentação e os diferentes resultados obtidos em cada uma delas.

	Linux	***Blockchain***
Usabilidade	A utilização da ferramenta com interface facilitou o processo de recolha da imagem. Contudo, na etapa de adulteração, foi necessário conhecimento de comandos do sistema operativo Linux para montar o dispositivo.	A utilização da aplicação somente requer a utilização da extensão do *MetaMask*.
Adulteração	A adulteração foi possível assim que se efetuou a montagem da imagem recolhida com a ferramenta *Guymager*, utilizou se os comandos integrados do *Linux*.	A adulteração foi possível na base de dados. Sem o acesso direto a base de dados, não seria possível adulterar o arquivo armazenado.
Validação	Para garantir a integridade do arquivo, é necessário utilizar comandos do *Linux* e/ou ferramentas como o *GtkHash* para gerar uma hash e verificar sua correspondência com a original.	O sistema valida a integridade do arquivo cada vez que o utilizador acede o histórico do registo, garantindo que, durante a análise das atualizações do registo, seja possível visualizar a integridade do arquivo em qualquer momento.
Pós adulteração	No sistema *Linux*, não há meios automáticos para detetar alterações sem o uso de *softwares* externos. A validação constante de *hashes* ou a verificação do estado do arquivo, são as formas disponíveis. Além disso, não é possível obter as mudanças específicas após uma adulteração, impossibilitando a identificação do que foi removido ou adicionado. Da mesma forma, não há opção de restaurar a imagem original após uma adulteração.	Qualquer tipo de alteração na base de dados é automaticamente detetada com a utilização do *trigger* implementado, fornecendo uma forma de rastrear as alterações. Os metadados e a *hash* encontram-se seguramente armazenados na *blockchain* com a identificação do responsável pela inserção. A adulteração é facilmente detetável, e aquilo que foi adulterado é facilmente recuperável e restaurável para o estado original.

Tabela 5.5: Comparação de Resultados

Após analisar a Tabela 5.5, é evidente que a abordagem desenvolvida traz algumas melhorias e formas adicionais de proteção dos dados em comparação com a recolha convencional no *Linux*. Se fosse possível armazenar os dados completos do arquivo na *blockchain*, não seria necessário criar um *trigger*, já que o modelo de uma *blockchain* é concebido para impedir o adulteramento de informações. O que é armazenado na *blockchain* permanece inalterado, uma vez que não há forma de remover ou editar qualquer informação.

Embora ambas as abordagens tenham os seus prós e contras e, apesar de serem semelhantes em certos pontos, divergem um pouco. Fica claro que o sistema desenvolvido é um pouco mais amigável ao utilizador em termos práticos. No entanto, é importante observar que apesar das diferenças e da sua possibilidade comparativa, pois a adulteração dos arquivos está presente nos dois casos, essas duas abordagens se complementariam se fossem utilizadas em conjunto.

6. Conclusão e trabalhos futuros

O crescimento anual de casos judiciais, conforme documentado por Eurojust (2022), exige uma abordagem para a gestão segura das evidências em processos legais. A potencial manipulação de evidências e a dificuldade em controlar tais alterações tornam-se preocupações permanentes neste contexto. Assim, explorar a aplicabilidade da tecnologia *blockchain* para o registo e rastreabilidade dessas evidências representa o principal objetivo deste trabalho de investigação.

A prova de conceito ou protótipo tecnológico desenvolvido neste estudo visa integrar essa inovação no âmbito jurídico, de modo a simplificar e fortalecer o processo para todos os envolvidos. A utilização da *blockchain* oferece não apenas uma camada adicional de segurança, como também uma cadeia imutável e transparente para cada evidência registada no sistema.

Os fundamentos teóricos assentam numa revisão da literatura com pesquisas sobre: i) registo de evidências e processos jurídicos; ii) a tecnologia *blockchain* e as suas diferentes variações, assim como a comparação de algumas *blockchains* mais comuns; iii) soluções existentes no mercado e propostas feitas noutros trabalhos.

Após a revisão da literatura, procedeu-se à análise e modelação, utilizando a engenharia de requisitos para descrever as funcionalidades do sistema. Foram criados diagramas de atividades para cada requisito documentado, de forma a proporcionar uma explicação mais compreensível ao leitor.

Assim que se concluiu a parte teórica do trabalho, avançou-se para o desenvolvimento da solução. Utilizou-se a *blockchain Ethereum* com a linguagem de programação *Solidity*. Foram desenvolvidos *smart contracts* que obedecessem aos requisitos definidos, e também se criou uma aplicação *web*, com a tecnologia *React*, para a integração com os *smart contracts*.

Dada a necessidade de armazenar arquivos e devido às limitações da *blockchain*, desenvolveu-se uma camada *back-end/API* para interligar com uma base de dados *MongoDB*. Para aumentar a integridade dos arquivos na base de dados, recorreu-se à implementação de *triggers* sobre a mesma, de forma a registar qualquer alteração que seja feita na base de dados.

De forma a validar a solução, realizou-se uma experimentação comparativa com o objetivo de comparar duas abordagens: a abordagem convencional elaborada no sistema Linux com ferramentas fornecidas pelo sistema Kali Linux, e a abordagem *blockchain* desenvolvida ao longo deste trabalho. A comparação entre ambas focou-se em quatro pontos principais: i) Usabilidade; ii) Integridade; iii) Validação; iv) Pós-adulteração. As principais conclusões obtidas indicam que a abordagem desenvolvida traz algumas melhorias e métodos adicionais de proteção dos dados em comparação com a abordagem convencional no Linux. Se fosse possível armazenar os dados completos dos arquivos na *blockchain*, não seria necessário criar um *trigger* nem um *back-end*, uma vez que o modelo de uma blockchain é concebido para impedir o adulteramento de informações.

Em suma, é possível afirmar que a introdução da *blockchain* traz avanços e melhorias para a transparência e rastreabilidade dos registros de evidências, um contexto que exige ser tratado com o maior cuidado possível para se aumentar a justiça nos processos.

6.1 Trabalhos futuros

À medida que o sistema foi sendo desenvolvido, o foco principal permaneceu na funcionalidade central, deixando alguns pontos menos críticos como adições para trabalhos futuros. Durante o processo de desenvolvimento, também foram apontadas melhorias a serem implementadas. Para organizar essas

ideias, os trabalhos futuros foram divididos em duas categorias: *"smart contracts"* e *"non smart contracts"*, ou seja, o que depende dos *smart contracts* e o que não depende.

Do lado dos *smart contracts* existem os seguintes pontos a serem aprimorados:

- Para armazenar mais informações, podemos expandir as classes. Isso permite capturar o máximo de detalhes necessários, ampliando a gama de situações que é possível abranger e tornando o sistema mais útil em diferentes cenários.
- Para ampliar a cobertura de testes do sistema, foram criados testes unitários para avaliar os *smart contracts*, que estão disponíveis no código-fonte da solução. No entanto, aumentar o numero dos mesmo para uma maior cobertura seria importante.
- Experimentar a aplicação da solução numa *blockchain* privada e avaliar a sua adaptação às exigências de negócio do sistema. Esta abordagem pode até incluir a tentativa de armazenar todos os dados dos ficheiros adicionados aos registos, com o intuito de eliminar a necessidade do atual *back-end* e da base de dados *MongoDB* utilizados no sistema.

Do lado do *non smart contracts* existem os seguintes pontos:

- Melhorar a interface de utilizador para tornar as paginas mais intuitivas.
- Modificar o registo de evidências para permitir o armazenamento de um arquivo durante a sua criação.
- Incluir uma página para visualização do histórico de um registo de um utilizador.
- Separar o *trigger* do *back-end* e criar uma aplicação isolada, ocultando-a dos utilizadores para isolar as duas funcionalidades. Da mesma forma, implementar uma base de dados totalmente independente do sistema para armazenar os eventos capturados pelo *trigger*, visando aumentar a segurança do sistema.
- Aumentar a segurança no registo com o envio de notificação para email para a verificação da conta e validação do utilizador que se registou.

Existem diversas outras funcionalidades que seria ótimo implementar. No entanto, é evidente que foram identificados pontos claros para melhoria à medida que o sistema foi sendo desenvolvido. Essas melhorias serão implementadas e aprimoradas posteriormente.

Bibliografia

A., Rodriguez & Smith J. 2023. *Ensuring chain of custody in digital forensic analysis: Best practices and future directions.* Volume 18, Issue 1, p45-58. Digital Forensic Science Journal.

Aggarwal, Shubhani & Neeraj Kumar. 2021. *Chapter fifteen - blockchain 2.0: Smart contracts* Volume 121. Elsevier. https://doi.org/10.1016/bs.adcom.2020.08.015.

Agile, Scaled. 2023. *Domain modeling.* scaledagileframework. https://scaledagileframework.com/domain-modeling/.

Anwar, Hasib. 2020. *Best blockchain platforms for enterprises.* 101blockchains. https://101blockchains.com/best-blockchain-platforms/.

Bano, Shehar, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn & George Danezis. 2017. *Consensus in the age of blockchains.* University College London. https://doi.org/10.48550/arXiv.1711.03936.

Betrybe. 2022. *Engenharia de requisitos: quais as etapas e como funciona?* betrybe. https://blog.betrybe.com/tecnologia/engenharia-de-requisitos-tudo-sobre/.

Binance. 2018. *A história da blockchain.* Binance. https://academy.binance.com/pt/articles/history-of-blockchain.

Binance. 2020. *Blockchains privadas, públicas e de consórcios - qual a diferença?* Academy Binance. https://academy.binance.com/pt/articles/private-public-and-consortium-blockchains-whats-the-difference.

Brocke, Jan Vom, Alan Hevner & Alexander Maedche. 2020. *Introduction to design science research.* Springer, Cham. https://doi.org/10.1007/978-3-030-46781-4_1.

Browne, Draycott. 2005. *What is the definition of evidence in uk criminal law?* Draycottbrowne. https://www.draycottbrowne.co.uk/investigations/types-evidence.

Budhi, Veera. 2022. *Advantages and disadvantages of blockchain technology.* Forbes. https://www.forbes.com/sites/forbestechcouncil/2022/10/20/advantages-and-disadvantages-of-blockchain-technology/?sh=19130cee3453.

Chain, Chiliz. 2023. *Blockchain.* Chiliz Chain Developer Guid. https://docs.chiliz.com/glossary/blockchain.

Chainlink. 2023. *Top 6 smart contract languages in 2023.* chain.link. https://chain.link/education-hub/smart-contract-programming-languages.

Citizens-Information. 2021. *Hala systems: Blockchain case study for saving lives and mitigating war crimes.* Consensys. https://consensys.net/blockchain-use-cases/social-impact/hala-systems-case-study/.

Cointelegraph. 2023. *Architectural components of the ethereum blockchain: What are they?* cointelegraph. https://cointelegraph.com/learn/architectural-components-of-the-ethereum-blockchain-what-are-they.

Ethereum. 2023a. *Consensus mechanisms.* Ethereum. https://ethereum.org/en/developers/docs/consensus-mechanisms/.

Ethereum. 2023b. *Ethereum development documentation.* Ethereum. https://ethereum.org/en/developers/docs/.

Ethereum. 2023c. *Ethereum whitepaper.* Ethereum. https://ethereum.org/en/whitepaper/#ethereum.

Eurojust. 2022. *Relatório anual da eurojust 2021.* Luxemburgo: Serviço das Publicações da União Europeia. https://www.eurojust.europa.eu/sites/default/files/assets/eurojust-annual-report-2021-pt.pdf.

George, Benedict. 2023. *What is a dapp? decentralized apps explained.* CoinDesk. https://www.coindesk.com/learn/what-is-a-dapp-decentralized-apps-explained/.

Gospel, Darlington. 2023. *Use axios npm to generate http requests.* Knowledgehut. https://www.knowledgehut.com/blog/web-development/http-requests-with-axios-npm#what-is%C2%A0axios?%C2%A0.

gvoncken. 2023. *Guymager introduction.* Guymager. https://guymager.sourceforge.io/.

Hayes, Adam. 2021. *Peer-to-peer (p2p) service: Definition, facts, and examples.* Investopedia. https://www.investopedia.com/terms/p/peertopeer-p2p-service.asp.

Heaven, Tristan et al. 2023. *Guymager introduction.* GitHub. https://github.com/gtkhash/gtkhash.

Heinrich, Nagel & Norton Jerry. 2022. *"evidence". encyclopedia britannica.* Britannica. https://www.britannica.com/topic/evidence-law.

Ho, Hock Lai. 2015. *The legal concept of evidence.* Stanford Encyclopedia of Philosophy. plato.stanford.edu.

IBM. 2021. *Use-case diagrams.* IBM. https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagrams-use-case.

IBM. 2023. *What is blockchain technology?* IBM. https://www.ibm.com/za-en/topics/what-is-blockchain.

Information, Citizens. 2022. *Real evidence.* citizensinformation. https://www.citizensinformation.ie/en/justice/evidence/real_evidence.html.

Infura. 2023. *Documentation.* infura. https://docs.infura.io/.

INTERPOL. 2019. *Interpol global guidelines for digital forensics laboratories.* INTERPOL.

Iredale, Gwyneth. 2020. *Top disadvantages of blockchain technology.* 101blockchains. https://101blockchains.com/disadvantages-of-blockchain/.

Kashettar, Swathi. 2023. *Top 5 blockchain programming languages for smart contracts.* analyticsinsight. https://www.analyticsinsight.net/top-5-blockchain-programming-languages-for-smart-contracts/.

Lamounier, Lucas. 2018. *O guia definitivo da tecnologia blockchain: Uma revolução para mudar o mundo.* 101blockchains. https://101blockchains.com/pt/tecnologia-blockchain-guia/.

Lauslahti, Kristian, Juri Mattila & Timo Seppälä. 2017. *Smart contracts – how will blockchain technology affect contractual practices?* ETLA.

Lawton, George. 2023. *Top 9 blockchain platforms to consider in 2023.* TechTarget. https://www.techtarget.com/searchcio/feature/Top-9-blockchain-platforms-to-consider.

Li, Kuan-Ching, Xiaofeng Chen, Hai Jiang & Elisa Bertino. 2019. *Essentials of blockchain technology* 1st edition. Chapman and Hall/CRC.

Li, Meng, Chhagan Lal, Mauro Conti & Donghui Hu. 2021. *Lechain: A blockchain-based lawful evidence management scheme for digital forensics* Volume 115. Future Generation Computer Systems. https://doi.org/10.1016/j.future.2020.09.038.

Li, Xiaoqi, Peng Jiang, Ting Chen, Xiapu Luo & Qiaoyan Wen. 2017. *A survey on the security of blockchain systems*. Elsevier.

Liu, Bin, Xiao Liang Yu, Shiping Chen & Xiwei Xu. 2017. *Blockchain based data integrity service framework for iot data*. IEEE.

Lucidchart. 2021. *O que é diagrama de atividades uml?* Lucidchart. https://www.lucidchart.com/pages/pt/o-que-e-diagrama-de-atividades-uml.

McCubbin, Gregory. 2023. *Intro to web3.js · ethereum blockchain developer crash course*. dappuniversity. https://www.dappuniversity.com/articles/web3-js-intro.

MetaMask. 2023. *Metamask*. MetaMask. https://metamask.io/.

Moreland, Kirsty. 2022. *Proof-of-work*. Ledger. https://www.ledger.com/academy/blockchain/what-is-proof-of-work.

Nakamoto, Satoshi. 2008. *Bitcoin: A peer-to-peer electronic cash system*. Bitcoin. https://bitcoin.org/bitcoin.pdf.

NestJS. 2023. *Nestjs docs*. nestjs. https://docs.nestjs.com/.

Olsena, Petter & Melania Borit. 2013. *How to define traceability* Volume 29, Issue 2. Trends in Food Science & Technology. https://doi.org/10.1016/j.tifs.2012.10.003.

P., Brzoska & Thompson P. 2022. *Ensuring the integrity of digital evidence: A case study on initial collection procedures.* Volume 17, Issue 3, p79-94. Journal of Digital Forensics, Security and Law. https://www.jdfsl.org/.

Paradigm, Visual. 2022. *What is use case diagram?* visual-paradigm. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/.

Pello, Rauno. 2018. *Design science research — a short summary*. Medium. https://medium.com/@pello/design-science-research-a-summary-bb538a40f669.

Philip, Abin Oommen & RA.K. Saravanaguru. 2023. *Multisource traffic incident reporting and evidence management in internet of vehicles using machine learning and blockchain* Volume 117, Part B. Engineering Applications of Artificial Intelligence. https://doi.org/10.1016/j.engappai.2022.105630.

Pratt, Mary K. 2021. *Top 10 benefits of blockchain technology for business*. TechTarget. https://www.techtarget.com/searchcio/feature/Top-10-benefits-of-blockchain-technology-for-business.

Providers, Association Forensic Science. 2019. *Guidelines for the minimum standard for the documentation, collection, packaging, and secure storage of physical evidence.* Association Forensic Science Providers.

React. 2023. *React*. React. https://react.dev/.

Rosenberg, Eric. 2022. *What is a consensus mechanism?* TheBalanceMoney. https://www.thebalancemoney.com/what-is-a-consensus-mechanism-5211399.

Saini, Ankita. 2021. *What is mongodb – working and features*. geeksforgeeks. https://www.geeksforgeeks.org/what-is-mongodb-working-and-features/.

Shah, Javed. 2022. *Permissionless vs. permissioned blockchains: Pros & cons*. 1kosmos. https://www.1kosmos.com/blockchain/permissionless/.

Solidity. 2023. *Solidity*. Solidity. https://soliditylang.org/about/.

Sommerville, Ian. 2015. *Software engineering* 10th edition. Pearson Education Limited.

Sufiyan, Taha. 2023. *What is node.js: A comprehensive guide*. Simplilearn. https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs#:~:text=js%3F-,Node.,Developers%20use%20Node.

Suite, Truffle. 2023a. *What is drizzle?* trufflesuite. https://trufflesuite.com/docs/drizzle/.

Suite, Truffle. 2023b. *What is ganache?* trufflesuite. https://trufflesuite.com/docs/ganache/.

Suite, Truffle. 2023c. *What is truffle?* trufflesuite. https://trufflesuite.com/docs/truffle/.

Swan, Melanie. 2015. *Blockchain: Blueprint for a new economy.* O'Reilly Media.

T., Nguyen & Wilson C. 2023. *Ensuring chain of custody in the digital forensic laboratory: A comprehensive approach to transporting and storing digital evidence.* Volume 15, Issue 2, p156-172. International Journal of Digital Forensics & Cyber Security.

Tasnim, Maisha Afrida, Abdullah Al Omar, Mohammad Shahriar Rahman & Md. Zakirul Alam Bhuiyan. 2018. *Crab: Blockchain based criminal record management system* Volume 11342. Springer, Cham. https://doi.org/10.1007/978-3-030-05345-1_25.

Tommaso, Anna, Puntarello Maria, Cannella Giovanni & Scalzo Giovanni ... 2023. *The chain of custody in the era of modern forensics: From the classic procedures for gathering evidence to the new challenges related to digital data.* Academic Editor. https://www.mdpi.com/2227-9032/11/5/634.

Tsai, Fu-Ching. 2021. *The application of blockchain of custody in criminal investigation process* Volume 192. Procedia Computer Science.

Watson, David & Andrew Jones. 2013. *Chapter 2 - forensic laboratory accommodation* Volume 13. Syngress. https://doi.org/10.1016/B978-1-59749-742-8.00002-9.

Y., Chen & Jones R. 2022. *Digital chain of custody documentation: Challenges and solutions.* Volume 39. Digital Investigation.

TRACEABILITY OF EVIDENCE IN THE LEGAL DOMAIN

Oleksandr Sierov
Master in Informatics Engineering
Supervisor Luís Gonzaga Martins Ferreira

BACKGROUND

Increasingly, **events involving aggression**, **abuse**, **violations of laws** and **principles**, **mistreatment**, and **criminal acts** are **frequent**.

According to Eurojust (2022), over the 8 years included in the study (from 2014 to 2021), there was an average annual **growth of 15% in the opening of new judicial processes** [1]. Holding those responsible for these crimes accountable, understanding who, how, and why, or penalizing those who commit these crimes, are tasks of high complexity, both due to the difficulty in finding evidence and safeguarding it.

This research aims to explore the application of Blockchain-related technologies as a means of supporting these requirements.

Can the immutability and traceability demanded and guaranteed in financial transactions be equally applied to these processes? Can evidence recorded using this process constitute future evidence in legal processes?

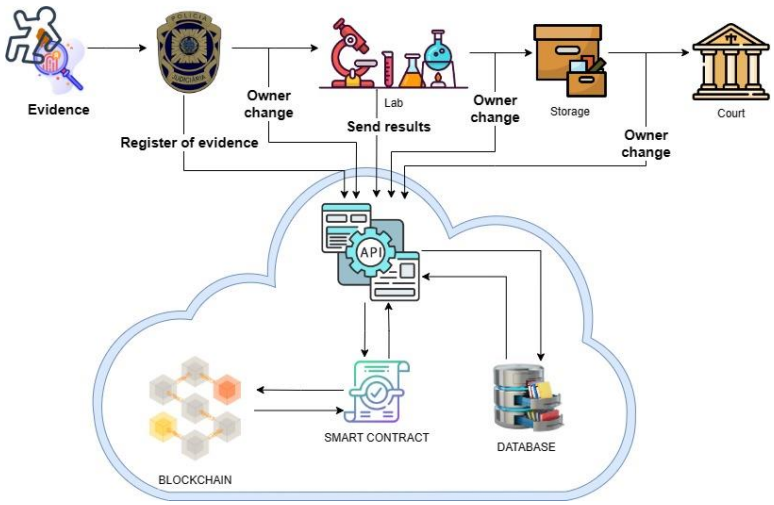
The research will go towards a proposal for evidence registration and traceability, based on Blockchain, to ensure its use in justice processes.

OBJECTIVES

The main objective of this research is to develop a solution that supports the registration of evidence using blockchain technology and ensures the traceability of the entire process from collection to its use in a legal proceeding.
To achieve this, the main objectives are:

1. Understand the requirements for tracking evidence in a blockchain chain.
2. Develop an architecture that supports traceability and registration of evidence using blockchain technology.
3. Implement the architecture on a local blockchain.
4. Demonstrate the applicability of the proposal with a case study of evidence registration in a domain to be determined.

TRACEABILITY OF EVIDENCES



METHODOLOGY

The methodology to be used in this research is based in twos approaches: i) Design Science Research (DSR), which is an investigative approach that aims to create new solutions and build a new reality to solve problems. The DSR process typically includes six stages: problem identification, definition of goals for a solution, conception and development of artifacts, demonstration of the solution, evaluation of the solution, and communication of results [2]; and ii) Experimental Design that will orient all practical developments and experimentations.

EXPECTED RESULTS

In the sequence of this research, the following results are expected:

- An experimental platform that support Blockchain and Smart Contracts;
- A Smart Contract for an evidence registration;
- A Proof of Concept (PoC) of blockchain for evidences registration;
- A visualization application for handling all the process: registration, monitorization and use of evidences in court.
- Scientific paper with publication of results

BIBLIOGRAPHY

[1] Eurojust. 2022. Relatório anual da eurojust 2021. Luxemburgo: Serviço das Publicações da União Europeia. https://www.eurojust.europa.eu/sites/default/files/assets/eurojust-annual-report-2021-pt.pdf

[2] Pello, Rauno. 2018. Design science research — a short summary. Medium. https://medium.com/@pello/design-science-research-a-summary-bb538a40f669.

CONTACTS
a16991@alunos.ipca.pt (Oleksandr Sierov)
lufer@ipca.pt (Luis Gonzaga Martins Ferreira)

LinkedIn Learning

Become a Blockchain Developer

Rota de aprendizagem concluída por Oleksandr Sierov
jun 20, 2023 at 04:59PM UTC • 14 horas 17 minutos

Principais competências abordadas

Criptografia **Blockchain** **Ethereum**



Líder de estratégia de conteúdo, Learning



Código do certificado: 9eb3aaf6f270e68a800f8835e5f2d3238fef1a820d2137edf4cf7b6c28ba40f0