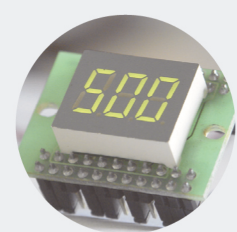
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Contratos Inteligentes para a Gestão Automatizada de Direitos Musicais

MARIANA QUINTAS PINHEIRO

novembro de 2025

POLITÉCNICO
DO PORTO

Politécnico do Porto

Instituto Superior de Engenharia do Porto

Contratos Inteligentes para a Gestão Automatizada de Direitos Musicais

Mariana Quintas Pinheiro

Mestrado em Engenharia Electrotécnica e de Computadores
Área de Especialização em Sistemas e Planeamento Industrial

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

7 de Novembro, 2025

Esta dissertação satisfaz, parcialmente, os requisitos que constam da Ficha de Unidade Curricular de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Electrotécnica e de Computadores, Área de Especialização em Sistemas e Planeamento Industrial.

Candidato: Mariana Quintas Pinheiro, Nº 1190870, `1190870@isep.ipp.pt`

Orientação Científica: Maria Benedita Campos Neves Malheiro, `mbm@isep.ipp.pt`

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

7 de Novembro, 2025

Resumo

A presente dissertação propõe o desenvolvimento de uma plataforma designada SoundSlice, que visa automatizar a gestão de direitos de autor em conteúdos musicais reutilizados e na criação de *mixes*, através da integração de tecnologias *blockchain* e contratos inteligentes. O sistema permite o registo de obras originais, reutilizações parciais, a combinação de múltiplas faixas em novas composições (*mixes*) e a atribuição automática de compensações aos titulares de direitos, assegurando transparência e rastreabilidade em todo o processo.

A solução combina uma infraestrutura centralizada, suportada por uma base de dados MongoDB e armazenamento de ficheiros GridFS, com uma camada descentralizada baseada em Ethereum, responsável pela execução dos contratos inteligentes que formalizam a partilha de *royalties*.

A nível prático, foi implementado um *frontend web* que permite o *upload*, análise, reutilização e criação de *mixes* musicais, bem como um *backend* Node.js que gere a lógica de negócio e a comunicação com a *blockchain*. O desenvolvimento da plataforma baseou-se nos conceitos teóricos e modelos de integração propostos pelos padrões *Smart Contracts for Media* (SC4M) e *Interactive Music Application Format* (IMAF), os quais orientaram a estruturação de metadados, a modelação de contratos e o desenho da arquitetura da plataforma.

Por fim, foram conduzidos testes funcionais, de desempenho e de usabilidade que demonstraram o correto funcionamento da plataforma, a eficiência na execução de transações e a aceitação positiva por parte dos utilizadores, validando a viabilidade e o contributo da abordagem proposta.

Palavras-Chave: *Blockchain*, Contratos Inteligentes, Criação de *Mixes*, Ethereum, Gestão de Direitos de Autor, IMAF, Reutilização Musical, SC4M.

Abstract

This dissertation presents the design and development of the SoundSlice platform. It aims to automate copyright management of reused musical contents and mix creation through the integration of blockchain technology and smart contracts. The system enables the registration of original works, the partial reuse, the combination of multiple tracks into new compositions (mixes), and the automatic distribution of royalties to rights holders, ensuring transparency and traceability throughout the process.

The solution combines a centralized infrastructure, supported by a MongoDB database and GridFS file storage, with a decentralized layer based on Ethereum, responsible for executing smart contracts that formalize royalty sharing.

A web frontend was developed to support the upload, analysis, reuse, and creation of musical mixes, while a Node.js backend handles the business logic and communication with the blockchain. The platform's design was guided by the theoretical foundations and integration principles of *Smart Contracts for Media* (SC4M) and *Interactive Music Application Format* (IMAF) frameworks, which informed the metadata structure, contract modeling, and overall system architecture.

Finally, functional, performance, and usability tests were conducted, confirming the platform's correct operation, transaction efficiency, and positive user perception, validating the feasibility and relevance of the proposed approach.

Keywords: Blockchain, Smart Contracts, Copyright Management, Ethereum, IMAF, Mix Creation, Music Reuse, SC4M.

Agradecimentos

A realização desta dissertação representou não apenas o culminar de um percurso académico, mas também uma etapa de crescimento pessoal e profissional.

Ao longo deste trajeto, tive o privilégio de aprender com professores e orientadores que me desafiaram a pensar, a questionar e a ir mais longe. A todos os docentes do Instituto Superior de Engenharia do Porto, deixo o meu profundo agradecimento pelo conhecimento e inspiração transmitidos.

Um agradecimento especial à minha orientadora, **Professora Benedita Malheiro**, pelo acompanhamento constante, pela orientação científica e pela inspiração que representou em cada fase deste trabalho.

Aos amigos que fiz no ISEP, obrigada por todos os momentos partilhados, pelas risadas, pelas longas horas de estudo e pelo apoio nos dias mais difíceis. Tornaram esta caminhada muito mais leve e memorável.

Aos meus amigos de fora desta instituição, obrigada por me lembrarem que há vida além dos prazos e trabalhos, por acreditarem em mim, mesmo quando o tempo parecia curto demais.

À minha família, não há palavras que expressem a gratidão que vos devo. Obrigada por serem o meu porto seguro, por cada palavra de incentivo, por cada gesto de carinho e por acreditarem sempre que eu era capaz. Tudo o que conquistei tem um pouco de cada um de vocês.

E ao Miguel, obrigada por estares sempre presente, pela paciência, pelo apoio silencioso, pelas palavras certas nas horas incertas e por me fazeres acreditar, todos os dias, que os sonhos valem o esforço.

A todos, o meu mais sincero e sentido **obrigada**. Este projeto é também vosso.

Índice

1 Introdução **1**
1.1 Contextualização . . . 1
1.2 Definição do Problema . . . 2
1.2.1 Objetivos . . . 3
1.2.2 Resultados esperados . . . 4
1.3 Plano de Trabalho . . . 4
1.4 Organização da Dissertação . . . 5

2 Estado da Arte **6**
2.1 Introdução . . . 6
2.2 *Smart Contracts* . . . 7
2.2.1 Origem e Evolução . . . 7
2.2.2 Aplicações . . . 8
2.3 Gestão Autoral Multimédia . . . 9
2.3.1 MPEG . . . 10
2.3.2 *Smart Contracts for Media* . . . 10
2.3.3 *Interactive Music Application Format* . . . 12
2.4 Trabalhos Relacionados . . . 13
2.4.1 Tiny Human and the Future of Music Licensing on Ethereum 13
2.4.2 The Impact of Digital Innovation and Blockchain on the Music Industry . . . 13
2.4.3 Blockchain and Smart Contracts: The Missing Link in Copyright Licensing . . . 14
2.4.4 Consortium Blockchain Smart Contracts for Musical Rights Governance in a *Collective Management Organizations* Use Case . . . 14
2.4.5 Fair Rewarding Mechanism in Music Industry Using Smart Contracts on Public-Permissionless Blockchain . . . 14
2.4.6 Smart Royalties: Blockchain Solutions for Music Metadata and Copyright . . . 15
2.4.7 Application of Blockchain Technology in Digital Music Copyright Management: A Case Study of the VNT Chain Platform 15

2.4.8 Decentralized Music Marketplace: A Blockchain-Based Rights Management Approach . . . 15
2.4.9 Comparação com a SoundSlice. . . . 15
2.5 Sumário . . . 17

3 Proposta de Solução 18
3.1 Introdução . . . 18
3.2 Análise de Requisitos . . . 18
3.2.1 Atores do Sistema . . . 19
3.2.2 Casos de Uso Principais . . . 19
3.3 Arquitetura . . . 20
3.4 *Blockchain* e Contratos Inteligentes . . . 23
3.4.1 Ethereum . . . 23
3.4.2 Solidity . . . 24
3.4.3 Hardhat . . . 24
3.5 Gestão de Direitos Musicais . . . 25
3.5.1 *Smart Contracts for Media* . . . 25
3.5.2 *Interactive Music Application Format* . . . 26
3.6 *Backend* . . . 26
3.6.1 Node.js e Express . . . 26
3.6.2 MongoDB e GridFS . . . 27
3.7 Processamento e Manipulação de Áudio . . . 27
3.8 *Frontend* . . . 28
3.8.1 HTML5, CSS3 e JavaScript . . . 28
3.8.2 Tailwind *Cascading Style Sheet* (CSS) . . . 30
3.8.3 Integração com o *Backend* . . . 30
3.8.4 Integração com a *Blockchain* . . . 31
3.8.5 Integração com o MongoDB . . . 34
3.9 Sumário . . . 35

4 Implementação e Integração 36
4.1 Implementação do *Frontend* . . . 36
4.1.1 Página Principal . . . 36
4.1.2 Registo e Autenticação de Utilizadores . . . 39
4.1.3 Perfil de Utilizador . . . 43
4.1.4 Apresentação de Músicas . . . 48
4.1.5 *Upload* de Músicas . . . 52
4.1.6 Reutilização de Músicas . . . 56
4.1.7 Criação de *Mixes* . . . 58
4.1.8 Páginas Adicionais . . . 60
4.2 Implementação do *Backend* . . . 62

4.2.1 Estrutura Geral da *Application Programming Interface* (API) 62
4.2.2 Gestão de Utilizadores 65
4.2.3 Gestão de Músicas 70
4.2.4 Gestão de Reutilizações 73
4.2.5 Criação e Exportação de *Mixes* 74
4.3 Contratos Inteligentes 75
4.3.1 *Deployment* dos Contratos Inteligentes 75
4.3.2 Execução do `MusicRights` 79
4.3.3 Execução do `MusicReuse` 80
4.3.4 Execução do `MusicMix` 81
4.4 Disponibilização do Código-Fonte 81
4.5 Sumário 82

5 Avaliação e Resultados 83
5.1 Introdução 83
5.2 Configuração Experimental 84
5.3 Métricas de Avaliação 85
5.4 Cenários de Teste 85
5.4.1 Testes de Desempenho 88
5.4.2 Testes de Usabilidade 88
5.5 Resultados e Análise 88
5.5.1 Testes Funcionais 88
5.5.2 Testes de Desempenho 90
5.5.3 Testes de Usabilidade 95
5.6 Sumário 96

6 Conclusão 97
6.1 Contribuições 97
6.2 Reflexões Críticas 98
6.3 Limitações 98
6.4 Trabalho Futuro e Recomendações 99

Referências 100

Anexo A Listagens de Código 106
A.1 Exportação de *Mixes* 106
A.2 Seleção de Trechos 109
A.3 Confirmação da Reutilização 111

Anexo B Contratos Digitais 114
B.1 MusicRights.sol 114
B.2 MusicReuse.sol 116

B.3 MusicMix.sol 118

Lista de Figuras

1.1 Problema atual na gestão de direitos musicais. 3

2.1 Estrutura dos *Smart Contracts for Media* (SC4M). 12
2.2 Estrutura do *Interactive Music Application Format* (IMAF). 13

3.1 Arquitetura da plataforma. 21
3.2 Fluxo de funcionamento da plataforma. 22
3.3 Fluxo de integração da plataforma SoundSlice com a *blockchain* Ethereum. 32

4.1 Interface da página principal da plataforma SoundSlice. 39
4.2 Interface da página de registo (`register.html`). 41
4.3 Interface da página de autenticação (`login.html`). 42
4.4 Interface da página do perfil (`profile.html`). 45
4.5 Interface da página do perfil – secção "Meus Pagamentos". 47
4.6 Interface da página do perfil – secção "Minhas Reutilizações". 48
4.7 Interface da página das músicas do utilizador (`minhas-musicas.html`). 50
4.8 Interface da página de edição de dados das músicas (`editar.html`). . 51
4.9 Interface da página das músicas dos outros utilizadores (`musicas-outros.html`). 53
4.10 Formulário de envio de música. 56
4.11 Interface da página de reutilização de músicas, com a seleção de trecho e cálculo automático do valor a pagar. 58
4.12 Interface da página de criação de *mixes*. 60
4.13 Página informativa "Sobre a SoundSlice". 61
4.14 Página de contacto da plataforma. 61
4.15 Página de políticas de privacidade da plataforma. 62

Lista de Tabelas

2.1 Comparação geral entre trabalhos relacionados e a proposta SoundSlice. 16

5.1 Resumo das métricas de avaliação. 86
5.2 Resultados dos Testes Funcionais. 89
5.3 Resultados da latência e volume de dados da API (1 pedido por *endpoint*). 90
5.4 Resultados consolidados da latência da API (10 pedidos por *endpoint*). 92
5.5 Métricas de desempenho da execução dos contratos inteligentes `MusicRights`. 93
5.6 Métricas de desempenho da execução dos contratos inteligentes `MusicReuse`. 94
5.7 Métricas de desempenho da execução dos contratos inteligentes `MusicMix`. 94
5.8 Resultados do teste SUS. 95

Lista de Listagens

3.1 *Deployment* do contrato inteligente `MusicRights` durante o registo da música 32
3.2 Criação automática do contrato `MusicReuse` no *backend* 33
3.3 Deploy do contrato `MusicMix` para composições colaborativas 34
4.1 Verificação de autenticação do utilizador. 37
4.2 Carregamento assíncrono das músicas mais recentes. 38
4.3 Renderização dinâmica das músicas recentes. 38
4.4 Gestão da navegação entre etapas do formulário. 40
4.5 Envio do formulário de registo para o *backend*. 40
4.6 Encerramento de sessão do utilizador. 42
4.7 Alternância entre secções do painel de perfil. 43
4.8 Carregamento dos dados de perfil do utilizador. 43
4.9 Atualização do perfil do utilizador. 44
4.10 Carregamento dos dados de pagamentos. 46
4.11 Carregamento das informações de reutilização. 47
4.12 Renderização dinâmica das músicas do utilizador. 49
4.13 Carregamento e reprodução do ficheiro áudio. 49
4.14 *Download* de contrato inteligente. 50
4.15 Obtenção das músicas públicas. 52
4.16 Extração e visualização dos metadados do ficheiro. 54
4.17 Configuração da ligação à *blockchain*. 54
4.18 Criação do contrato inteligente na *blockchain*. 54
4.19 Gestão dinâmica de co-titulares. 56
4.20 Iniciação do servidor Express e carregamento das rotas. 63
4.21 *Middleware* de autenticação JWT. 64
4.22 Modelo de dados do utilizador. 65
4.23 Processo de registo de um novo utilizador. 66
4.24 Autenticação do utilizador e emissão do *token* JWT. 67
4.25 Atualização do perfil de utilizador. 68
4.26 Estrutura das rotas de utilizador. 69
4.27 Modelo de dados da música. 70
4.28 Controlador básico de criação e listagem de músicas. 71

4.29 Configuração da ligação à *blockchain*. 76
4.30 *Deployment* do contrato `MusicRights` no *frontend*. 77
4.31 *Deployment* do contrato `MusicReuse`. 77
4.32 *Deployment* do contrato `MusicMix`. 78
A.1 Exportação e registo de uma *mix*. 106
A.2 Seleção e armazenamento do trecho reutilizado. 109
A.3 Criação do registo de reutilização. 112
B.1 Contrato `MusicRights.sol` 114
B.2 Contrato `MusicReuse.sol` 116
B.3 Contrato `MusicMix.sol` 118

Lista de Equações

4.1 Valor do pagamento da reutilização de um trecho musical. 57
5.1 Média das classificações SUS. 96

Lista de Acrónimos

ABI *Application Binary Interface*

API *Application Programming Interface*

AVCO *Audio Value Chain Ontology*

BLOB *Binary Large Object*

BSON *Binary* JSON

CEL *Contract Expression Language*

CMO *Collective Management Organization*

codec codificador/descodificador

CORS *Cross-Origin Resource Sharing*

CPU *Central Processing Unit*

CSS *Cascading Style Sheet*

DAO *Decentralized Autonomous Organization*

DeFi *Decentralized Finance*

DLT *Distributed Ledger Technology*

DOM *Document Object Model*

ECMA European Computer Manufacturers Association

ETH *Ether*

EVM Ethereum *Virtual Machine*

FLAC *Free Lossless Audio Codec*

HTML *HyperText Markup Language*

HTTP *HyperText Transfer Protocol*

IA Inteligência Artificial

IEC International Electrotechnical Commission

IMAF *Interactive Music Application Format*

IoT *Internet of Things*

IPFS *InterPlanetary File System*

ISEP Instituto Superior de Engenharia do Porto

ISO International Organization for Standardization

JSON JavaScript *Object Notation*

JWT JSON *Web Token*

MCO *Media Contract Ontology*

MEEC Mestrado em Engenharia Electrotécnica e Computadores

MIDI *Musical Instrument Digital Interface*

MP3 MPEG *Audio Layer* III

MPEG Moving Picture Experts Group

MVCO *Media Value Chain Ontology*

NFT *Non Fungible Token*

NoSQL *Not only* SQL

NPM Node Pack Manager

PDF *Portable Document Format*

QR *Quick Response*

RDF *Resource Description Framework*

REST *Representational State Transfer*

RPC *Remote Procedure Call*

SC4M *Smart Contracts for Media*

SHA-256 Secure Hash Algorithm-256 bits

SPI Sistemas e Planeamento Industrial

SUS *System Usability Scale*

URL *Uniform Resource Locator*

VNT *Value Network Technology*

WAV *Waveform Audio File Format*

XML *eXtended Markup Language*

Glossário

Smart Contract

Contrato inteligente que executa automaticamente as cláusulas acordadas quando determinadas condições são satisfeitas, sem necessidade de intermediários.

Blockchain

Tecnologia de registo distribuído que permite armazenar transações de forma imutável, transparente e descentralizada.

Ethereum

Plataforma de *blockchain* descentralizada que suporta a execução de contratos inteligentes e aplicações descentralizadas.

Express

Framework minimalista para Node.js que facilita a criação de servidores *web* e *Application Programming Interfaces* do tipo *Representational State Transfer*, permitindo a gestão de rotas e de pedidos *HyperText Transfer Protocol*.

Gas

Unidade de medida do custo computacional necessário para executar operações e transações na rede Ethereum.

GridFS

Sistema de armazenamento de ficheiros do MongoDB que permite guardar e recuperar ficheiros binários de grandes dimensões.

Hardhat

Ambiente de desenvolvimento de contratos inteligentes que permite compilar, testar e fazer o *deployment* de *smart contracts* em redes locais ou de teste.

HTML

HyperText Markup Language, linguagem de estruturação e formatação do conteúdo de páginas *web* baseada em *tags*, que são interpretadas pelo navegador.

IMAF

Interactive Music Application Format, especificação do Moving Picture Experts Group de representação e manipulação de conteúdos musicais interativos e reutilizáveis.

MongoDB

Base de dados *Not only* SQL orientada a documentos, utilizada para armazenar dados de forma flexível em formato *Binary* JSON.

Node Pack Manager

Gestor de pacotes para Node.js, usado para instalar e gerir dependências de projetos JavaScript.

Node.js

Ambiente de execução de JavaScript do lado do servidor que permite executar códigos JavaScript fora do navegador. Possui uma arquitetura assíncrona e orientada por eventos.

SC4M

Smart Contracts for Media, especificação da International Organization for Standardization/International Electrotechnical Commission de conversão de contratos digitais em contratos inteligentes executáveis na *blockchain*.

Solidity

Linguagem de programação utilizada para desenvolver contratos inteligentes que correm na rede Ethereum.

Capítulo 1

Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Tese/Dissertação (TEDI) no 2º semestre do 2º ano do Mestrado em Engenharia Electrotécnica e Computadores-Sistemas e Planeamento Industrial no Instituto Superior de Engenharia do Porto (ISEP).

1.1 Contextualização

A revolução digital transformou radicalmente a forma como a música é criada, distribuída e consumida. Com o aparecimento de plataformas de *streaming*, das redes sociais e de plataformas de criação musical, assistidas por Inteligência Artificial (IA), tornou o acesso à produção e divulgação musical cada vez mais acessível. Esta facilidade no acesso trouxe, também, outras consequências a este setor, principalmente no que toca à gestão de direitos de autor.

A fragmentação do mundo musical digital, a reutilização de obras em novos contextos e o crescimento de modelos colaborativos de criação levantam questões complexas sobre a autoria, a propriedade intelectual e a compensação financeira de criadores e intérpretes. Num contexto em que milhões de músicas são publicadas diariamente, torna-se inviável assegurar manualmente a identificação e uma remuneração justa de cada utilização.

Neste sentido, surgem tecnologias descentralizadas como uma possibilidade de reavaliar os modelos tradicionais da gestão de direitos de autor. A tecnologia *blockchain*, pela sua natureza imutável e distribuída, possibilita a criação de registos

digitais permanentes de autoria e licenciamento. Em conjunto com os *smart contracts*, estes sistemas permitem automatizar a execução de contratos e a distribuição de receitas, garantindo maior transparência, segurança e eficiência.

A presente dissertação enquadra-se neste domínio emergente e propõe o desenvolvimento de uma plataforma denominada SoundSlice, que combina conceitos de *blockchain*, *smart contracts* e gestão de direitos digitais. O principal objetivo não é apenas demonstrar a viabilidade técnica da solução, mas sobretudo analisar de que forma a tecnologia *blockchain* e os contratos inteligentes podem beneficiar o processo de gestão de direitos de autor, assegurando uma gestão imutável, transparente e automaticamente compensatória para os criadores.

Neste sentido, o trabalho procura compreender o impacto que estas tecnologias podem ter na automatização dos processos de licenciamento, na distribuição proporcional de receitas e na eliminação de intermediários, explorando a integração dos padrões *Smart Contracts for Media* (SC4M) e *Interactive Music Application Format* (IMAF) do Moving Picture Experts Group (MPEG) como base tecnológica para a implementação e validação da proposta.

1.2 Definição do Problema

O modelo tradicional de gestão de direitos musicais apresenta limitações significativas face às exigências da economia digital. A identificação e remuneração de utilizações parciais de obras é, atualmente, um processo moroso e pouco transparente, que depende fortemente de intermediários e da confiança em entidades centralizadas.

Na prática, este modelo leva frequentemente a atrasos nos pagamentos, discrepâncias entre as utilizações reais e as receitas distribuídas, e à exclusão de pequenos criadores que não dispõem de mecanismos eficazes para gerir os seus direitos. Além disso, a ausência de um sistema universal de identificação e registo de obras dificulta a interoperabilidade entre plataformas e aumenta o risco de conflitos de propriedade intelectual.

O problema central desta dissertação consiste, portanto, em desenvolver uma solução tecnológica que permita a gestão automatizada de direitos musicais, assegurando a rastreabilidade, transparência e remuneração proporcional de todas as utilizações de conteúdo, independentemente da sua complexidade ou origem.

Na Figura 1.1 ilustra-se de forma esquemática esta problemática. Músicos, criadores e compositores disponibilizam as suas obras em diferentes plataformas digitais, que atuam como intermediárias na divulgação e monetização dos conteúdos. Contudo, a ausência de mecanismos automáticos de rastreamento e validação de direitos conduz frequentemente à reutilização indevida de obras musicais, à utilização não remunerada de excertos e à perda de rastreabilidade sobre a autoria original.

Este cenário evidencia a dificuldade em assegurar uma remuneração justa e proporcional para todos os intervenientes no processo criativo, comprometendo a transparência, a eficiência e a equidade do ecossistema musical digital.

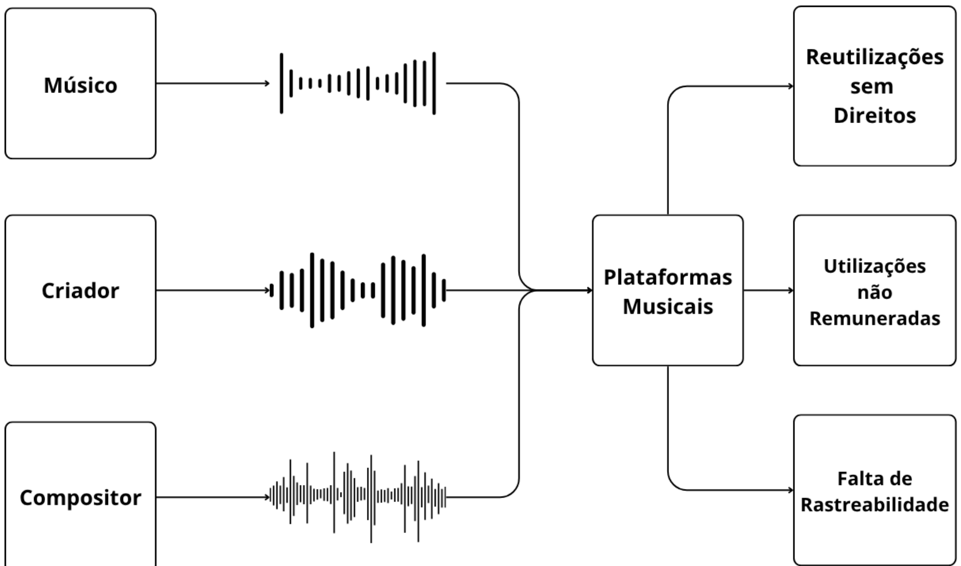


Figura 1.1: Problema atual na gestão de direitos musicais.

A resposta a este problema requer uma abordagem multidisciplinar, combinando conceitos de engenharia de software, tecnologias *blockchain*, contratos inteligentes e processamento de áudio.

1.2.1 Objetivos

O objetivo geral deste trabalho é conceber e implementar uma plataforma descentralizada para a gestão automatizada de direitos musicais, capaz de executar contratos inteligentes que assegurem a compensação financeira proporcional dos titulares de direitos sempre que uma obra é reutilizada.

De forma mais detalhada, os objetivos específicos incluem:

- Analisar o estado da arte sobre *blockchain*, *Smart Contract*, gestão de direitos digitais e reutilização de conteúdos musicais;
- Avaliar o potencial dos pacotes SC4M e IMAF e propor um modelo de integração entre ambos;
- Desenvolver a plataforma SoundSlice, composta por um *frontend web* e um *backend* em Node.js com ligação a uma rede *blockchain*;
- Implementar funcionalidades de registo, *upload*, reutilização de obras e criação de *mixes*, com cálculo automático do valor a pagar com base na percentagem reutilizada;

- Simular o acionamento de contratos inteligentes para efetuar pagamentos automáticos na rede;
- Avaliar a solução em termos de desempenho, usabilidade e aplicabilidade prática no contexto musical.

1.2.2 Resultados esperados

Pretende-se demonstrar que é possível automatizar a gestão de direitos musicais através de uma plataforma baseada em contratos inteligentes, capaz de garantir um modelo justo, transparente e auditável de distribuição de receitas.

Os principais resultados esperados são:

- A criação de uma prova de conceito funcional da plataforma SoundSlice;
- A integração dos ideais dos pacotes SC4M e IMAF para registo e identificação de obras musicais;
- A implementação de contratos inteligentes que automatizem o processo de licenciamento e remuneração;
- A disponibilização de um sistema que permita criar *mixes* com reutilizações e músicas já criadas e calcular automaticamente a compensação proporcional dos autores;
- A validação experimental através de testes funcionais, de desempenho e de usabilidade;
- A demonstração do potencial da solução para ser aplicada em contextos reais de gestão de direitos musicais.

1.3 Plano de Trabalho

O plano de trabalho da dissertação foi estruturado em seis fases principais, permitindo uma progressão lógica desde a investigação teórica até à validação prática da solução:

Fase 1 – Revisão do Estado da Arte dedicada ao estudo aprofundado das tecnologias relacionadas com a gestão de direitos digitais, *blockchain* e contratos inteligentes, bem como análise das soluções IMAF e SC4M;

Fase 2 – Análise e Conceção centrada na definição dos requisitos funcionais e não funcionais da plataforma, modelação da arquitetura e identificação das interfaces entre componentes;

Fase 3 – Desenvolvimento do *Backend* focada na implementação da *Application Programming Interface* (API) Node.js/Express e na integração com a rede *blockchain* e com a base de dados MongoDB;

Fase 4 – Desenvolvimento do *Frontend* devotada à criação da interface *web* SoundSlice em *HyperText Markup Language* (HTML), Tailwind *Cascading Style Sheet* (CSS) e JavaScript, assegurando consistência visual e usabilidade;

Fase 5 – Integração e Testes dedicada à interligação do *frontend*, *backend* e da camada *on-chain*, seguida de testes funcionais, de desempenho e de usabilidade;

Fase 6 – Avaliação e Documentação de análise dos resultados, discussão crítica e elaboração do relatório final.

1.4 Organização da Dissertação

Este documento encontra-se estruturado em seis capítulos:

Capítulo 1 — Introdução apresenta o contexto, a problemática, os objetivos e o plano de trabalho da dissertação;

Capítulo 2 — Estado da Arte discute os conceitos fundamentais, as tecnologias relacionadas e os trabalhos correlacionados nas áreas de *blockchain*, *smart contracts*, IMAF e SC4M;

Capítulo 3 — Proposta de Solução descreve a arquitetura e o modelo conceptual da plataforma SoundSlice;

Capítulo 4 — Implementação e Integração detalha o desenvolvimento prático da solução, as tecnologias utilizadas e os desafios técnicos superados;

Capítulo 5 — Avaliação e Resultados apresenta os testes efetuados, a análise dos resultados obtidos;

Capítulo 6 — Conclusões apresenta as conclusões gerais e sugestões para trabalhos futuros.

Capítulo 2

Estado da Arte

Este capítulo apresenta o enquadramento teórico da gestão de direitos de autor baseada em *blockchain* e contratos inteligentes. Explora a evolução e aplicações dos *smart contracts*, os padrões SC4M e IMAF do MPEG, e analisa trabalhos relacionados no domínio musical. Por fim, identifica as limitações das soluções existentes e destaca o contributo inovador da proposta.

2.1 Introdução

O mundo musical sofreu uma transformação radical com a revolução digital. A criação, disseminação e monetização dos conteúdos musicais constituem desafios sem precedentes para a gestão dos direitos de autor. O surgimento de plataformas de *streaming* e de criação, manipulação e distribuição baseadas em Inteligência Artificial (IA) tornou possível a disseminação massiva da música, aumentando a complexidade do rastreamento e da correta compensação financeira dos criadores.

Estas alterações transformaram radicalmente o paradigma da gestão dos direitos de autor. Tradicionalmente, os direitos de autor e o consequente reconhecimento e remuneração era realizado de forma manual pelos próprios criadores. A música manipulada por Inteligência Artificial (IA), frequentemente resulta de reutilizações criativas de fragmentos de obras existentes, que podem estar sujeitos a direitos de autor. Assim, a criação de música, imagens e vídeo através da IA a partir de dados de treino obtidos sem autorização dos autores podem constituir uma violação indireta dos direitos, mesmo não existindo um autor associado a este tipo de conteúdos.

Neste contexto, os *smart contracts*, uma tecnologia *blockchain*, emergem como uma solução para garantir a transparência, automatização e equidade na distribuição destes contratos.

2.2 *Smart Contracts*

Um *smart contract* ou contrato inteligente é um programa que reside na *blockchain* e executa automaticamente termos e condições predefinidos quando determinados requisitos de um acordo ou contrato são cumpridos. Uma vez completadas, as transações são rastreáveis e irreversíveis [1].

2.2.1 Origem e Evolução

O conceito de *smart contract* foi introduzido por volta de 1993 pelo cientista da computação Nick Szabo, que os definiu como um conjunto de promessas virtuais com alguns protocolos associados para fazer com que sejam cumpridos, que eliminariam o uso de intermediários, garantindo uma execução segura baseada apenas na lógica programada [2]. Szabo usou uma analogia bastante prática para explicar este conceito: a máquina de venda automática. Quando alguém insere uma moeda e escolhe um produto, a máquina entrega automaticamente esse produto — tudo sem intervenção humana. Ele via esse processo como um exemplo de contrato autoexecutável, em que ambas as partes confiam na lógica embutida na máquina. Szabo acreditava que este tipo de lógica poderia ser aplicada a vários contextos — desde pagamentos automáticos até aluguer de carros que se desativariam sozinhos caso os pagamentos falhassem [3].

Antes da *blockchain*, entre 1994 e 2008, apesar do potencial revolucionário, os *smart contracts* permaneceram, durante muitos anos, como um conceito teórico. A razão principal era a ausência de uma infraestrutura tecnológica capaz de os suportar de forma segura e descentralizada e a execução de contratos digitais dependia de servidores centralizados — sujeitos a manipulação e falhas. Além disso, as ferramentas de criptografia ainda estavam a dar os primeiros passos e eram complexas de implementar. Durante esse período, outras ideias ligadas ao mundo das moedas digitais também começaram a surgir. Nick Szabo, por exemplo, propôs o “Bit Gold”, um sistema de moeda digital descentralizada que muitos consideram um precursor direto do Bitcoin [4]. No entanto, mais uma vez, faltava a infraestrutura certa para que essas ideias se tornassem realidade. Tudo começou a mudar em 2009, com o lançamento do Bitcoin por Satoshi Nakamoto. Embora o Bitcoin trouxesse um sistema de *blockchain* funcional, as suas capacidades de *smart contracts* eram bastante limitadas. Foi só em 2015, com o lançamento da plataforma Ethereum por Vitalik Buterin [5], que os *smart contracts* se tornaram uma realidade. O Ethereum foi o primeiro sistema criado especificamente para executar contratos

inteligentes, com uma linguagem própria de programação Solidity e um ambiente de execução chamado Ethereum *Virtual Machine* (EVM) [6]. Atualmente, existem múltiplas plataformas como a Solana[1], Cardano[2] e Tezos[3] que pretendem melhorar a escalabilidade, segurança e eficiência dos contratos inteligentes.

Desde então, os *smart contracts* passaram a ser amplamente utilizados em diversas áreas como *Decentralized Finance* (DeFi), *Non Fungible Tokens* (NFTs), jogos, identidade digital ou gestão de direitos de autor. Assim, a ideia lançada por Szabo nos anos 90, que parecia futurista e fora do alcance tecnológico da altura, acabou por se tornar um dos pilares da revolução digital do século XXI, graças ao avanço das *blockchains* e das criptomoedas. Os contratos inteligentes são executados na *blockchain*, o que implica que os termos são armazenados numa base de dados distribuída e não podem ser modificados. As transações também são processadas pela *blockchain*, o que automatiza pagamentos e contrapartidas.

2.2.2 Aplicações

A partir do momento em que começaram a ser aplicados, os *smart contracts* passaram a ser utilizados em várias áreas. A capacidade que estes contratos têm de executar automaticamente termos acordados, sem a necessidade de um intermediário, tornou-os extremamente úteis numa vasta gama de áreas:

***Decentralized Finance* (DeFi)** utilizam os *smart contracts* para substituir instituições financeiras tradicionais, permitindo a realização de operações como empréstimos, trocas de ativos, criação de instrumentos financeiros e obtenção de juros. Certas plataformas, como a Uniswap[4] e Aave[5], operam exclusivamente através de contratos inteligente, assegurando que a execução das transações decorre segundo as regras predefinidas [7].

Jogos utilizam os contratos inteligentes para desenvolver ecossistemas digitais onde os jogadores possuem e controlam os ativos que adquirem ou conquistam no jogo. Personagens, equipamentos ou moedas virtuais podem ser tokenizados e negociados livremente entre os usuários, o que permite interações entre jogos distintos. Este sistema possibilita a criação de um ecossistema económico descentralizado, além de incentivar novos modelos de monetização para os jogadores e criadores de conteúdo [8].

Identidade Digital recorre aos contratos inteligentes para permitir que os utilizadores controlem os seus dados. Com tecnologia *blockchain*, é possível construir

[1] `https://solana.com/`
[2] `https://cardano.org/`
[3] `https://tezos.com/`
[4] `https://app.uniswap.org/`
[5] `https://aave.com/`

identidades digitais seguras e que podem ser verificadas, com as quais o próprio utilizador controla toda a informação que decide partilhar. Isso permite, por exemplo, a execução de processos de autenticação onde não são expostos dados sensíveis aumentando a privacidade e a proteção contra fraudes [9].

Cadeia de Abastecimento e Logística usa os contratos inteligentes para acompanhar o percurso de produtos desde a origem até ao consumidor final. A possibilidade de integração com sensores de *Internet of Things* (IoT) e sistemas de monitorização permitem a automatização de ações como a verificação de entregas, validação de condições de transporte, e o pagamento em etapas. Este procedimento ajuda num nível maior de transparência e confiança entre os *stakeholders* da cadeia de valor, ao mesmo tempo reduzem a probabilidade de fraudes ou até mesmo perda [10].

Imobiliário e Notariado utilizam os contratos inteligentes para simplificar processos administrativos, incluindo a aceleração de transações. Por exemplo, o registo automático da transferência de propriedade ocorreria imediatamente após a confirmação do pagamento. Além disso, as regras para verificação de identidade e assinaturas digitais podem ser incorporadas, tornando ainda mais ágil o tempo e os recursos necessários para a venda e compra de propriedades, bem como a execução de escrituras notariais [11].

Votação Eletrónica adota contratos inteligentes para criar sistemas de votação imutáveis, transparentes e verificáveis. Esses sistemas garantem que cada voto dado seja único, anónimo e capturado com precisão dentro da *blockchain*. Esse tipo de solução é especialmente relevante para eleições internas organizacionais, como no caso de *Decentralized Autonomous Organizations* (DAOs) [12].

Gestão de Direitos de Autor é um processo bastante complexo e moroso, que tradicionalmente recorre a terceiros para assegurar todos os serviços, como por exemplo, as sociedades de gestão coletiva, as editoras ou as plataformas de distribuição digital. Esta situação tem vindo a mudar com a adoção de novas tecnologias, como a *blockchain* e os *smart contracts*, nomeadamente no domínio da multimédia através dos padrões *Smart Contracts for Media* [13] e *Interactive Music Application Format* [14] do MPEG.

2.3 Gestão Autoral Multimédia

Os direitos, bem como as receitas inerentes, poderão ser atribuídos, controlados e distribuídos de forma automatizada, através de *smart contracts*. Com esta tecnologia é possível estabelecer de forma perene e pública os termos de utilização de obras criativas, como músicas, vídeos ou imagens, e assegurar que os autores ou titulares

de direitos são pagos automaticamente, proporcionalmente à sua parte, de forma a remunerá-los sempre que a obra seja utilizada ou negociada. Isto resulta numa menor dependência dos intermediários, acelera os processos de pagamento e, por consequência, aumenta a confiança entre as partes [15].

Um contrato inteligente pode, ainda, codificar regras relativas à redistribuição de receitas. Tomando como exemplo a obra musical que reutiliza pedaços de outras obras, um *smart contract* pode ser programado para dividir automaticamente os lucros entre todos os autores e executantes. Assim, os contratos inteligentes revelam-se uma ferramenta poderosa para modernizar a gestão dos direitos de autor, proporcionando um sistema mais justo, transparente e alinhado com a realidade digital em constante evolução.

2.3.1 MPEG

O Moving Picture Experts Group (MPEG) é um grupo de trabalho da International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) responsável pelo desenvolvimento de padrões internacionais para compressão, descompressão, processamento e representação codificada de imagens em movimento, áudio e suas combinações [16]. Neste contexto, MPEG tem desenvolvido uma série de padrões e especificações para permitir uma gestão de direitos de autor mais eficiente e descentralizada no setor de medias digitais. Embora o MPEG não seja um criador de contratos inteligentes, os seus padrões constituem uma fundação sólida para a implementação de contratos inteligentes no domínio dos conteúdos multimédia.

A complexidade crescente associada à gestão de direitos de autor no universo digital, especialmente num contexto em que a reutilização e a remistura de conteúdos são práticas comuns, tem levado ao desenvolvimento de soluções tecnológicas que permitem garantir uma gestão transparente, automática e eficiente dos direitos dos criadores. Neste cenário, destacam-se dois padrões propostos pelo MPEG: *Smart Contracts for Media* (SC4M) e o *Interactive Music Application Format* (IMAF). Ambas integram a arquitetura de normalização internacional da série ISO/IEC, oferecendo abordagens complementares para a gestão descentralizada de conteúdos multimédia e respetivos direitos de utilização e remuneração.

Estas tecnologias permitem o rastreio do uso de conteúdos digitais e a execução automática de licenças e pagamentos, promovendo maior eficiência, transparência e respeito pelos direitos dos criadores [15, 17, 18].

2.3.2 *Smart Contracts for Media*

O padrão *Smart Contracts for Media* (SC4M) definido pela norma ISO/IEC 21000-23, visa transformar contratos digitais expressos em *eXtended Markup Language*

(XML) e em ontologias *Resource Description Framework* (RDF),baseados nas normas do MPEG-21, em contratos inteligentes executáveis em tecnologias de registo distribuído nomeadamente *blockchain* e *Distributed Ledger Technology* (DLT). O principal objetivo deste padrão consiste em permitir a automação de aspetos contratuais relacionados com a distribuição e utilização de conteúdos digitais, nomeadamente o pagamento de *royalties*, a aplicação de permissões e restrições, e a garantia de transparência e auditabilidade na gestão de direitos.

O SC4M recorre a um conjunto de ontologias e linguagens definidas pelo MPEG para representar tanto os conteúdos como os agentes e as ações associadas. Entre estas ontologias destacam-se a *Media Value Chain Ontology* (MVCO), que modela obras, utilizadores e atividades; a *Audio Value Chain Ontology* (AVCO), com foco em conteúdos áudio; e a *Media Contract Ontology* (MCO), que representa contratos, partes envolvidas, fluxos de pagamento e notificações. De forma complementar, a linguagem *Contract Expression Language* (CEL) permite codificar cláusulas contratuais em XML com base lógica deôntica, ou seja, incluindo permissões, obrigações e proibições.

A conversão entre contratos digitais MPEG-21 e contratos inteligentes pode seguir dois fluxos. A conversão direta (*forward*) transforma um contrato descrito em CEL/MCO num contrato inteligente específico para uma DLT, gerando automaticamente o código em linguagens como Solidity (Ethereum), Michelson (Tezos) ou TEAL (Algorand). Já a conversão inversa (*backward*) permite reconstruir o contrato original a partir do *smart contract* implementado, assegurando a equivalência entre o texto técnico e a versão contratual compreensível para os utilizadores.

Uma das funcionalidades centrais do SC4M é a associação entre cláusulas contratuais narrativas e as respetivas cláusulas codificadas nos *smart contracts.* Este alinhamento semântico e técnico é fundamental para garantir que os termos jurídicos acordados entre as partes são efetivamente os que serão executados de forma automática pela tecnologia de registo distribuído. Para apoiar a adoção do SC4M, o grupo MPEG disponibiliza software de referência, implementado em XML/Python e RDF/JavaScript, que permite criar contratos, gerar e validar código de contratos inteligentes, e realizar testes de conformidade com casos reais, como o *streaming* de conteúdos ou a venda de ativos digitais.

O SC4M representa um passo importante para aproximar a formalização jurídica da execução automática em contextos descentralizados, promovendo uma gestão de direitos mais justa, transparente e interoperável. Esta norma é particularmente relevante no contexto atual da indústria dos media, que enfrenta desafios relacionados com a monetização e a rastreabilidade da utilização de conteúdos num ecossistema digital em constante evolução. A Figura 2.1 representa a estrutura dos SC4M.

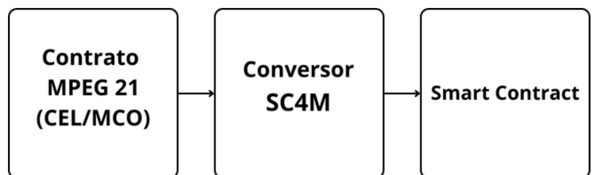


Figura 2.1: Estrutura dos SC4M.

2.3.3 *Interactive Music Application Format*

O *Interactive Music Application Format* (IMAF) foi padronizado pelo MPEG no âmbito da norma ISO/IEC 23000-23. Este formato, que foi concebido para suportar conteúdos musicais interativos e reutilizáveis, visa facilitar a criação, edição, reutilização e rastreamento de obras musicais — algo particularmente relevante num contexto onde os conteúdos são remisturados e reutilizados com frequência.

No contexto da reutilização criativa e da manipulação de conteúdos musicais, o IMAF oferece uma estrutura para organizar e descrever fragmentos musicais que podem ser combinados de forma dinâmica, com base nas escolhas do utilizador ou em regras predefinidas [19]. Esta capacidade torna o IMAF particularmente relevante para aplicações de música generativa, experiências imersivas e plataformas de criação colaborativa. O IMAF pode ser integrado em várias componentes:

- Unidades Musicais - Representações segmentadas de conteúdos musicais, que podem incluir ficheiros de áudio, MIDI, ou eventos musicais anotados.
- Descritores Interativos - Informações que permitem descrever as condições de ativação de certas unidades musicais com base em interações do utilizador.
- Mapeamento Temporal - Permite sincronizar eventos musicais entre diferentes camadas e formatos.
- Metadados - Informação sobre direitos, autores, estrutura rítmica/harmónica, ou *tags* de estilo musical.
- Regras de Composição Dinâmica - Condições e lógica que determinam a progressão musical de forma não-determinística ou ramificada.

O IMAF é, ainda, frequentemente proposto em conjunto com mecanismos de proteção de direitos, designadamente os definidos no MPEG-21 ou os *Smart Contracts for Media* (SC4M). A sinergia entre estes dois padrões pretende garantir que cada entidade musical carregue os seus próprios termos de licenciamento, permitindo ao sistema verificar automaticamente a permissão da reutilização dos conteúdos. A Figura 2.2 representa a estrutura do IMAF.

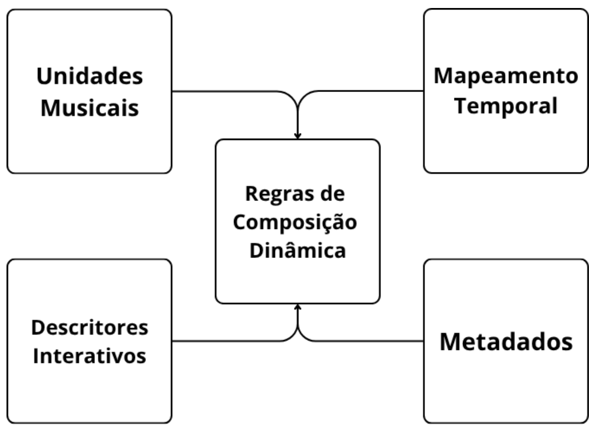


Figura 2.2: Estrutura do IMAF.

2.4 Trabalhos Relacionados

Diversos trabalhos têm explorado a integração de tecnologias descentralizadas, como *blockchain* e contratos inteligentes, no domínio da gestão de direitos autorais. Nesta secção são analisadas propostas representativas, com o objetivo de identificar as suas principais contribuições e limitações, realçando de que forma a presente dissertação difere e introduz um contributo inovador.

2.4.1 Tiny Human and the Future of Music Licensing on Ethereum

Este trabalho [20] documenta a primeira aplicação de contratos inteligentes à distribuição automática de *royalties*. Em 2016, a artista Imogen Heap, em parceria com a plataforma Ujo Music, lançou a música Tiny Human utilizando um contrato inteligente Ethereum para distribuir automaticamente as receitas entre todos os colaboradores da obra. O projeto registou mais de 8500 transações com latência média de 16 s, demonstrando a viabilidade de um sistema de licenciamento transparente e descentralizado. A solução *open source* implementada baseia-se exclusivamente em contratos inteligentes, embora apresente limitações ao nível dos custos de Gas e da escalabilidade da rede subjacente.

2.4.2 The Impact of Digital Innovation and Blockchain on the Music Industry

Este relatório [21] de 2017 apresenta uma análise económica do impacto das tecnologias emergentes na indústria musical. Os autores destacam a *blockchain* e os contratos inteligentes como ferramentas para aumentar a transparência e reduzir os custos operacionais. Embora não inclua uma implementação técnica, o relatório

identifica áreas de transformação, como a rastreabilidade das execuções e os pagamentos diretos aos artistas. Defende ainda a necessidade de uma regulamentação adaptativa e de sustentabilidade financeira no ecossistema digital.

2.4.3 Blockchain and Smart Contracts: The Missing Link in Copyright Licensing

Este documento [22] de 2018 analisa o potencial da *blockchain* para transformar o licenciamento de direitos de autor em múltiplos domínios — incluindo texto, imagem e música — e identifica os contratos inteligentes como mecanismos promissores para automatizar transações e assegurar a rastreabilidade dos conteúdos. Analisa ainda os problemas jurídicos da imutabilidade, os riscos de erro nos registos e a ausência de um enquadramento legal robusto para contratos autoexecutáveis. Embora o trabalho seja de natureza teórica e não apresente métricas de desempenho ou código aberto, reforça a importância de integrar tecnologia com políticas de proteção de criadores, introduzindo a noção de sustentabilidade regulatória.

2.4.4 Consortium Blockchain Smart Contracts for Musical Rights Governance in a *Collective Management Organization*s Use Case

Este trabalho [23] de 2020 propõe a utilização de uma rede *blockchain* como infraestrutura para a gestão coletiva de direitos musicais através de um consórcio. A rede permissionada é composta por nós validados, representando editoras, artistas e *Collective Management Organization*s (CMOs), que recorrem a contratos inteligentes para registar licenças, rastrear utilizações e distribuir *royalties* automaticamente. Adota o mecanismo de consenso *Proof-of-Authority*, obtendo uma validação média de 1,9 s e reduzindo 47 % do consumo energético face a redes *Proof-of-Work*.

2.4.5 Fair Rewarding Mechanism in Music Industry Using Smart Contracts on Public-Permissionless Blockchain

Este artigo [24], datado de 2022, propõe um sistema de recompensa justa para artistas baseado em contratos inteligentes na rede *Ethereum* pública. O modelo distribui automaticamente receitas entre as partes envolvidas com tempo médio de confirmação de 12 s e custo médio de 0,0004 ETH por transação. Os autores discutem ainda a migração para mecanismos de consenso mais eficientes, designadamente *Proof-of-Stake*), como estratégia de sustentabilidade ambiental face ao elevado consumo energético do *Proof-of-Work*.

2.4.6 Smart Royalties: Blockchain Solutions for Music Metadata and Copyright

Este estudo [25] de 2023 apresenta uma arquitetura baseada em *blockchain* para gerir direitos musicais. A proposta procura resolver discrepâncias nos metadados que afetam a distribuição de *royalties*, utilizando contratos inteligentes para automatizar o processo de pagamento. Adota um modelo híbrido composto por *sidechains* interligadas com entidades de licenciamento e por NFTs representando obras musicais únicas. Os autores defendem que a descentralização pode reduzir custos e aumentar a eficiência da gestão de direitos de autor. Não apresentam métricas de desempenho, mas analisam os desafios da interoperabilidade internacional e das normas de dados musicais.

2.4.7 Application of Blockchain Technology in Digital Music Copyright Management: A Case Study of the VNT Chain Platform

Este trabalho [26] de 2024 reporta uma implementação híbrida (*on-chain*/*off-chain*) para gestão de direitos musicais. A solução combina a *blockchain Value Network Technology* (VNT) com armazenamento descentralizado baseado em *InterPlanetary File System* (IPFS) e algoritmos de *fingerprinting* acústico para verificação de originalidade. As transações contratuais são executadas na cadeia, enquanto os ficheiros de áudio e metadados são processados externamente, otimizando desempenho. Apresenta uma latência média de 1,87 s e uma taxa de erro inferior a 2 na identificação de semelhanças musicais.

2.4.8 Decentralized Music Marketplace: A Blockchain-Based Rights Management Approach

Em 2024, Mendonca et al. propõem a criação de um mercado musical descentralizado baseado em *tokens* [27]. Cada música é representada por um *token* que pode ser comercializado entre utilizadores, permitindo a divisão de propriedade e a compensação automática via contratos inteligentes. A solução alcançou latência média de 4,2 s e consumo de 52 000 unidades de *gas* por transação, demonstrando a viabilidade do modelo.

2.4.9 Comparação com a SoundSlice.

De forma geral, os trabalhos analisados demonstram o potencial da tecnologia *blockchain* e dos contratos inteligentes para aumentar a transparência e automatizar a gestão de direitos autorais. No entanto, a maioria das propostas existentes concentra-se

na distribuição de receitas ou no registo de obras completas, não abordando a reutilização parcial nem o cálculo proporcional de direitos. A plataforma SoundSlice diferencia-se por integrar uma abordagem mais completa e prática, que combina a análise automática de reutilização musical com a execução contratual proporcional, assegurando uma remuneração justa e transparente para todos os titulares. Além disso, adota uma arquitetura híbrida *on-chain/off-chain* que otimiza o desempenho e a sustentabilidade do sistema, mantendo a rastreabilidade das transações e a integridade dos metadados. Esta integração multidimensional permite à SoundSlice superar as limitações das soluções anteriores, apresentando-se como uma proposta inovadora, escalável e centrada no criador, capaz de responder aos desafios contemporâneos da economia musical digital. Por último, a atual proposta adota os padrões *Smart Contracts for Media* (SC4M) e *Interactive Music Application Format* (IMAF) do MPEG, alinhando-se com as melhores práticas e recomendações internacionais para o desenvolvimento de sistemas de gestão de direitos de autor aplicáveis a conteúdos musicais. Na Tabela 2.1 estão representadas as principais semelhanças e diferenças dos trabalhos relacionados relativamente à proposta.

Trabalho	**Semelhanças**	**Diferenças / Contributos**
Tiny Human (Heap, 2016)	Usa *blockchain* para automatizar a distribuição de receitas e eliminar intermediários.	Introduz apenas a venda de obras completas; a presente dissertação acrescenta reutilização parcial, criação de *mixes* e cálculo proporcional de direitos.
The Impact of Digital Innovation (De Leon et al., 2017)	Visa uma indústria musical mais justa através da automação e descentralização.	Proposta conceptual; a presente dissertação inclui implementação funcional centrada na reutilização e mistura de trechos musicais.
Blockchain and Smart Contracts (Bodo et al., 2018)	Propõe a adoção de contratos inteligentes para licenciamento automatizado.	Trabalho teórico; a presente dissertação concretiza uma integração prática e avaliada.
Consortium Blockchain for CMO Governance (Kapsoulis et al., 2020)	Utiliza contratos inteligentes para registo e distribuição de direitos.	Foco em governação institucional de grandes entidades; a presente dissertação centra-se no criador individual e na reutilização descentralizada.
Fair Rewarding Mechanism (Halgamuge et al., 2022)	Usa contratos inteligentes para transparência e distribuição proporcional de receitas.	Foco em vendas diretas de obras completas; a dissertação aborda reutilizações parciais e *mixes*.
Smart Royalties (Sharp et al., 2023)	Automatiza remuneração autoral via contratos inteligentes.	Não contempla reutilização de conteúdos nem remuneração proporcional.
VNT Chain Platform (Shi et al., 2024)	Processamento híbrido *on-chain/off-chain* para gestão de direitos.	Foco em verificação de obras originais; a dissertação expande para gestão de reutilização e fragmentação musical.
Decentralized Music Marketplace (Mendonça et al., 2024)	Emprega contratos inteligentes e *tokens* de propriedade.	Limita-se à compra e venda; a dissertação introduz o mecanismo de reutilização proporcional.

Tabela 2.1: Comparação geral entre trabalhos relacionados e a proposta SoundSlice.

2.5 Sumário

Este capítulo apresentou o enquadramento teórico e tecnológico da gestão autoral baseada em *blockchain* e contratos inteligentes, destacando os padrões SC4M e IMAF, bem como diversos trabalhos congéneres. A análise evidenciou que, apesar dos avanços existentes, continuam a faltar soluções que suportem a reutilização parcial de obras e o cálculo proporcional de direitos e implementem os padrões MPEG do domínio.

O capítulo seguinte apresenta a plataforma SoundSlice, que foi desenhada para colmatar estas lacunas.

Capítulo 3

Proposta de Solução

Neste capítulo são apresentadas as principais ferramentas, tecnologias e *frameworks* que serão utilizados no desenvolvimento da solução proposta. O objetivo é contextualizar a sua seleção, destacando as funcionalidades mais relevantes para a implementação prática da plataforma SoundSlice.

3.1 Introdução

A plataforma SoundSlice foi concebida com o objetivo de permitir o registo, a reutilização, a criação de *mixes* e a remuneração automática de obras musicais, recorrendo à integração entre contratos inteligentes e uma infraestrutura de dados centralizada. O nome SoundSlice resulta da combinação das palavras *Sound* (som) e *Slice* (fatia), refletindo o conceito de dividir uma obra musical em partes reutilizáveis. Este nome simboliza a capacidade da plataforma de selecionar trechos específicos de uma composição (*slices*) e associar-lhes contratos inteligentes que garantem a atribuição de créditos e compensações de forma automática e proporcional. A SoundSlice assume-se como um protótipo funcional que demonstra como tecnologias descentralizadas e padrões podem ser aplicados na gestão de direitos de autor no setor musical.

3.2 Análise de Requisitos

A definição clara dos requisitos é essencial para compreender as funcionalidades do sistema e garantir que a solução desenvolvida responde às necessidades dos seus

utilizadores. Nesta fase, são identificados os atores envolvidos e os principais casos de uso que orientam o desenho da plataforma.

3.2.1 Atores do Sistema

Os atores representam todas as entidades externas que interagem com a aplicação, podendo ser pessoas ou sistemas externos que comunicam com o núcleo da plataforma.

Atores Humanos

- **Utilizador Criador** — Pessoa responsável por registar as suas obras originais na plataforma, submetendo ficheiros de áudio. É o titular inicial dos direitos de autor e recebe automaticamente a remuneração sempre que a sua obra é reutilizada.
- **Utilizador Reutilizador** — Pessoa que reutiliza trechos musicais de obras já registadas, criando novas composições ou *mixes*. É responsável pelo pagamento proporcional aos autores originais, através da execução dos contratos inteligentes.

Atores Técnicos

- *Blockchain* Ethereum — Rede descentralizada responsável pela execução dos contratos inteligentes `MusicRights`, `MusicReuse` e `MusicMix`. Garante a imutabilidade, rastreabilidade e transparência das transações.
- Carteira Digital (MetaMask) — Ferramenta que permite aos utilizadores assinar transações e interagir com a *blockchain* de forma segura.
- Sistema de Armazenamento (MongoDB/GridFS) — Componente de persistência de dados que comunica com a aplicação para armazenar metadados e ficheiros de áudio, embora não faça parte da camada de interação direta com o utilizador.

3.2.2 Casos de Uso Principais

Os casos de uso fundamentais da plataforma SoundSlice são apresentados a seguir. Cada caso representa uma funcionalidade central do sistema e está associado a um ou mais atores.

- Registo de Utilizador — Permite o registo e autenticação de criadores e reutilizadores.

- Submissão de Música Original — Envolve o *upload* de um ficheiro, extração de metadados e criação do contrato `MusicRights`.
- Análise de Reutilização — O sistema reutiliza um ficheiro já inserido, calculando a percentagem de reutilização e o valor devido.
- Criação de *Mixes* — Combina vários trechos de áudio para gerar uma nova composição e aciona o contrato `MusicMix`.
- Remuneração Automática — Distribui os valores de *royalties* entre os titulares de direitos através do contrato `MusicReuse`.
- Consulta de Contratos e Metadados — Permite visualizar o histórico de obras, transações e contratos criados, tanto na base de dados como na *blockchain*.

3.3 Arquitetura

A SoundSlice foi concebida de forma modular, integrando componentes interligados que asseguram a gestão completa do ciclo de vida de uma música — desde o registo inicial até à reutilização e remuneração automática dos titulares de direitos.

O sistema está dividido em quatro grandes componentes:

Frontend – Responsável pela interface de interação com o utilizador. Foi desenvolvido em HTML5, CSS3, Tailwind CSS e JavaScript, e permite aos artistas e produtores efetuar operações como o registo de novas músicas, o upload de ficheiros, a análise de reutilizações, criação de *mixes* e a consulta dos contratos inteligentes associados. As páginas foram desenhadas para serem responsivas e intuitivas, proporcionando uma experiência de utilização coerente e acessível.

Backend – Constitui o núcleo lógico da aplicação e foi implementado com Node.js e o *framework* Express. É responsável por gerir a comunicação entre o *frontend*, a base de dados e a *blockchain*, disponibilizando uma API do tipo *Representational State Transfer* (REST) que processa pedidos *HyperText Transfer Protocol* (HTTP) com *payload* em JavaScript *Object Notation* (JSON). Nesta camada estão também implementadas as principais lógicas de negócio, como a validação de utilizadores, o cálculo de compensações e o registo de transações *on-chain*.

Base de Dados MongoDB – Permite armazenar de forma flexível os dados do utilizador, dados de músicas e endereços de contratos. Para o armazenamento de ficheiros de grandes dimensões (como faixas de áudio e imagens de capa), é utilizada a extensão GridFS, que divide os ficheiros em blocos (*chunks*) e os gere de forma eficiente. A ligação entre o servidor e a base de dados é feita através da biblioteca `Mongoose`, garantindo uma interação segura e estruturada.

Blockchain – A componente descentralizada da solução baseia-se na rede Ethereum. Aqui estão criados os contratos inteligentes escritos em Solidity, responsáveis por formalizar e automatizar as operações de registo, reutilização e distribuição de *royalties*. A biblioteca `ethers.js` assegura a comunicação entre o *backend* e os contratos, permitindo o envio e leitura de transações na *blockchain*. Durante a fase de desenvolvimento, o ambiente Hardhat foi utilizado para compilar e testar localmente os contratos antes da sua implantação em redes públicas de teste.

A Figura 3.1 representa a arquitetura da plataforma SoundSlice.

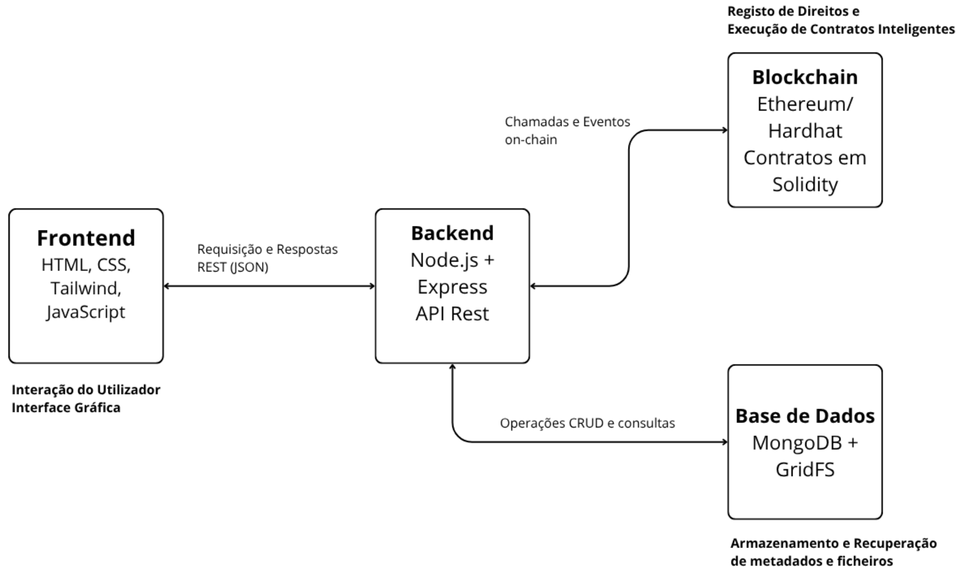


Figura 3.1: Arquitetura da plataforma.

A Figura 3.2 apresenta de forma detalhada o funcionamento interno da plataforma SoundSlice, evidenciando a integração entre as camadas centralizadas (base de dados) e as camadas descentralizadas (*blockchain*). O diagrama mostra o percurso completo das músicas desde o seu *upload* inicial até à criação dos contratos inteligentes `MusicRights`, `MusicReuse` e `MusicMix`, responsáveis por formalizar e automatizar a gestão dos direitos de autor e respetiva remuneração.

O diagrama evidencia como o processo de registo, reutilização e criação de *mixes* é totalmente automatizado. Cada música carregada pelo utilizador gera o contrato `MusicRights` que formaliza a autoria e o preço base do criador. Quando ocorre uma reutilização parcial, depois do pagamento da respetiva remuneração, é criado o contrato `MusicReuse`, associado ao excerto selecionado e ao seu valor tendo em conta a percentagem utilizada. Por sua vez, a combinação de múltiplas faixas gera o contrato `MusicMix`, responsável pela agregação de múltiplos participantes e faixas e pela distribuição de *royalties* entre todos os criadores originais.

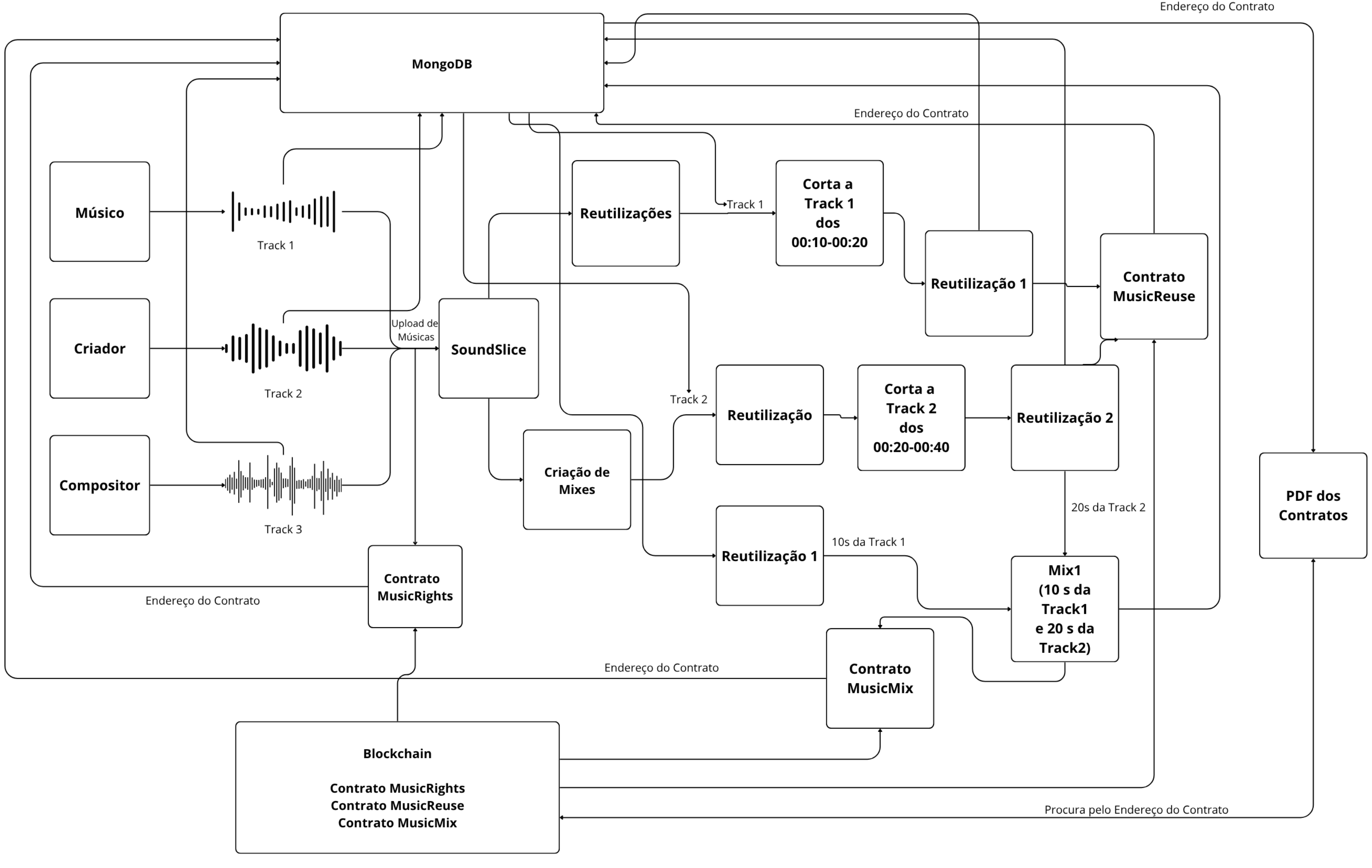


Figura 3.2: Fluxo de funcionamento da plataforma.

Todas as informações são agregadas na base de dados MongoDB e, de forma imutável, na *blockchain*. O sistema gera um comprovativo digital em PDF com os endereços de contrato e detalhes das operações.

3.4 *Blockchain* e Contratos Inteligentes

A tecnologia *blockchain* constitui a base de funcionamento dos contratos inteligentes, garantindo transparência, segurança e descentralização na execução das transações. Esta secção descreve os principais componentes utilizados no desenvolvimento da solução proposta, nomeadamente a rede Ethereum, a linguagem Solidity e o ambiente de desenvolvimento Hardhat.

3.4.1 Ethereum

A plataforma Ethereum foi escolhida por ser atualmente a rede mais madura e amplamente utilizada para o desenvolvimento de contratos inteligentes e aplicações descentralizadas. A Ethereum *Virtual Machine* (EVM) garante que o código dos contratos é executado de forma imutável e auditável, permitindo criar regras de distribuição de *royalties* que permanecem inalteradas após o registo [28].

Uma das desvantagens da utilização direta da *mainnet* do Ethereum são os custos associados às transações (*gas fees*) e o risco de comprometer fundos. Desta forma, existem redes de teste ou *testnet*, que são utilizadas para contornar este problema, replicando uma rede real mas com moedas sem valor económico. Assim, foi possível implementar uma destas redes e testar os contratos digitais de uma forma muito semelhante às redes reais, garantindo uma validação funcional sem custo financeiro. Atualmente, entre várias *testnet*, a Sepolia e a Holešky assumem maior relevância, substituindo redes mais antigas como Goerli ou Ropsten, entretanto descontinuadas. Além destas redes públicas, o projeto recorreu também à rede local providenciada pelo Hardhat, que permite simular transações com *Ether* (ETH) fictício, tornando o processo de desenvolvimento mais ágil e seguro [29].

No âmbito deste projeto, o recurso a uma *testnet* foi essencial para validar a criação e a execução dos contratos inteligentes que suportam a plataforma SoundSlice. Esta escolha permitiu simular todo o ciclo de vida da gestão de direitos, desde o registo inicial de uma música até à sua reutilização e respetiva compensação automática dos titulares, sem incorrer em custos reais. Assim, a Ethereum e as suas redes de teste constituiram a base tecnológica que garante a fiabilidade e a escalabilidade da solução desenvolvida.

3.4.2 Solidity

A linguagem de programação adotada para o desenvolvimento dos contratos inteligentes foi a Solidity. Criada em 2014 por Gavin Wood, cofundador da Ethereum, a Solidity é atualmente a linguagem mais utilizada na implementação de contratos inteligentes na Ethereum *Virtual Machine* [30]. A sua sintaxe foi inspirada em linguagens como JavaScript, Python e C++, tornando-a bastante acessível para os programadores familiarizados com estas linguagens [31]. As principais características desta linguagem são:

- Suporte a tipos de dados complexos.
- Possibilidade de criar estruturas de controlo e funções modulares.
- Existência de mecanismos de herança e reutilização de código.

Estas funcionalidades permitem estruturar contratos de forma clara e eficiente, promovendo a legibilidade do código. Além disso, a linguagem inclui mecanismos de segurança, como o controlo explícito do acesso a funções e a gestão de exceções, fundamentais para prevenir vulnerabilidades exploráveis em ambientes descentralizados [32].

No entanto, a Solidity também tem desvantagens. A curva de aprendizagem ser acentuada para programadores sem experiência em programação orientada a contratos inteligentes, podendo os erros causar danos irreversíveis na *mainnet* dada a imutabilidade do código [33]. Por esta razão, a comunidade Ethereum disponibiliza ferramentas de apoio ao desenvolvimento, como compiladores (`solc`), analisadores estáticos e ambientes de teste, que auxiliam na deteção precoce de erros e no reforço da segurança dos contratos [31].

No contexto da plataforma SoundSlice, a utilização da Solidity revelou-se essencial para modelar a lógica de negócio associada à gestão de direitos musicais. Foram definidas estruturas para representar informações como o título, o autor e a duração das obras musicais, bem como funções para registar novas criações, contabilizar reutilizações e distribuir automaticamente os valores devidos aos titulares dos direitos. Deste modo, a linguagem desempenhou um papel central na construção da camada de automatização e da confiança que sustenta a solução proposta.

3.4.3 Hardhat

O desenvolvimento e a validação de contratos inteligentes exigem um ambiente de testes que permita simular de forma segura as condições de execução numa *blockchain.* Para este efeito, foi utilizado o *framework* Hardhat, uma das ferramentas do ecossistema Ethereum [34].

O Hardhat disponibiliza uma *blockchain* local que funciona como rede de desenvolvimento, permitindo executar transações com ETH fictício, depurar contratos e medir custos de *gas* sem qualquer risco financeiro.

Esta tecnologia tem várias aplicações, entre elas, a compilação de contratos escritos em Solidity, a gestão de tarefas automatizadas através de *scripts* em JavaScript ou TypeScript, a possibilidade de integrar bibliotecas de teste como Mocha e Chai [35], e a geração de relatórios detalhados sobre o consumo de unidades de *gas* [36].

Estas características tornam a ferramenta especialmente adequada para ciclos de desenvolvimento iterativos, nos quais é necessário testar repetidamente a lógica de negócio antes de proceder ao *deployment* numa rede pública [34]. Comparando com outras ferramentas, o Hardhat distingue-se pela flexibilidade, pela integração com a biblioteca `Ethers.js` e pela existência de um console interativo que simplifica a execução de chamadas a contratos durante a fase de depuração [37].

No contexto da plataforma SoundSlice, o Hardhat foi essencial para a criação de um ambiente de desenvolvimento controlado. Com recurso à rede local disponibilizada pela ferramenta, foi possível compilar e testar contratos inteligentes, simular operações de registo e reutilização de músicas e avaliar a distribuição de *royalties* sem custos associados. Esta abordagem garantiu não apenas a fiabilidade dos contratos, mas também a eficiência do processo de desenvolvimento, reduzindo significativamente o tempo necessário para identificar e corrigir erros antes da implementação em redes de teste públicas.

Importa referir que, apesar de o Hardhat reproduzir com grande fidelidade o comportamento de uma rede Ethereum, as latências obtidas neste ambiente local correspondem a um cenário idealizado. Em redes reais (*realnets*), como a Sepolia ou a *mainnet*, o tempo de propagação e confirmação das transações é naturalmente superior devido à comunicação entre múltiplos nós e à variabilidade de carga.

3.5 Gestão de Direitos Musicais

A gestão de direitos musicais exige soluções interoperáveis, seguras e automatizadas. Os padrões *Smart Contracts for Media* e *Interactive Music Application Format*, desenvolvidas pelo MPEG, oferecem modelos padronizados para a execução de contratos inteligentes e a representação de metadados de licenciamento, promovendo eficiência e rastreabilidade no ecossistema digital.

3.5.1 *Smart Contracts for Media*

No âmbito da plataforma SoundSlice, o pacote SC4M[1], que implementa o correspondente padrão apresentado no Capítulo 2, fornece uma base de contratos inteligentes

[1] https://www.iso.org/standard/82527.html

especializados na gestão de conteúdos multimédia. A utilização deste pacote permitiu acelerar a definição da lógica contratual necessária para o registo e reutilização de músicas, evitando a criação de contratos totalmente novos e beneficiando de um modelo já alinhado com práticas normalizadas de gestão de direitos. Assim, o SC4M contribuiu para garantir interoperabilidade e consistência no tratamento de metadados e na associação entre obras musicais e titulares de direitos.

3.5.2 *Interactive Music Application Format*

O pacote IMAF[2]que implementa o correspondente padrão apresentado no Capítulo 2, tem impacto direto na representação da informação musical e nos mecanismos de interação com os conteúdos. No contexto do SoundSlice, a sua principal contribuição está na normalização da descrição de trechos musicais e na ligação entre estes e os contratos inteligentes. Este formato possibilita a identificação mais clara de reutilizações parciais de uma obra, facilitando o cálculo proporcional de compensações e reforçando a transparência do processo. A integração do IMAF, ainda que parcial, assegura que a plataforma se encontra preparada para adotar padrões reconhecidos na indústria e compatíveis com soluções futuras.

Apesar do IMAF não realizar a deteção automática de músicas, o seu papel é essencial na normalização da representação de conteúdos e trechos musicais, permitindo que a informação sobre reutilizações seja descrita de forma estruturada e facilmente associada a contratos inteligentes.

3.6 *Backend*

O *backend* da aplicação é composto pelo Node.js, Express e MongoDB, formando uma *stack* leve e escalável para o desenvolvimento de APIs REST com persistência de dados orientada a documentos.

3.6.1 Node.js e Express

O Node.js é um *framework* de código aberto, multiplataforma, baseado no interpretador V8 do Google e que permite a execução de códigos JavaScript fora de um navegador *web*. A principal característica do Node.js é a sua arquitetura assíncrona e orientada por eventos. O *runtime* do Node.js, que é composto por uma única *thread* designada *Event Loop*, é responsável por executar o código JavaScript. Não tendo necessidade de criar *threads* adicionais, o código torna-se mais simples de manter [38]. O Express.js, ou apenas Express, é um *framework* para Node.js que fornece recursos mínimos para construção de servidores *web* e facilita a criação de APIs do

[2]https://www.iso.org/standard/53644.html

tipo REST. O Express oferece um conjunto robusto de recursos para o desenvolvimento de servidores *web*, incluindo roteamento e manipulação de pedidos HTTP. O Node Pack Manager (NPM) é um poderoso gestor de pacotes que faz parte do Node.js, utilizado para instalar bibliotecas e dependências do projeto. O Node Pack Manager facilitou a gestão de pacotes, como Express, tornando a configuração do ambiente de desenvolvimento mais eficiente [39].

3.6.2 MongoDB e GridFS

O MongoDB é uma base de dados não relacional — *Not only* SQL (NoSQL) — orientada a documentos, que armazena informação em formato *Binary* JSON (BSON), uma extensão de JavaScript *Object Notation* (JSON). Ao contrário das bases de dados relacionais tradicionais, que usam tabelas rígidas com esquemas estruturados, o MongoDB oferece maior flexibilidade na forma como os dados são definidos e armazenados [40]. Esta característica é particularmente útil em projetos que envolvem diferentes tipos de informação, como metadados musicais, utilizadores e contratos inteligentes, que podem variar em formato e complexidade.

Uma das vantagens do MongoDB é a sua escalabilidade horizontal, que permite distribuir dados por diferentes servidores, assegurando desempenho mesmo em cenários com grande volume de informação. No entanto, conteúdos multimédia como ficheiros de áudio ou imagens podem ultrapassar o limite de tamanho de um único documento no MongoDB. Para resolver esse problema, o sistema disponibiliza o GridFS, um mecanismo de armazenamento de ficheiros que divide os dados em blocos (*chunks*) e os distribui pela base de dados [41]. Desta forma, é possível armazenar e recuperar ficheiros de grandes dimensões de forma eficiente, sem comprometer a estrutura da base de dados.

No contexto da plataforma SoundSlice, o MongoDB foi utilizado para armazenar a informação relativa aos utilizadores e às músicas registadas, enquanto o GridFS serviu para gerir os ficheiros de áudio e imagens de forma escalável. Com esta abordagem, tornou-se possível manter os metadados e os conteúdos multimédia integrados no mesmo sistema, simplificando a gestão e garantindo a consistência entre a informação textual e os ficheiros associados.

3.7 Processamento e Manipulação de Áudio

O FFmpeg é um conjunto de bibliotecas e aplicações de processamento de áudio e vídeo, amplamente utilizada em aplicações profissionais e de código aberto [42]. Na plataforma SoundSlice, o FFmpeg atua como o motor responsável pela manipulação dos ficheiros musicais, permitindo:

- extrair trechos específicos de uma música original, com base nos tempos definidos pelo utilizador;
- combinar múltiplas faixas de áudio para gerar um novo ficheiro final;
- calcular a duração exata de cada trecho e converter formatos, assegurando compatibilidade;
- gerar o *hash* e os metadados técnicos do ficheiro resultante.

A integração do FFmpeg no *backend* foi realizada através da biblioteca `fluent-ffmpeg`, que permite invocar comandos diretamente a partir do servidor Node.js. Por exemplo, ao selecionar um excerto musical na interface de reutilização, o sistema recebe os instantes de início e fim (`start`, `end`) e invoca o FFmpeg para criar um novo ficheiro temporário contendo apenas o trecho pretendido. Com esta abordagem, cada reutilização ou composição é gerada a partir de ficheiros processados localmente, assegurando uma correspondência direta entre o conteúdo manipulado e o registo armazenado na base de dados.

3.8 *Frontend*

O *frontend* da aplicação utiliza HTML5, CSS3 (Tailwind CSS) e JavaScript, permitindo criar uma interface responsiva, moderna e eficiente. Esta camada comunica directamente com o *backend*, garantindo a integração funcional entre a apresentação e a lógica da aplicação.

3.8.1 HTML5, CSS3 e JavaScript

A linguagem de anotação de hipertexto – *HyperText Markup Language* (HTML) – é uma linguagem que compõe a maior parte das páginas *web online* [43]. O HTML é utilizado para estruturar e formatar o conteúdo de páginas *web* através de tags, que são interpretadas pelo navegador. O HTML5 é a versão mais recente da linguagem de marcação responsável pela estruturação da maioria das páginas da *web* [44]. Introduziu novos elementos semânticos, como `<section>`, `<nav>` e `<article>`, passando a oferecer suporte nativo para conteúdos multimédia através das *tags* `<audio>` e `<video>`. Esta versão foi determinante para o desenvolvimento da plataforma SoundSlice, permitindo integrar de forma simples elementos como leitores de áudio diretamente no navegador, sem necessidade de *plugins* externos. A representação estruturada do conteúdo HTML de uma página *web* é feita através do *Document Object Model* (DOM), que é uma interface de programação que permite o acesso e manipulação dos elementos. O DOM define a estrutura em forma de árvore dos elementos, sendo cada um representado por um nó na estrutura do documento.

Principais elementos de uma página HTML:

- `<html>` : É o elemento raiz da página, de maneira hierárquica, está acima de todos.
- `<head>` : Refere-se à cabeça da página, onde é possível inserir configurações que auxiliam na renderização, assim como o título da página. Ele está dentro do elemento *<html>*, portanto ele é um elemento filho.
- `<title>` : Define o título da página, aquele que ficará visível na aba do navegador. É uma ramificação do `<head>`, sendo um elemento filho.
- `<body>` : É o corpo da página, onde serão inseridos os elementos visuais que deverão aparecer, como os textos. Ele também está hierarquicamente inserido dentro do `<html>`, portanto, é filho dele.
- `<h1>` : Refere-se ao título principal que ficará visível no corpo da página. Como está dentro do `<body>`, consequentemente pode ser considerado como filho.
- `<p>` : Constrói um parágrafo de texto de acordo com o conteúdo inserido dentro dele. Também está dentro do `body`, portanto é elemento filho.

Assim, o DOM é uma representação em forma de árvore do conteúdo HTML que permite que todos os elementos sejam acessados através do JavaScript [45].

A linguagem CSS é usada para definir o estilo de elementos escritos numa linguagem de anotação como HTML, separando o conteúdo da representação visual [46]. Permite a definição dos diferentes estilos – alinhamento, fontes (tipo, cor e dimensão), fundo, etc. – num ficheiro CSS, que se declara no cabeçalho das páginas HTML da aplicação *web*. Desta forma, os estilos definidos ficam disponíveis para aplicar aos elementos HTML das páginas. A versão mais recente, conhecida como CSS3, introduziu uma série de melhorias significativas que permitem criar interfaces mais modernas, dinâmicas e responsivas [47]. Ao separar a camada de estilo da estrutura do conteúdo, o CSS3 promove uma maior legibilidade do código, facilita a manutenção e possibilita a reutilização de estilos em diferentes páginas.

O JavaScript é uma linguagem de programação interpretada, estruturada e de alto nível, amplamente utilizada no desenvolvimento de aplicações para a *web* [48]. Inicialmente criada em 1995 pela Netscape como uma linguagem de *scripting* para navegadores, rapidamente evoluiu para uma das tecnologias centrais da *World Wide Web*, em conjunto com o HTML e o CSS [49]. Atualmente, o JavaScript encontra-se padronizado pela especificação ECMAScript, mantida pela European Computer Manufacturers Association (ECMA) International, que define novas funcionalidades e garante a interoperabilidade entre diferentes navegadores [50]. Entre as principais características do JavaScript destacam-se a sua natureza dinâmica, o modelo de orientação a objetos baseada em protótipos, e a possibilidade de manipular o DOM em tempo real, permitindo criar interfaces interativas e responsivas. Além

disso, a linguagem suporta programação assíncrona através de *callbacks*, *promises* e *async/await*, funcionalidades essenciais para lidar com operações dependentes de rede, como pedidos a uma API. No contexto da plataforma SoundSlice, o JavaScript foi fundamental para implementar a lógica do *frontend* e gerir as interações do utilizador com a interface. Através desta linguagem foi possível, por exemplo, validar formulários de registo e autenticação, manipular dinamicamente elementos de página, e comunicar com o *backend* através de pedidos HTTP em formato JSON. Deste modo, o JavaScript assegurou a ligação entre a camada visual e a lógica da aplicação, permitindo uma experiência de utilização mais fluida e intuitiva.

3.8.2 Tailwind CSS

O Tailwind CSS é um *framework* de estilização baseado em classes utilitárias que segue a filosofia *utility-first* [51]. Ao contrário de outros *frameworks*, nomeadamente o Bootstrap, que disponibilizam componentes prontos a usar, o Tailwind oferece um conjunto extenso de classes simples que podem ser combinadas para construir interfaces totalmente personalizadas [52].

Uma das principais vantagens desta abordagem é a rapidez na criação de protótipos e a facilidade em manter uma identidade visual consistente em todo o projeto. Como as regras de estilo são aplicadas diretamente nas classes HTML, reduz-se a necessidade de escrever folhas de estilo complexas, tornando o desenvolvimento mais ágil e a manutenção mais simples. Além disso, o Tailwind já integra suporte para conceitos modernos como design responsivo, *dark mode* e personalização de temas através de ficheiros de configuração.

No caso da plataforma SoundSlice, a utilização do Tailwind CSS foi fundamental para acelerar a construção das páginas da interface. O *framework* permitiu criar rapidamente *layouts* modernos e adaptáveis a diferentes tamanhos de ecrã, garantindo um aspeto profissional sem exigir grande esforço adicional em CSS manual. Deste modo, o Tailwind CSS não só contribuiu para a eficiência no desenvolvimento, como também ajudou a proporcionar uma experiência de utilização mais coerente e agradável.

3.8.3 Integração com o *Backend*

A camada de *backend* da plataforma SoundSlice foi desenvolvida em Node.js com o *framework* Express, responsável por gerir todas as comunicações entre o *frontend*, a base de dados e a *Blockchain*. Esta camada atua como intermediária entre o navegador e os restantes serviços, recebendo pedidos HTTP no formato JSON e respondendo com dados estruturados, garantindo uma separação clara entre a interface do utilizador e a lógica de negócio. O *backend* é composto por uma API do tipo REST que centraliza todas as operações de criação, consulta e atualização de dados,

nomeadamente o registo de novas músicas, a gestão de reutilizações e a geração de contratos derivados (*mixes*). Cada uma destas operações corresponde a um conjunto de controladores e rotas como:

Principais rotas da API REST

- `POST /api/tracks/upload` — responsável por receber o ficheiro enviado no *frontend*, processar os metadados e armazená-los no MongoDB;
- `POST /api/tracks/:id/reuse/select` — calcula a percentagem de reutilização e o valor a pagar com base na duração do trecho;
- `POST /api/tracks/:id/reuse/confirm` — confirma a transação e aciona o *smart contract* correspondente na *blockchain*.
- `POST /api/tracks/export` — cria uma nova *mix*, armazenando o ficheiro no GridFS e gerando automaticamente o contrato `MusicMix`.

O *backend* também integra funções específicas de comunicação com a *blockchain* através da biblioteca `ethers.js`. Estas funções são encapsuladas em módulos utilitários, como `blockchainReuse.js` e `blockchainMix.js`, permitindo realizar o *deployment* de contratos inteligentes diretamente a partir do servidor, de forma segura e automática. Durante este processo, o servidor utiliza a variável de ambiente `SERVER_PRIVATE_KEY` para assinar transações de forma autenticada, garantindo que apenas operações válidas e autorizadas são executadas na rede *blockchain*.

Para além da gestão de contratos e metadados, o *backend* é também responsável por controlar o fluxo entre o utilizador e a base de dados. Cada ação no *frontend*, nomeadamente envio de uma música, o pagamento de uma reutilização ou a criação de uma *mix*, gera um pedido à API REST, que é validado, processado e armazenado. O resultado é então devolvido em formato JSON, permitindo que a interface atualize dinamicamente as informações apresentadas ao utilizador, como o estado da transação, o endereço do contrato criado e o valor pago.

3.8.4 Integração com a *Blockchain*

A integração da plataforma SoundSlice com a *blockchain* Ethereum ocorre em três momentos distintos, que correspondem às principais etapas do ciclo de vida de uma obra musical: (*i*) o registo inicial da música, que cria o contrato `MusicRights`; (*ii*) a reutilização de excertos, associada ao contrato `MusicReuse`; e (*iii*) a criação de composições derivadas, tratada através do contrato `MusicMix`. Em todas as etapas, a interação é realizada através da biblioteca `ethers.js`, garantindo uma comunicação segura e eficiente entre o *frontend*, o *backend* e a rede *blockchain*. A Figura 3.3

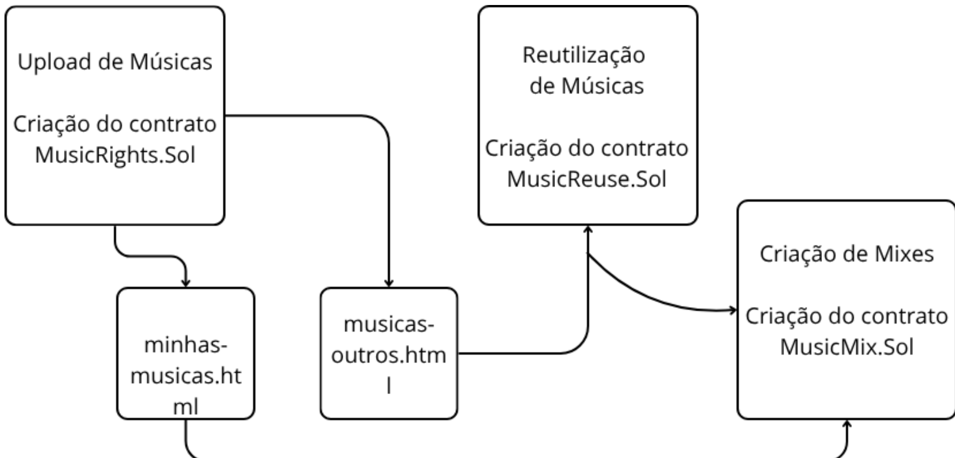


Figura 3.3: Fluxo de integração da plataforma SoundSlice com a *blockchain* Ethereum.

resume os três tipos de contratos e respetivos momentos de criação na plataforma SoundSlice.

Esta arquitetura modular garante que todas as ações relacionadas com a gestão de direitos — desde o registo da obra até à sua reutilização e combinação — são documentadas de forma imutável na *blockchain*. Assim, o sistema assegura não apenas a transparência das transações, mas também a confiança e a auditabilidade de todas as relações contratuais estabelecidas no mundo musical.

Registo da Música Original — Contrato `MusicRights`

Quando um criador faz o *upload* de uma música pela primeira vez, é gerado automaticamente um contrato inteligente `MusicRights`, que formaliza a propriedade e os direitos associados à obra. Este contrato contém informações como o título, o autor, o *hash* do ficheiro, o preço base, o género, o formato e a duração. Além disso, é possível definir vários titulares e respetivas percentagens de participação, o que permite representar obras de coautoria diretamente na *blockchain*. O *deployment* deste contrato é feito através de um *script* Hardhat (`deploy.cjs`), mostrado de forma simplificada na Listagem 3.1.

```
1 const factory = new ethers.ContractFactory(CONTRACT_ABI,
    CONTRACT_BYTECODE, signer);
2 const contract = await factory.deploy(
3   title, artist, fileHash, priceWei, format, genre, duration,
      extra, holders, shares
4 );
5 await contract.waitForDeployment();
6 const contractAddress = await contract.getAddress();
7 console.log("Contrato criado:", contractAddress);
```

Listagem 3.1: *Deployment* do contrato inteligente `MusicRights` durante o registo da música

Este processo cria um contrato único e imutável para cada música registada, servindo como certificado digital de autoria e garantindo que os direitos e percentagens dos titulares são públicos e verificáveis.

Reutilização de Obras — Contrato `MusicReuse`

O segundo nível de integração ocorre quando um utilizador reutiliza parcialmente uma música existente. Após selecionar o excerto e efetuar o pagamento via carteira digital `MetaMask`, o *backend* aciona automaticamente a função `deployMusicReuseContract()`, que implanta um novo contrato `MusicReuse` na rede Ethereum. Este contrato associa o trecho reutilizado à obra original e regista dados como o autor original, o reutilizador, a percentagem de reutilização e o valor pago – ver a Listagem 3.2.

```
const contract = await factory.deploy({
  reuseId: params.reuseId,
  originalId: params.originalId,
  creatorWallet: params.creatorWallet,
  reuserWallet: params.reuserWallet,
  reusePercent: BigInt(params.reusePercent),
  valuePaid: ethers.parseEther(params.valueEth.toString()),
  format: params.format,
  genre: params.genre,
  snippetDuration: BigInt(params.snippetDuration)
});
await contract.waitForDeployment();
```

Listagem 3.2: Criação automática do contrato `MusicReuse` no *backend*

Este contrato é armazenado na rede de teste local do Hardhat durante a fase de desenvolvimento, garantindo a transparência e rastreabilidade de cada reutilização musical.

Criação de Composições Derivadas — Contrato `MusicMix`

Por fim, o contrato `MusicMix` representa o caso em que o utilizador cria uma nova composição a partir de várias músicas previamente registadas. A função `deployMusicMixContract()` é responsável por criar o contrato correspondente, recebendo a

lista de músicas originais, os seus autores, percentagens de reutilização e partilhas de valor – consultar a Listagem 3.3.

```
const sources = (params.sources || []).map((s) => ({
  originalId: s.originalId,
  title: s.title,
  creatorName: s.creatorName,
  creatorWallet: s.creatorWallet,
  reusePercent: BigInt(s.reusePercent),
  valueShare: ethers.parseEther(String(s.valueShare))
}));

const contract = await factory.deploy(
  params.mixId, params.mixTitle,
  params.mixCreator, params.mixWallet,
  params.format, params.genre, sources
);
await contract.waitForDeployment();
```

Listagem 3.3: Deploy do contrato `MusicMix` para composições colaborativas

O contrato `MusicMix` distribui automaticamente as remunerações entre todos os criadores originais incluídos na *mix*, assegurando que cada titular recebe a sua quota proporcional de forma descentralizada.

3.8.5 Integração com o MongoDB

A plataforma SoundSlice utiliza o MongoDB como sistema principal de armazenamento de dados, tanto para os metadados musicais como para os registos dos contratos inteligentes e dos utilizadores. Esta escolha deve-se à flexibilidade da sua estrutura orientada a documentos, que permite representar de forma eficiente diferentes tipos de conteúdos musicais e as suas relações de reutilização.

A ligação à base de dados é gerida através da biblioteca `Mongoose`, que facilita a definição de esquemas e modelos com tipagem explícita. O modelo `Track` é utilizado para representar qualquer tipo de obra registada na plataforma — músicas originais, reutilizações e *mixes* — distinguindo-se através do campo `type`, que pode assumir os valores:

- `"original"` — quando a faixa é carregada pelo autor pela primeira vez;
- `"reuse"` — quando corresponde a uma reutilização parcial de uma obra existente;
- `"mix"` — quando resulta da combinação de várias faixas ou trechos reutilizados.

O campo `user` estabelece a relação direta entre cada música e o utilizador que a criou, permitindo identificar rapidamente a autoria e aplicar regras de remuneração baseadas nos contratos inteligentes associados.

Os ficheiros de áudio e as imagens de capa são armazenados no GridFS, o sistema interno do MongoDB para ficheiros binários de grande dimensão. O GridFS divide os ficheiros em blocos de 255 kB e armazena-os em coleções específicas — `uploads.files` e `uploads.chunks`. Cada documento da coleção `tracks` guarda o identificador (`fileId`) do ficheiro correspondente, mantendo uma ligação direta entre os metadados e o conteúdo real. O MongoDB é também responsável por gerir as relações entre obras originais e reutilizadas. Cada reutilização (`reuse`) mantém uma referência ao identificador da música original, o que permite rastrear a sua origem e calcular automaticamente a compensação correspondente.

Esta abordagem simplifica a gestão de diferentes tipos de conteúdos dentro de uma única coleção, facilitando consultas agregadas e mantendo a coerência entre os dados armazenados localmente e os contratos inteligentes registados na *blockchain*. Deste modo, a plataforma SoundSlice garante a rastreabilidade completa de todas as obras, desde o *upload* inicial até às reutilizações e combinações subsequentes, assegurando a transparência e a correta atribuição de direitos aos seus autores.

3.9 Sumário

Este capítulo apresentou a proposta técnica da plataforma SoundSlice, desde a arquitetura geral até à implementação das suas principais camadas — *frontend*, *backend*, base de dados e *blockchain*. Foram descritas as tecnologias selecionadas, as suas funções e o modo como interagem de forma integrada para garantir a rastreabilidade e a automatização dos direitos musicais.

O desenvolvimento e a integração dos componentes da plataforma são detalhados no capítulo seguinte.

Capítulo 4

Implementação e Integração

Este capítulo descreve a implementação prática da plataforma SoundSlice, apresentando as componentes desenvolvidas no *frontend*, *backend* e na camada de contratos inteligentes. O objetivo é demonstrar como as especificações definidas na proposta de solução foram traduzidas em artefactos funcionais, bem como evidenciar a integração entre as diferentes camadas da arquitetura — desde a interface web até à execução de transações na *blockchain*.

4.1 Implementação do *Frontend*

Esta secção descreve as diferentes páginas constituintes da interface, detalhando as funcionalidades oferecidas.

4.1.1 Página Principal

A página `index.html` constitui o ponto de entrada da aplicação SoundSlice. O seu principal objetivo é apresentar a interface inicial ao utilizador, permitindo navegar para as restantes secções da plataforma, nomeadamente o registo, o *login* e a página `musicas-outros.html` que mostra todas as músicas públicas que foram inseridas.

O corpo do ficheiro é composto por três blocos principais: o cabeçalho (*header*), o conteúdo principal (*main*) e o rodapé (*footer*). A barra de navegação adapta-se consoante o estado de autenticação do utilizador. Quando este se encontra autenticado, o *link* de *login* é substituído por um menu com o nome e a imagem de perfil,

acompanhado de uma lista suspensa com as opções "Perfil", que redireciona para a página `profile.html` e "Logout".

A autenticação é gerida no lado do cliente através do armazenamento do *token* de sessão (`authToken`) no `localStorage`. Sempre que a página é carregada, é invocada a função `loadUser()`, responsável por validar o *token* e obter as informações do utilizador autenticado através do *endpoint* `/api/users/me` do *backend*. A Listagem 4.1 ilustra esta funcionalidade.

```
async function loadUser() {
  const token = localStorage.getItem("authToken");
  if (!token) {
    document.getElementById("auth-link").style.display = "inline-
        block";
    document.getElementById("userMenuWrapper").classList.add("
        hidden");
    return;
  }

  const res = await fetch(`${API}/api/users/me`, {
    headers: { Authorization: "Bearer " + token }
  });

  if (res.ok) {
    const user = await res.json();
    document.getElementById("welcomeName").textContent =
      "Olá " + (user.firstname || "Utilizador");
    document.getElementById("profilePic").src =
      user.profilePic || "images/default-avatar.png";
    document.getElementById("auth-link").style.display = "none";
    document.getElementById("userMenuWrapper").classList.remove("
        hidden");
  }
}
```

Listagem 4.1: Verificação de autenticação do utilizador.

Quando o *token* é válido, a função altera dinamicamente a interface, exibindo o nome do utilizador e o seu avatar. Caso contrário, mantém o *link* de autenticação visível. O *backend*, por sua vez, valida o *token* recebido, consulta a base de dados MongoDB e devolve um objeto JSON com os dados do utilizador, incluindo o nome e o *Uniform Resource Locator* (URL) da imagem de perfil armazenada em GridFS. A função `logoutBtn` permite encerrar a sessão removendo o *token* local e redirecionando o utilizador para a página de *login*.

O corpo da página é composto por um *banner* com uma breve introdução que apresenta a missão da plataforma, outro *banner* que convida o utilizador a descobrir

as várias faixas e redireciona para a página `musicas-outros.html`. Uma das funcionalidades dinâmicas centrais desta página é a exibição das músicas mais recentes publicadas na plataforma. O carregamento é efetuado de forma assíncrona através da função `fetchLatestMusic()`, que envia um pedido ao *endpoint* `/api/tracks/-feed?sort=recent` e recebe uma lista de obras em formato JSON (ver Listagem 4.2).

```
1 async function fetchLatestMusic() {
2   const res = await fetch(`${API}/api/tracks/feed?sort=recent`);
3   if (!res.ok) throw new Error('Erro ao carregar músicas');
4   const musics = await res.json();
5   renderLatest(musics.slice(0, 4));
6 }
```

Listagem 4.2: Carregamento assíncrono das músicas mais recentes.

A função `renderLatest()`, apresentada na Listagem 4.3, processa o resultado recebido e gera dinamicamente os cartões de música que compõem a grelha de apresentação. Cada cartão contém a imagem de capa, o título, o autor e um *link* que direciona para a página `reuse.html`, onde o utilizador pode escutar e reutilizar a obra.

```
1 function renderLatest(musics) {
2   const grid = document.getElementById('latestGrid');
3   grid.innerHTML = '';
4   musics.forEach((m, i) => {
5     const card = document.createElement('a');
6     card.href = `reuse.html?id=${m._id}`;
7     card.innerHTML = `
8       <div class="h-40 w-full overflow-hidden">
9         <img src="${m.coverId ? `${API}/api/tracks/${m._id}/cover`
              : 'images/placeholder.png'}"
10             alt="Capa da musica ${m.title}" class="h-40 w-full
                  object-cover">
11      </div>
12      <div class="p-4">
13        <h4 class="font-semibold text-slate-900">${m.title}</h4>
14        <p class="text-sm text-slate-500">por ${m.artist}</p>
15      </div>`;
16    grid.appendChild(card);
17  });
18 }
```

Listagem 4.3: Renderização dinâmica das músicas recentes.

Este processo efetua a integração direta entre a camada de apresentação e o

servidor *Node.js*. O *backend* responde ao pedido consultando a base de dados MongoDB, ordenando as músicas mais recentes e devolvendo um conjunto limitado de resultados. As capas são obtidas através de pedidos adicionais ao *endpoint* `/api/tracks/{id}/cover`, que recupera o ficheiro armazenado no GridFS. A Figura 4.1 apresenta a interface da página `index.html`.

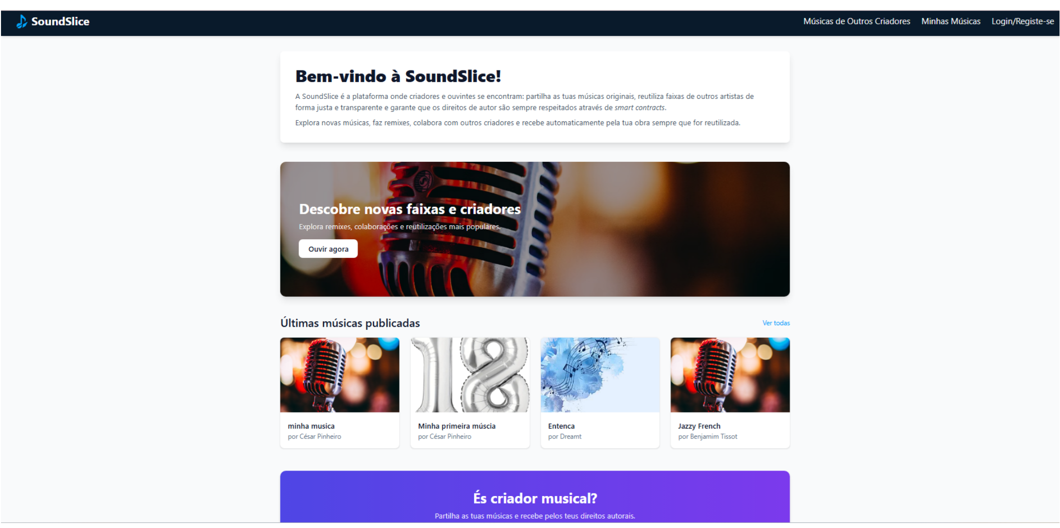


Figura 4.1: Interface da página principal da plataforma SoundSlice.

4.1.2 Registo e Autenticação de Utilizadores

A autenticação e o registo de utilizadores constituem o primeiro passo funcional da plataforma SoundSlice, assegurando que todas as interações, como o *upload*, reutilização e criação de músicas, são associadas de forma segura e inequívoca a um autor registado. O sistema implementa uma abordagem de autenticação híbrida: o *frontend* trata da recolha e validação dos dados, enquanto o *backend* é responsável pela persistência e verificação das credenciais, pela encriptação das palavras-passe e pela emissão de JSON *Web Tokens* (JWTs).

A secção divide-se em duas partes: (*i*) a página de registo (`register.html`), onde é criado o perfil de utilizador; e (*ii*) a página de autenticação (`login.html`), que permite o acesso às funcionalidades da plataforma.

Registo de Utilizadores

A página `register.html` foi concebida como um formulário de múltiplas etapas, permitindo recolher de forma estruturada as informações pessoais, os dados de pagamento e a confirmação de consentimentos legais. O formulário é dividido em três secções:

- **Dados Pessoais** — inclui o nome, apelido, nome artístico, data de nascimento, email, palavra-passe, telefone e foto de perfil;

- **Pagamentos** — recolhe o país, cidade, morada e endereço de carteira Ethereum do utilizador, com validação do formato `0x[a-fA-F0-9]{40}`;
- **Confirmação** — apresenta um resumo dos dados inseridos e solicita a aceitação dos termos de utilização e da política de privacidade.

A navegação entre as etapas é controlada pela função `showStep()`, que ativa a secção correspondente e atualiza o estado visual do menu de progresso, presente na Listagem 4.4.

```
1 const showStep = (index) => {
2   steps.forEach((step, i) => {
3     step.classList.toggle('active', i === index);
4     menuSteps[i].classList.toggle('active', i === index);
5   });
6   backBtn.style.display = index > 0 ? 'block' : 'none';
7   nextBtn.textContent = index === steps.length - 1 ? 'Submeter' :
        'Proximo';
8 };
```

Listagem 4.4: Gestão da navegação entre etapas do formulário.

Antes do envio dos dados, o *frontend* executa um conjunto de validações destinadas a garantir a integridade e coerência da informação:

- verificação da força da palavra-passe (mínimo de 8 caracteres, contendo maiúsculas, números e símbolos);
- correspondência entre a palavra-passe e a confirmação;
- cálculo da idade mínima (15 anos) a partir da data de nascimento;
- validação do formato do endereço Ethereum;
- obrigatoriedade de aceitação dos termos legais e políticas de privacidade.

Quando o utilizador conclui o formulário, é criado um objeto `FormData()` contendo todos os campos, incluindo ficheiros binários (como a foto de perfil). Estes dados são enviados para o *backend* através de um pedido `POST` ao *endpoint* `/api/users/register`, como é demonstrado na Listagem4.5.

```
1 const formData = new FormData();
2 formData.append("firstname", document.getElementById('firstname').
      value);
3 formData.append("lastname", document.getElementById('lastname').
      value);
4 formData.append("email", document.getElementById('email').value);
```

```
5 formData.append("password", document.getElementById('password').
    value);
6 formData.append("ethWallet", document.getElementById('ethWallet').
    value);
7 ...
8 const res = await fetch('http://localhost:3000/api/users/register'
    , {
9   method: 'POST',
10  body: formData
11 });
```

Listagem 4.5: Envio do formulário de registo para o *backend*.

No *backend*, o controlador `userController.js` recebe a requisição, valida os campos, encripta a palavra-passe com `bcrypt`, armazena os dados do utilizador na coleção `users` da base de dados MongoDB e grava a imagem de perfil no GridFS. Em caso de sucesso, devolve uma resposta JSON com o código `200` e a mensagem de confirmação. A Figura 4.2 apresenta a interface da página de registo.

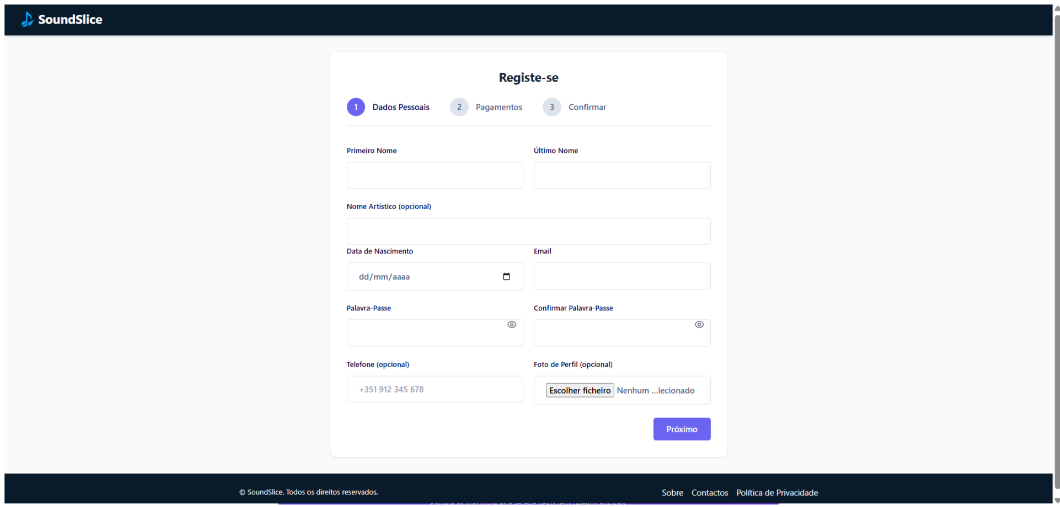


Figura 4.2: Interface da página de registo (`register.html`).

Autenticação de Utilizadores

A autenticação (*Login* e *Logout*) é realizada através da página `login.html`, que recolhe as credenciais de acesso e as envia para o *backend*. O processo é composto por três etapas principais:

- recolha do `email` e da `palavra-passe`;
- envio das credenciais via `POST` para o *endpoint* `/api/users/login`;
- receção de um *token* JWT que será armazenado localmente no `localStorage`.

O *backend* valida as credenciais e, em caso de sucesso, emite um *token* JWT assinado, contendo o identificador do utilizador. Esse *token* é utilizado nas páginas seguintes para autenticar pedidos via cabeçalho Authorization: Bearer <token>.

O *logout* é implementado de forma simples, eliminando o *token* armazenado localmente e redirecionando o utilizador para a página inicial ou de *login*. Este processo é invocado pelo botão "Logout" no menu do utilizador como está presente na Listagem 4.6.

```
1 document.getElementById("logoutBtn")?.addEventListener("click", ()
      => {
2   localStorage.removeItem("authToken");
3   window.location.href = "login.html";
4 });
```

Listagem 4.6: Encerramento de sessão do utilizador.

A implementação do sistema de autenticação do SoundSlice garante uma integração fluida entre a camada de interface e o servidor, combinando segurança e simplicidade de utilização. A Figura 4.3 apresenta a interface da página de *login*. O uso de *tokens* JWT permite que o estado de autenticação seja mantido de

Figura 4.3: Interface da página de autenticação (login.html).

forma descentralizada, sem necessidade de sessões no servidor, o que é particularmente vantajoso numa arquitetura baseada em microserviços e comunicação com contratos inteligentes. Além disso, a recolha do endereço Ethereum no momento do registo estabelece uma correspondência direta entre a identidade digital e a identidade *blockchain* do utilizador, um aspeto essencial para a remuneração automática e a validação de propriedade autoral nas fases seguintes da aplicação.

4.1.3 Perfil de Utilizador

A página `profile.html` permite ao utilizador aceder e gerir as suas informações pessoais, acompanhar o histórico de pagamentos e consultar as reutilizações associadas às suas obras musicais. Esta interface representa o núcleo da componente de gestão de identidade da plataforma SoundSlice, integrando dados provenientes da base de dados e da *blockchain* através de *endpoints* específicos do *backend*.

A página foi desenvolvida seguindo uma estrutura modular com três secções principais, Perfil, Pagamentos e Reutilizações. O acesso aos dados depende da autenticação prévia do utilizador, garantida por um *token* JWT armazenado no navegador. A troca entre secções é feita dinamicamente através da manipulação do DOM, sem recarregamento da página. Quando o utilizador clica numa opção do menu, a função seguinte oculta todas as secções e exibe apenas a selecionada, como se mostra na Listagem 4.7.

```
1 document.querySelectorAll("aside nav a").forEach(link => {
2   link.addEventListener("click", e => {
3     e.preventDefault();
4     document.querySelectorAll(".section").forEach(sec => sec.
          classList.add("hidden"));
5     document.getElementById(link.dataset.section).classList.remove
          ("hidden");
6   });
7 });
```

Listagem 4.7: Alternância entre secções do painel de perfil.

Secção "Meu Perfil"

A primeira secção permite ao utilizador visualizar e atualizar os seus dados pessoais, incluindo o nome, contacto, país, cidade, endereço Ethereum e fotografia de perfil. A função `loadProfile()` da Listagem 4.8 é executada assim que a página é carregada para solicitar os dados do utilizador autenticado ao *backend*, através do *endpoint* `/api/users/me`. Este pedido é autenticado com o *token* JWT obtido no *login* (ver Listagem 4.8).

```
1 async function loadProfile() {
2   const token = localStorage.getItem("authToken");
3   if (!token) {
4     window.location.href = "login.html";
5     return;
6   }
7
```

```
8     const res = await fetch("http://localhost:3000/api/users/me", {
9       headers: { Authorization: "Bearer " + token }
10    });
11
12    if (!res.ok) {
13      window.location.href = "login.html";
14      return;
15    }
16
17    const data = await res.json();
18    document.getElementById("firstname").value = data.firstname || "
        ";
19    document.getElementById("lastname").value = data.lastname || "";
20    document.getElementById("phone").value = data.phone || "";
21    document.getElementById("ethWallet").value = data.ethWallet || "
        ";
22    document.getElementById("country").value = data.country || "";
23    document.getElementById("city").value = data.city || "";
24    if (data.profilePic) {
25      document.getElementById("profilePreview").src = data.
          profilePic;
26    }
27  }
```

Listagem 4.8: Carregamento dos dados de perfil do utilizador.

O *backend* devolve um objeto JSON com as informações do utilizador, recolhidas a partir da coleção `users` da base de dados MongoDB. A imagem de perfil é servida através de um *stream* do GridFS.

A atualização dos dados é feita através de um pedido `PUT` para o *endpoint* `/api/users/update` com um `FormData()`. A Listagem 4.9 ilustra a submissão do formulário.

```
1   document.getElementById("profileForm").addEventListener("submit",
      async (e) => {
2     e.preventDefault();
3     const token = localStorage.getItem("authToken");
4     const formData = new FormData();
5     formData.append("firstname", document.getElementById("firstname"
        ).value);
6     formData.append("lastname", document.getElementById("lastname").
        value);
7     formData.append("phone", document.getElementById("phone").value)
        ;
8     formData.append("ethWallet", document.getElementById("ethWallet"
        ).value);
```

```
9       formData.append("country", document.getElementById("country").
            value);
10      formData.append("city", document.getElementById("city").value);
11
12      const fileInput = document.getElementById("profilePic");
13      if (fileInput.files.length > 0)
14        formData.append("profilePic", fileInput.files[0]);
15
16      const res = await fetch("http://localhost:3000/api/users/update"
            , {
17        method: "PUT",
18        headers: { Authorization: "Bearer " + token },
19        body: formData
20      });
21    });
```

Listagem 4.9: Atualização do perfil do utilizador.

No *backend*, o controlador `userController.js` recebe a requisição, atualiza os campos indicados na base de dados e substitui a imagem no GridFS, caso tenha sido enviada uma nova. O utilizador é notificado do sucesso da operação através de uma mensagem gerada pela função `notify()`. A Figura 4.4 apresenta a interface da página "Meu Perfil".

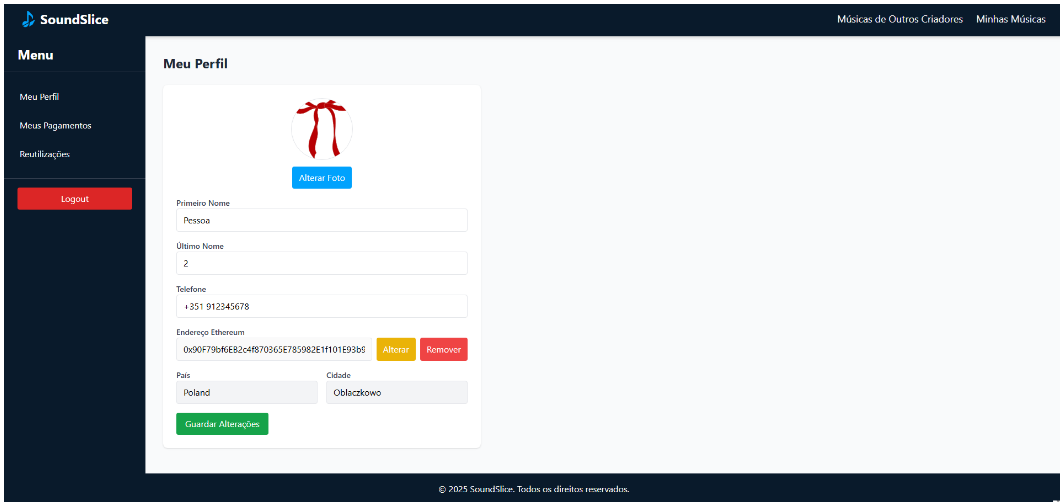


Figura 4.4: Interface da página do perfil (`profile.html`).

Secção "Meus Pagamentos"

A segunda secção apresenta o histórico de transações do utilizador, permitindo visualizar tanto os valores recebidos por reutilizações das suas músicas como os pagamentos efetuados por reutilizar faixas de outros autores. Os dados são obtidos através do *endpoint* `/api/users/payments`, que devolve uma lista de objetos JSON com os

campos `date`, `valueEth`, `direction`, `counterparty` e `trackTitle`. O resultado é formatado dinamicamente numa tabela, como se apresenta na Listagem 4.10.

```
async function loadPayments() {
  const token = localStorage.getItem("authToken");
  const res = await fetch("http://localhost:3000/api/users/
      payments", {
    headers: { Authorization: "Bearer " + token },
  });
  const payments = await res.json();

  const table = document.getElementById("paymentsTable");
  table.innerHTML = payments.map(p => `
    <tr class="border-b hover:bg-gray-50">
      <td class="p-2">${new Date(p.date).toLocaleDateString()}</td
          >
      <td class="p-2">${p.valueEth} ETH</td>
      <td class="p-2">${p.direction === "in" ? "Recebido de " : "
          Pago a "}${p.counterparty}</td>
      <td class="p-2">${p.trackTitle}</td>
    </tr>`).join("");
}
```

Listagem 4.10: Carregamento dos dados de pagamentos.

O *backend* processa estes dados a partir da coleção `transactions`, que é alimentada automaticamente pelos contratos inteligentes sempre que ocorre uma reutilização musical. Assim, esta secção constitui a principal ligação visível entre a camada *web* e a *blockchain*, refletindo as transferências realizadas na rede Ethereum (simulada via Hardhat). A Figura 4.5 apresenta a interface da secção "Meus Pagamentos".

Secção "Reutilizações"

A última secção apresenta dois conjuntos de informações:

- **As minhas músicas reutilizadas** — lista de faixas do utilizador que foram reutilizadas por outros criadores, com indicação do número total de reutilizações;
- **Músicas que reutilizei** — obras de terceiros que o utilizador integrou nas suas criações, acompanhadas da percentagem de uso.

Os dados são carregados a partir do *endpoint* `/api/users/reuses`, que devolve um objeto JSON com as listas `reusedByOthers` e `myReuses`. A renderização é feita através da função `loadReuses()` (ver Listagem 4.11).

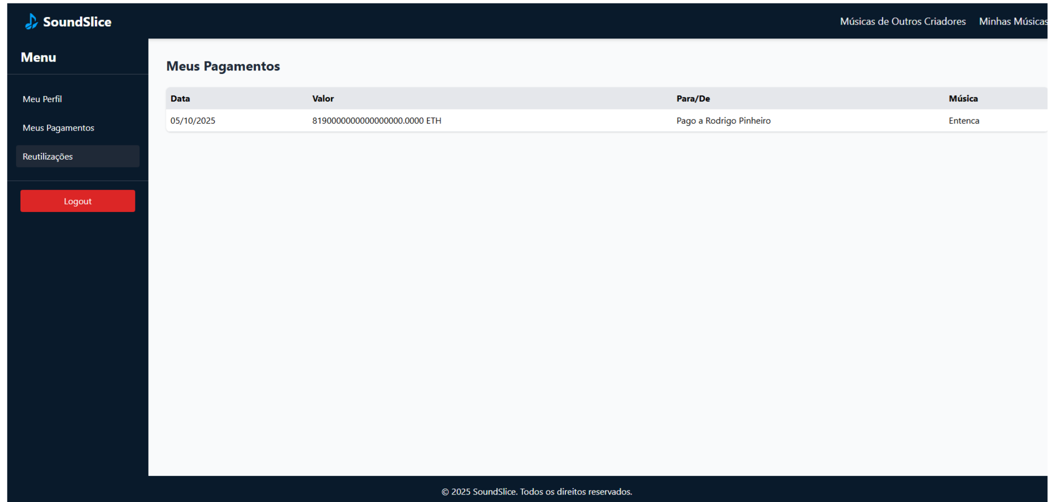


Figura 4.5: Interface da página do perfil – secção “Meus Pagamentos”.

```
1  async function loadReuses() {
2    const token = localStorage.getItem("authToken");
3    const res = await fetch("http://localhost:3000/api/users/reuses"
         , {
4      headers: { Authorization: "Bearer " + token },
5    });
6    const data = await res.json();
7
8    const myReuses = document.getElementById("myReuses");
9    const usedByMe = document.getElementById("usedByMe");
10
11   myReuses.innerHTML = data.reusedByOthers?.length
12    ? data.reusedByOthers.map(r => '<li>"${r.title}" - ${r.
         totalReuses} reutilizações</li>').join("")
13
14   usedByMe.innerHTML = data.myReuses?.length
15     ? data.myReuses.map(r => `<li>Reutilizaste "${r.title}" - ${r.
          percent}% usado</li>`).join("")
16     : `<li class="text-gray-400">Ainda nao reutilizaste nenhuma mú
          sica.</li>`;
17 }
```

Listagem 4.11: Carregamento das informações de reutilização.

A Figura 4.6 apresenta a interface da secção “Minhas Reutilizações”.

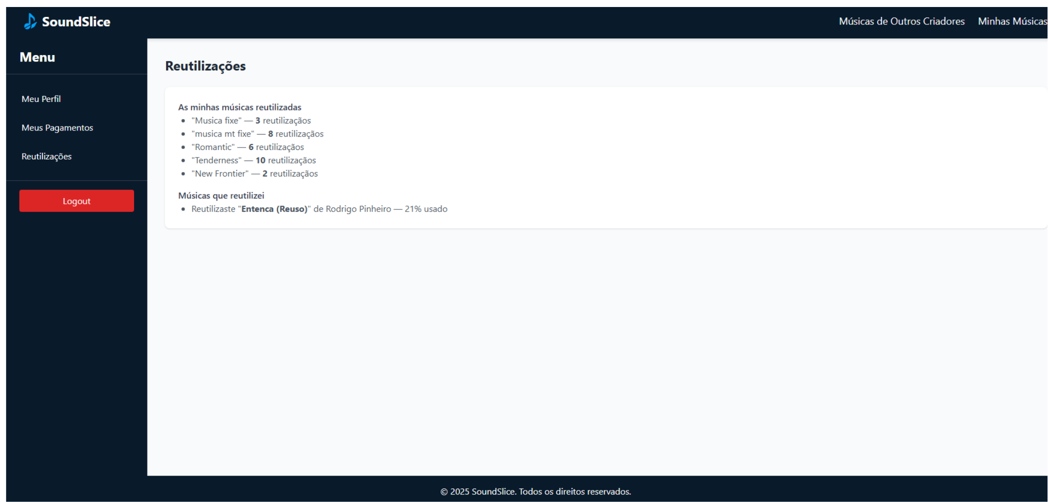


Figura 4.6: Interface da página do perfil – secção "Minhas Reutilizações".

4.1.4 Apresentação de Músicas

A funcionalidade de apresentação de músicas constitui um dos pilares centrais da plataforma SoundSlice, sendo responsável por listar, filtrar, reproduzir e gerir as obras musicais registadas pelos utilizadores. Esta secção compreende duas interfaces complementares:

- **Minhas Músicas** — página privada (`minhas-musicas.html`) onde o utilizador pode visualizar e gerir as suas próprias faixas;
- **Músicas de Outros Criadores** — página pública (`musicas-outros.html`) dedicada à exploração e reutilização de obras de outros artistas.

Os dados das músicas e os ficheiros associados são armazenados no MongoDB via GridFS e, nos casos de reutilização ou *mixes*, existe uma ligação direta com a camada de *smart contracts* na *blockchain* Ethereum (via Hardhat).

Página "Minhas Músicas"

A página `minhas-musicas.html` apresenta todas as músicas registadas pelo utilizador autenticado. Cada faixa é renderizada dinamicamente em formato de *card*, incluindo:

- imagem de capa obtida do GridFS;
- título e nome do artista;
- tipo da obra (original, reutilização ou *mix*);

- controlos de reprodução do áudio;
- e botões para edição, contrato e eliminação.

A estrutura base de cada cartão é gerada pela função `card()`, ilustrada abaixo na Listagem 4.12.

```
function card(m) {
  const el = document.createElement('div');
  el.className = 'card bg-white rounded-lg overflow-hidden shadow'
      ;
  const tipo = m.type === 'mix' ? 'Mix Musical'
             : m.type === 'reuse' ? 'Reutilização'
             : 'Original';
  el.innerHTML = `
    <div class="h-40 relative">
      <img src="${API}/api/tracks/${m._id}/cover"
           class="h-40 w-full object-cover"
           onerror="this.src='images/music-placeholder.jpg'">
      <span class="absolute top-2 left-2 text-xs bg-black/60 text-
          white px-2 py-1 rounded">
        ${m.visibility === 'public' ? 'Pública' : 'Privada'}
      </span>
    </div>
    <div class="p-4">
      <h4 class="font-semibold">${m.title}</h4>
      <span class="inline-block text-xs px-2 py-1 rounded bg-
          indigo-600 text-white">${tipo}</span>
      <audio controls class="w-full mt-3"></audio>
      <div class="flex gap-2 mt-3">
        <button onclick="editTrack('${m._id}')">Editar</button>
        <button onclick="downloadContract('${m._id}', '${m.type}')
            ">Contrato</button>
        <button onclick="openDeleteModal('${m._id}')">Apagar</
            button>
      </div>
    </div>`;
  const audioEl = el.querySelector('audio');
  loadAudio(m._id, audioEl);
  return el;
}
```

Listagem 4.12: Renderização dinâmica das músicas do utilizador.

A Listagem 4.13 ilustra o carregamento do ficheiro de áudio através do *endpoint* `/api/tracks/:id/file`, utilizando autenticação JWT e um fluxo binário do tipo *Binary Large Object* (BLOB).

```
async function loadAudio(trackId, audioEl) {
  const token = localStorage.getItem("authToken");
  const res = await fetch(`${API}/api/tracks/${trackId}/file`, {
    headers: { Authorization: "Bearer " + token }
  });
  const blob = await res.blob();
  audioEl.src = URL.createObjectURL(blob);
}
```

Listagem 4.13: Carregamento e reprodução do ficheiro áudio.

A Figura 4.7 apresenta a interface da página “Minhas Músicas”.

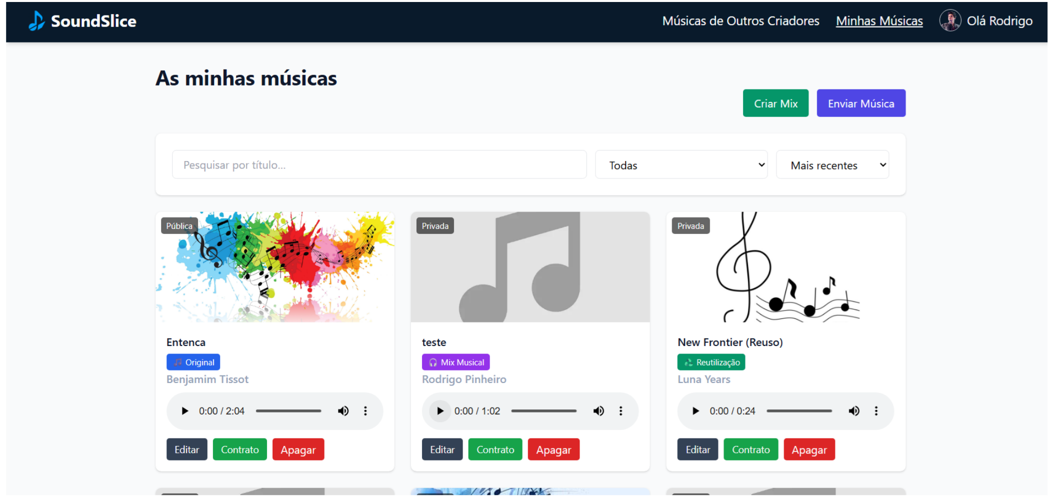


Figura 4.7: Interface da página das músicas do utilizador (`minhas-musicas.html`).

Nesta página o utilizador pode ainda efetuar:

- **Edição de Músicas** — Clicando no botão “Editar”, o utilizador é redirecionado para a página `editar.html`, cuja interface é apresentada na Figura 4.8. Esta permite alterar o título, o artista, a visibilidade e a imagem de capa de uma música previamente registada, preservando os metadados e o contrato inteligente associado.

- ***Download* de Contratos** — Clicando no botão “Contrato”, o utilizador pode descarregar o contrato associado a cada música — original, reutilização ou *mix* — em formato *Portable Document Format* (PDF). A função `downloadContract()` da Listagem 4.14 invoca diferentes *endpoints*, consoante o tipo da música, para descarregar o PDF do contrato.

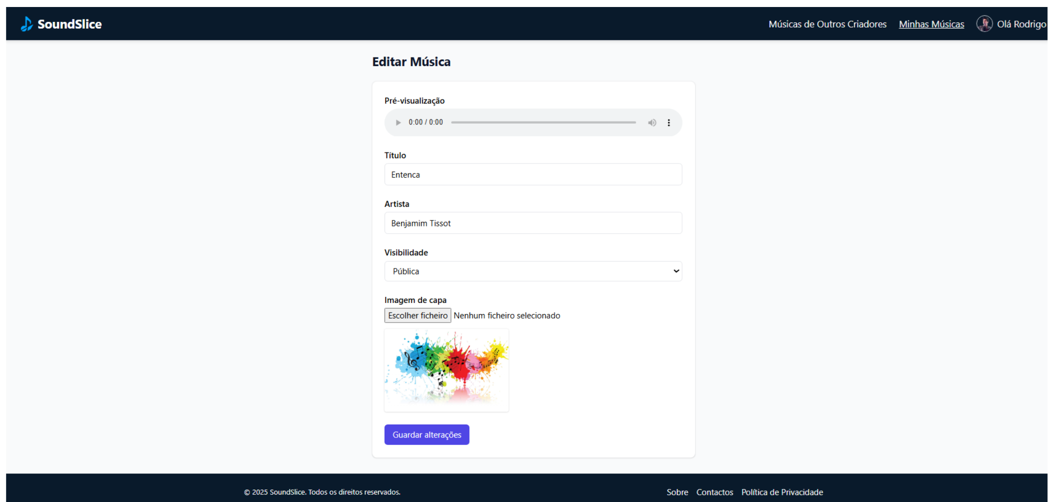


Figura 4.8: Interface da página de edição de dados das músicas (`editar.html`).

```
window.downloadContract = async (id, type = "original") =>
    {
  const token = localStorage.getItem("authToken");
  const url =
    type === "reuse"
      ? `${API}/api/tracks/${id}/reuse/contract-pdf`
      : type === "mix"
        ? `${API}/api/mix/${id}/contract-pdf`
        : `${API}/api/tracks/${id}/contract-pdf`;

  const res = await fetch(url, { headers: { Authorization:
      `Bearer ${token}` }});
  const blob = await res.blob();
  const blobUrl = URL.createObjectURL(blob);
  const a = document.createElement("a");
  a.href = blobUrl;
  a.download = `contrato-${id}.pdf`;
  a.click();
};
```

Listagem 4.14: *Download* de contrato inteligente.

Estes contratos são gerados no *backend* através do módulo `PDFKit`, incorporando dados do *smart contract* (endereços, percentagens, montante em ETH e *hash* da transação).

- **Eliminação de Músicas** — Clicando no botão “Apagar”, o utilizador pode remover a música associada. A eliminação de músicas é implementada com confirmação modal, enviando um pedido `DELETE` para `/api/tracks/:id`.

Página "Músicas de Outros Criadores"

A página `musicas-outros.html` disponibiliza o catálogo público da plataforma, apresentando faixas que podem ser ouvidas e reutilizadas por outros utilizadores. O seu funcionamento é semelhante ao da página anterior, mas sem necessidade de autenticação obrigatória. Cada faixa é apresentada com a capa, nome do artista, género musical e um botão de reprodução. O botão "Reutilizar" redireciona o utilizador para a página `reuse.html`, iniciando o processo de licenciamento e geração de um novo contrato inteligente. Assim, o utilizador pode proceder à:

- **Aplicação de Filtros e Paginação** — A página oferece filtros por título, criador, género musical e ordenação (recentes, populares ou alfabética). As faixas são obtidas através do *endpoint* público `/api/tracks/feed` (ver Listagem 4.15).

```
1     async function fetchFeed({ q = '', genre = '', sort = '
        recent' }) {
2       const r = await fetch(`${API}/api/tracks/feed?q=${
          encodeURIComponent(q)}&genre=${encodeURIComponent(
          genre)}&sort=${sort}`);
3       if (!r.ok) return [];
4       return await r.json();
5     }
```

Listagem 4.15: Obtenção das músicas públicas.

  A renderização é feita progressivamente (8 faixas por página), permitindo uma experiência fluida mesmo em catálogos extensos.

- **Integração com a Reutilização e *Smart Contracts*** — Pressionar o botão "Reutilizar" para iniciar o fluxo de criação de um novo contrato inteligente entre o criador original e o reutilizador. O *backend* gere a extração dos metadados da faixa, gera a proposta de contrato (com valores proporcionais à percentagem de utilização) e encaminha esses dados para a página `reuse.html`, onde o utilizador pode selecionar o trecho a reutilizar. A Figura 4.9 apresenta a interface da página "Músicas de Outros Criadores".

4.1.5 *Upload* de Músicas

A funcionalidade de *upload* de músicas constitui o ponto de entrada do fluxo de registo de músicas na plataforma SoundSlice. É a partir desta página que o utilizador insere uma nova obra musical, define as suas propriedades (título, artista, género, visibilidade e preço base), faz o carregamento dos ficheiros de áudio e imagem de

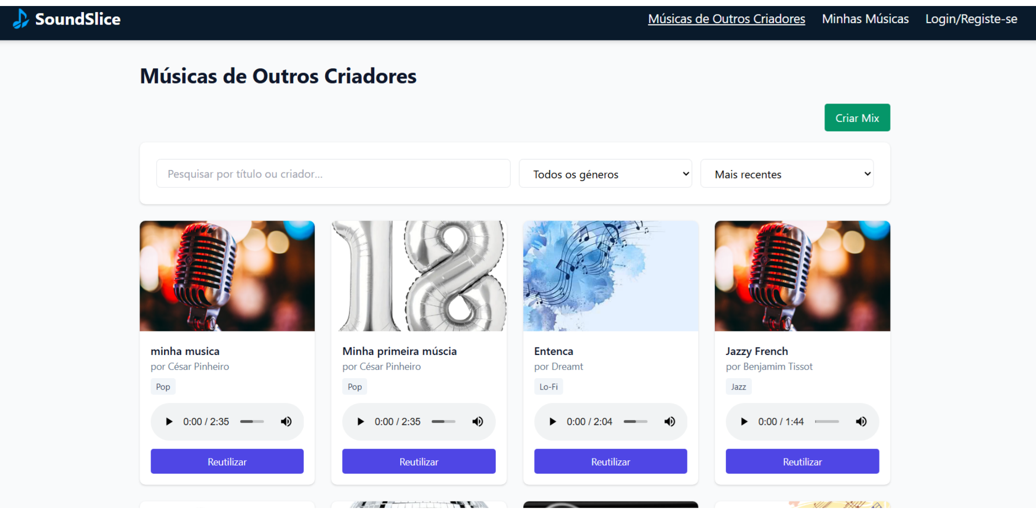


Figura 4.9: Interface da página das músicas dos outros utilizadores (`musicas-outros.html`).

capa, e desencadeia a criação automática do contrato inteligente correspondente na *blockchain*.

Interface e Estrutura do Formulário

O formulário principal (`uploadForm`) recolhe todos os dados necessários para o registo de uma nova música. Os campos obrigatórios incluem título, artista, género musical, ficheiro de áudio, visibilidade e preço base em ETH. Além destes, é possível definir co-titulares de direitos e carregar uma imagem de capa opcional.

O campo *preview* permite ouvir o ficheiro antes do envio, sendo os metadados técnicos (formato, duração e *hash*) automaticamente preenchidos após a análise do ficheiro.

Extração Automática de Metadados

Assim que o utilizador seleciona um ficheiro de áudio, este é enviado para o *endpoint* `/api/tracks/metadata`, onde o *backend* utiliza as bibliotecas `music-metadata` e `ffmpeg` para extrair propriedades técnicas, incluindo:

- formato e codificador/descodificador (codec) – MPEG *Audio Layer* III (MP3), *Waveform Audio File Format* (WAV), *Free Lossless Audio Codec* (FLAC), etc.;
- duração em segundos;
- taxa de amostragem e *bitrate*;
- Secure Hash Algorithm-256 bits (SHA-256) usado para identificação na *blockchain*;

- e metadados adicionais embutidos (álbum, género, ano, etc.).

A Listagem 4.16 apresenta obtenção dos metadados a partir da resposta JSON do *backend*.

```
document.getElementById("file").addEventListener("change", async (
    e) => {
  const file = e.target.files[0];
  const formData = new FormData();
  formData.append("file", file);
  const r = await fetch(`${API}/api/tracks/metadata`, { method: "
      POST", body: formData });
  const data = await r.json();
  document.getElementById("metadataPreview").textContent = JSON.
      stringify(data, null, 2);
  document.getElementById("format").value = data.format;
  document.getElementById("duration").value = data.duration;
});
```

Listagem 4.16: Extração e visualização dos metadados do ficheiro.

Desta forma, o utilizador tem acesso aos atributos técnicos da música antes de a registar na *blockchain*.

Interação com a *Blockchain* e Criação do Contrato

Após o preenchimento do formulário, o evento `submit` do formulário desencadeia um processo duplo:

1. Criação do contrato inteligente no ambiente Hardhat (via `ethers.js`);
2. *Upload* dos ficheiros e registo dos dados no servidor *backend*.

A ligação à rede é feita através do *provider* da carteira digital MetaMask, utilizando a rede local (`chainId 0x7A69`) (ver Listagem 4.17).

```
const provider = new ethers.BrowserProvider(window.ethereum);
const accounts = await provider.send("eth_requestAccounts", []);
const signer = await provider.getSigner(accounts[0]);
```

Listagem 4.17: Configuração da ligação à *blockchain*.

O contrato inteligente é instanciado a partir do seu *Application Binary Interface* (ABI) e `bytecode` e recebe como parâmetros os atributos da música e os respetivos titulares (consultar Listagem 4.18).

```
1 const factory = new ethers.ContractFactory(CONTRACT_ABI,
      CONTRACT_BYTECODE, signer);
2 const contract = await factory.deploy(
3   title, artist, fileHash, priceWei, format, genre, duration,
        extra, holders, shares
4 );
5 await contract.waitForDeployment();
6 const contractAddress = await contract.getAddress();
7 console.log("Contrato criado:", contractAddress);
```

Listagem 4.18: Criação do contrato inteligente na *blockchain*.

Este contrato representa a entidade digital da música na *blockchain*, armazenando dados imutáveis como:

- título e artista;
- *hash* do ficheiro;
- preço base em ETH;
- género, formato e duração;
- titulares e percentagens de partilha.

No momento da criação, é emitido um evento na *blockchain* com o endereço do contrato, permitindo ao *backend* associar essa informação ao documento correspondente na base de dados.

Upload e Registo no *Backend*

Concluída a criação do contrato inteligente, é preparado um objeto `FormData` com todos os elementos do formulário e enviado para o *endpoint* `/api/tracks/upload`.

Este *endpoint* executa três operações no *backend*:

- armazenamento do ficheiro áudio e imagem de capa no GridFS;
- persistência dos metadados e do endereço do contrato no MongoDB;
- atualização do índice de músicas associadas ao utilizador.

A resposta positiva do servidor confirma o sucesso do envio e redireciona o utilizador para a página `minhas-musicas.html`, onde a nova faixa aparece imediatamente listada.

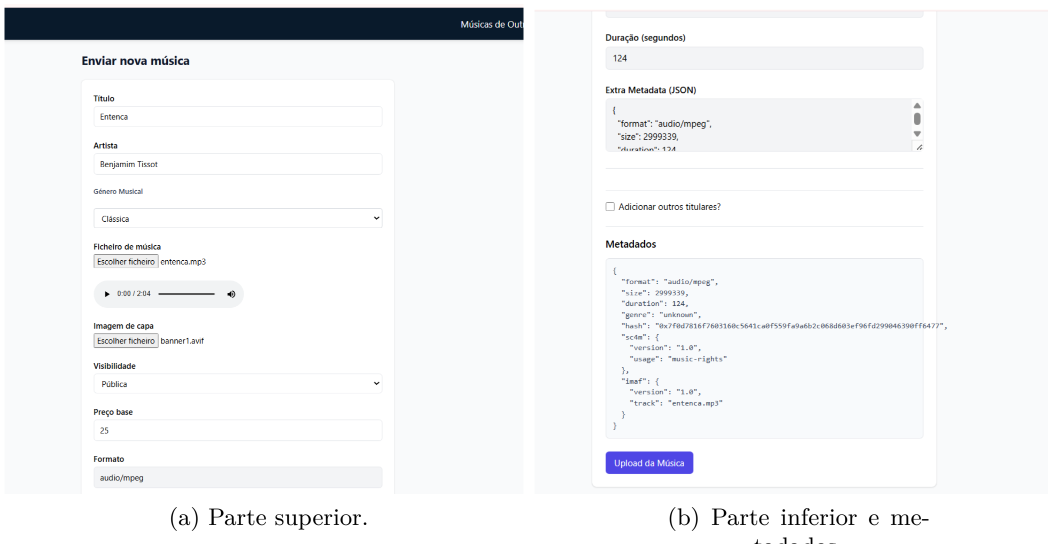
Gestão de Co-Titulares e Direitos Partilhados

A secção de co-titulares permite adicionar outros criadores e respetivas percentagens de partilha, simulando uma divisão de direitos de autor. A interface é dinâmica e permite adicionar ou remover campos conforme necessário (Listagem 4.19).

```
1 addHolderBtn.addEventListener("click", () => {
2   const div = document.createElement("div");
3   div.innerHTML = `
4     <input type="text" placeholder="Nome do titular" class="
          holderName"/>
5     <input type="number" placeholder="%" class="holderPercent"/>
6     <button class="removeHolder">X</button>`;
7   div.querySelector(".removeHolder").onclick = () => div.remove();
8   holdersList.appendChild(div);
9 });
```

Listagem 4.19: Gestão dinâmica de co-titulares.

Cada titular adicional é registado na *blockchain* através dos arrays `holders[]` e `shares[]`, garantindo a rastreabilidade e a remuneração proporcional em futuras reutilizações ou *mixes*. As Figuras 4.10a e 4.10b apresentam a interface do formulário de *upload* de músicas.



(a) Parte superior.

(b) Parte inferior e metadados.

Figura 4.10: Formulário de envio de música.

4.1.6 Reutilização de Músicas

A reutilização de músicas é uma das funcionalidades mais relevantes da plataforma SoundSlice porque materializa o objetivo principal do sistema ao permitir que um

utilizador reutilize parcial ou totalmente uma obra musical registada, garantindo simultaneamente a compensação automática dos titulares de direitos.

A interface é composta por um *player* de áudio e um seletor visual (*waveform slider*) que permite ao utilizador definir o início e o fim do trecho a reutilizar. Esta seleção é representada por um retângulo azul com controlos laterais (`handle-left` e `handle-right`) que podem ser arrastados para ajustar o intervalo desejado.

A função `updateSelectionUI()` calcula em tempo real valor a pagar pela reutilização de um trecho de música (V_{trecho}) em função do valor base da faixa (V_{faixa}) e do rácio entre a duração do trecho (Δt_{trecho}) e a duração total da faixa original (Δt_{faixa}), conforme a Equação 4.1.

$$V_{trecho} = V_{faixa} \times \frac{\Delta t_{trecho}}{\Delta t_{faixa}} \tag{4.1}$$

Comunicação com o *Backend*

Quando o utilizador confirma a operação, a aplicação envia uma requisição `POST` para o *endpoint* `/api/tracks/:id/reuse/select`, com os instantes de início e fim do trecho selecionado. O servidor realiza a análise e devolve um objeto JSON com:

- o valor a pagar (`value`) em wei[1];
- o endereço do criador original (`payTo`);
- a percentagem de reutilização;
- o identificador e nome do *snippet* gerado.

Interação com a *Blockchain*

O *frontend* utiliza a biblioteca `ethers.js` para estabelecer ligação com a carteira MetaMask do utilizador e proceder à transação financeira. Após a confirmação do pagamento na *blockchain*, é efetuada uma segunda chamada `POST` para o *endpoint* `/api/tracks/:id/reuse/confirm`, que regista permanentemente a reutilização e associa os metadados do novo *snippet* ao utilizador que realizou a operação.

Armazenamento e Persistência

Os dados do trecho reutilizado são guardados tanto na base de dados MongoDB como localmente no navegador (através de `localStorage`), o que permite que o utilizador insira as reutilizações num processo de criação de *mixes*.

[1] $1\,\text{ETH} = 1 \times 10^{18}$ wei.

Integração com Contratos Inteligentes

Cada reutilização é associada ao contrato inteligente `MusicReuse.sol`, responsável por assegurar a correspondência entre o pagamento realizado e a atualização do número total de reutilizações da obra original. Este contrato, descrito em detalhe na Secção 4.3, é acionado automaticamente após a confirmação da transação, garantindo a imutabilidade e transparência do registo. A Figura 4.11 apresenta a interface do processo de reutilização de músicas.

Figura 4.11: Interface da página de reutilização de músicas, com a seleção de trecho e cálculo automático do valor a pagar.

4.1.7 Criação de *Mixes*

A página `editor.html`, designada Editor de *Mix*, representa a fase final do processo criativo na plataforma SoundSlice. Nesta etapa, o utilizador pode combinar múltiplos trechos de áudio (*snippets*) previamente registados, quer provenientes de reutilizações de obras de outros criadores, quer de músicas próprias, gerando uma nova composição musical. A interface, ilustrada na Figura 4.12, é constituída por três componentes principais:

- **Músicas adicionadas** — secção que lista os trechos incluídos na *mix*;
- ***Player* de pré-audição** — que permite ouvir o resultado final concatenado;
- **Nome da *Mix*** — campo onde o utilizador atribui um título à nova composição.

O utilizador pode adicionar trechos de duas origens distintas: músicas de outros criadores (através do botão `Adicionar Música`) ou músicas pessoais (`Adicionar das Minhas Músicas`) que direciona para a página `minhas-musicas-mix.html`. Em ambos os casos, o sistema redireciona o utilizador para as respetivas páginas de seleção, guardando temporariamente no `localStorage` o parâmetro `fromEditor`, para permitir o regresso automático ao editor após a escolha do trecho.

A página `minhas-musicas-mix.html` é estruturalmente semelhante à página `minhas-musicas.html`, mas com um comportamento adaptado. Em vez de apresentar opções de edição ou eliminação, permite selecionar uma música do próprio utilizador para ser incluída diretamente no editor de *mixes*. Desta forma, a interface é otimizada para o contexto de criação, eliminando ações irrelevantes e simplificando o processo de seleção.

Cada *snippet* é representado por um objeto JSON contendo informação detalhada, incluindo o título, artista, percentagem reutilizada, endereço do criador original e o URL do ficheiro de áudio. Estes dados são carregados localmente e apresentados dinamicamente através da função `renderSnippets()`, permitindo ao utilizador:

- remover trechos individualmente;
- reordenar a sequência dos trechos através da biblioteca `SortableJS`;
- pré-escutar o resultado final com um clique.

Construção do Áudio Final

A função `buildMixAudio()` é responsável por unir todos os trechos numa única faixa contínua, utilizando a API Web Audio. O sistema carrega cada ficheiro de áudio via `fetch()`, decodifica-o em *buffers* e concatena-os num novo *AudioBuffer* com base na taxa de amostragem do navegador. Por fim, o áudio combinado é convertido para o formato WAV através da função `bufferToWave()`, sendo disponibilizado no *player* principal para escuta imediata. Este processo é realizado inteiramente do lado do cliente, assegurando uma experiência fluida e sem necessidade de processamento intensivo no servidor.

Exportação e Integração com o *Backend*

Após a finalização da *mix*, o utilizador pode exportar o resultado através do botão "Exportar Mix". Ao clicar, é criado um objeto `FormData` que contém:

- o ficheiro de áudio final (`mix.wav`);
- o título da *mix*;
- e a lista completa de *snippets* utilizados.

Esta informação é enviada para o *endpoint* `/api/mix/export` no *backend*, que armazena o ficheiro no sistema GridFS do MongoDB, associando-o ao utilizador e gerando automaticamente um novo registo de faixa. Após a exportação bem-sucedida, a *mix* é apresentado na página `minhas-musicas.html`, juntamente com as restantes criações do utilizador. O armazenamento temporário no `localStorage`

é limpo e a aplicação redireciona automaticamente o utilizador para a listagem principal.

Cada *mix* é registada na *blockchain* como uma nova obra composta, cuja lógica é implementada no contrato `MixRights`. Este contrato referencia os endereços dos contratos originais associados a cada trecho, garantindo uma distribuição transparente das remunerações futuras em caso de reutilização adicional da *mix*.

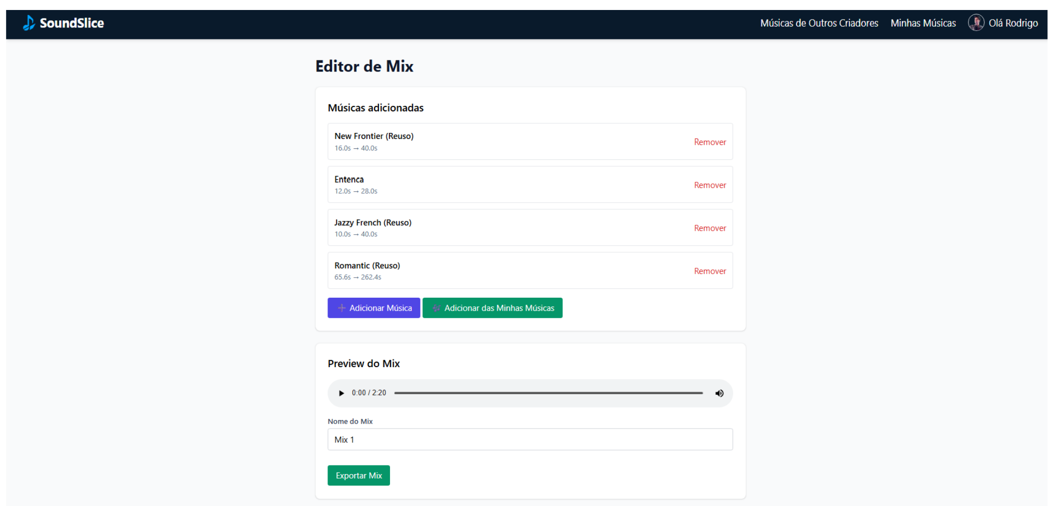


Figura 4.12: Interface da página de criação de *mixes*.

4.1.8 Páginas Adicionais

Para complementar as funcionalidades principais da plataforma SoundSlice, foram desenvolvidas um conjunto de páginas informativas e institucionais que asseguram a coerência visual e a completude da aplicação *web*. Estas páginas, ainda que não interajam diretamente com a *blockchain* ou com a base de dados, desempenham um papel essencial na experiência do utilizador, garantindo transparência, apoio e conformidade legal.

Página "Sobre"

A página `sobre.html`, ilustrada na Figura 4.13, apresenta uma descrição geral da plataforma e a sua proposta de valor. Explica o funcionamento base da SoundSlice, destacando:

- a publicação de músicas originais pelos criadores;
- a possibilidade de reutilização parcial de faixas de outros artistas;
- a remuneração automática dos titulares de direitos;

- e a transparência garantida pela utilização de contratos inteligentes.

A estrutura desta página segue o mesmo modelo das restantes, com cabeçalho (`header`) e rodapé (`footer`) comuns, mantendo a identidade visual da aplicação. O conteúdo é puramente informativo e serve como ponto de introdução para novos utilizadores. A Figura 4.13 apresenta a interface da página "Sobre a SoundSlice".

Figura 4.13: Página informativa "Sobre a SoundSlice".

Página "Contactos"

A página `contactos.html` disponibiliza os meios de contacto com a equipa da plataforma, incluindo email, telefone e morada. Adicionalmente, contém um formulário simples para o envio de mensagens, que poderá futuramente ser ligado a um *endpoint* no *backend* para comunicação direta com o suporte. O formulário é composto por três campos principais (nome, e-mail e mensagem) e um botão de envio estilizado com Tailwind CSS. A Figura 4.14 apresenta a interface da página de contactos.

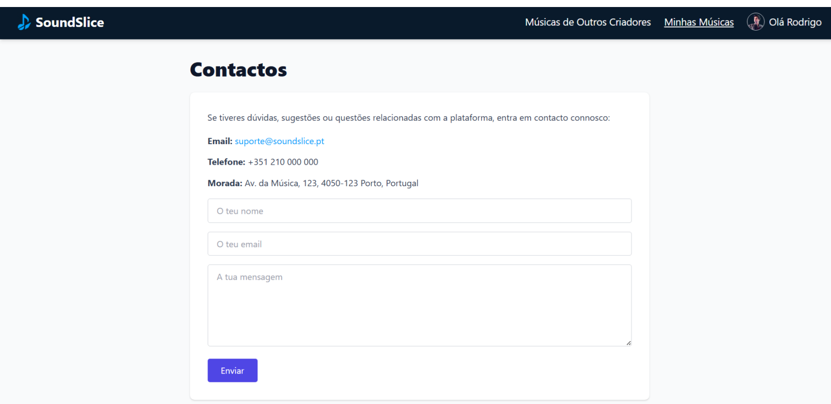


Figura 4.14: Página de contacto da plataforma.

Página "Política de Privacidade"

A página `polpriv.html` (Política de Privacidade) cumpre o papel legal de informar o utilizador sobre a recolha e o tratamento de dados pessoais. Nesta secção são apresentados os pontos fundamentais:

- os tipos de dados recolhidos (nome, email, dados de pagamento e interações);
- a forma como são utilizados para o funcionamento da plataforma;
- as medidas de segurança aplicadas;
- e os direitos de acesso, correção e eliminação dos dados do utilizador.

O conteúdo é apresentado de forma estruturada e acessível, permitindo a conformidade com as normas de proteção de dados. Assim como nas restantes páginas, mantém-se a consistência visual e o carregamento dinâmico da sessão do utilizador (verificação de autenticação e exibição do menu de perfil). A Figura 4.15 apresenta a interface da página de política de privacidade.

Figura 4.15: Página de políticas de privacidade da plataforma.

4.2 Implementação do *Backend*

Esta secção descreve a arquitetura do *backend*, detalhando a estrutura da API, os mecanismos de gestão de utilizadores, músicas, incluindo reutilizações e misturas, bem como a criação e execução dos *smart contracts*.

4.2.1 Estrutura Geral da API

O *backend* da plataforma SoundSlice foi desenvolvido em Node.js, utilizando o *framework* Express para a criação de uma API REST modular e escalável. A arquitetura segue um modelo em camadas, separando de forma clara a lógica de negócio,

a gestão de dados e os serviços externos, garantindo uma manutenção simples e uma integração direta com a *blockchain* e com o *frontend web*. A estrutura base do projeto é composta pelos seguintes diretórios:

- `controllers/` — contém os controladores responsáveis pelo processamento da lógica principal das operações (utilizadores, músicas e *mixes*);
- `routes/` — define os *endpoints* públicos da API e associa-os aos respetivos controladores;
- `models/` — inclui os esquemas de dados Mongoose para as coleções MongoDB;
- `middleware/` — contém o *middleware* de autenticação JWTs que protege as rotas privadas;
- `config/services/` — integra serviços externos como a camada de comunicação com a *blockchain* (`onchain.js`);
- `utils/` — agrega utilitários de suporte à execução de contratos inteligentes (`blockchainReuse.js` e `blockchainMix.js`);
- `server.js` — ponto de entrada da aplicação e responsável por iniciar o servidor e carregar as rotas principais.

Configuração e Iniciação do Servidor

O ficheiro `server.js` é responsável por iniciar a aplicação Express, configurar o `middleware` e estabelecer a ligação à base de dados MongoDB. O servidor é configurado para aceitar pedidos *Cross-Origin Resource Sharing* (CORS), processar pedidos JSON e interpretar formulários *multipart* enviados pelos formulários do *frontend*. O excerto da Listagem 4.20 apresenta a iniciação do servidor e a importação das rotas principais.

```
1  import express from "express";
2  import mongoose from "mongoose";
3  import dotenv from "dotenv";
4  import cors from "cors";
5  import userRoutes from "./routes/userRoutes.js";
6  import trackRoutes from "./routes/trackRoutes.js";
7  import mixRoutes from "./routes/mixRoutes.js";
8
9  dotenv.config();
10 const app = express();
11 app.use(cors());
12 app.use(express.json());
13 app.use(express.urlencoded({ extended: true }));
```

```
14
15 mongoose.connect(process.env.MONGO_URI)
16   .then(() => console.log("Ligação ao MongoDB estabelecida."))
17   .catch(err => console.error("Erro ao ligar ao MongoDB:", err));
18
19 app.use("/api/users", userRoutes);
20 app.use("/api/tracks", trackRoutes);
21 app.use("/api/mix", mixRoutes);
22
23 const PORT = process.env.PORT || 3000;
24 app.listen(PORT, () => console.log(`Servidor a correr na porta ${
       PORT}`));
```

Listagem 4.20: Iniciação do servidor Express e carregamento das rotas.

Modelos de Dados e Persistência

Cada entidade principal do sistema (utilizador, faixa musical, pagamento e *mix*) é representada por um modelo Mongoose, definido no diretório `models/`. Estes modelos especificam os campos, tipos de dados e relações entre coleções.

Middleware de Autenticação

Para proteger as rotas privadas, como o carregamento e edição de músicas, é utilizado um *middleware* de autenticação baseado em JWTs. O ficheiro `auth.js`, presente em `middleware/`, valida o *token* de acesso enviado no cabeçalho `Authorization: Bearer <token>` e anexa a informação do utilizador autenticado ao objeto `req`. O código simplificado encontra-se na Listagem 4.21.

```
1  import jwt from "jsonwebtoken";
2
3  export function verifyToken(req, res, next) {
4    const token = req.headers.authorization?.split(" ")[1];
5    if (!token) return res.status(401).json({ error: "Token não
         fornecido" });
6
7    try {
8      const decoded = jwt.verify(token, process.env.JWT_SECRET);
9      req.userId = decoded.id;
10     next();
11   } catch (err) {
12     return res.status(403).json({ error: "Token inválido ou
          expirado" });
13   }
14 }
```

Listagem 4.21: *Middleware* de autenticação JWT.

Este *middleware* é aplicado nas rotas que exigem autenticação, como o *upload* de músicas, a criação de *mixes* e a consulta do perfil do utilizador, garantindo que apenas utilizadores registados têm acesso a essas funcionalidades.

Serviços Auxiliares e Integração Externa

No diretório `config/services/`, o ficheiro `onchain.js` estabelece a ligação entre o servidor Node.js e o ambiente de execução Hardhat, onde são implantados os contratos inteligentes. Este serviço encapsula a lógica de comunicação com a *blockchain* e é reutilizado em diferentes controladores, como os responsáveis pela reutilização e criação de *mixes*. A sua implementação é detalhada na Secção 4.3.

4.2.2 Gestão de Utilizadores

Esta camada foi desenvolvida em Node.js e Express, recorrendo ao MongoDB para a persistência de dados e a JWTs para a autenticação segura.

Modelo de Dados

O modelo `User.js` define a estrutura de armazenamento dos perfis, utilizando o Mongoose como camada de abstração do MongoDB. Cada documento de utilizador contém informação pessoal, dados de autenticação e metadados associados à aceitação dos termos de utilização e política de privacidade. A Listagem 4.22 apresenta o esquema de dados principal.

```
1  const mongoose = require('mongoose');
2
3  const userSchema = new mongoose.Schema({
4    firstname: String,
5    lastname: String,
6    artistName: String,
7    dob: Date,
8    email: { type: String, unique: true },
9    password: String,
10   phone: String,
11   country: String,
12   city: String,
13   ethWallet: { type: String },
14   profilePic: { type: mongoose.Schema.Types.ObjectId, ref: '
       uploads.files' },
15   acceptPayments: Boolean,
16   terms: Boolean,
```

```
17    privacy: Boolean,
18    resetPasswordToken: String,
19    resetPasswordExpires: Date
20  }, { timestamps: true });
21
22  module.exports = mongoose.model('User', userSchema);
```

Listagem 4.22: Modelo de dados do utilizador.

Este modelo permite a associação de uma fotografia de perfil armazenada em GridFS, bem como o registo da carteira digital do utilizador, essencial para as operações de pagamento e integração com os contratos inteligentes.

Registo de Utilizadores

O processo de registo é gerido pela função `registerUser`, que valida os dados obrigatórios, encripta a palavra-passe com *bcrypt*, armazena opcionalmente a fotografia de perfil no GridFS e cria o documento correspondente na base de dados. A verificação dos campos de consentimento (`terms` e `privacy`) garante que o utilizador aceita explicitamente as políticas da plataforma. O excerto da Listagem 4.23 ilustra a lógica de registo.

```
1  const registerUser = async (req, res) => {
2    const { firstname, lastname, dob, email, password, country, city
       ,
3            phone, ethWallet, acceptPayments, terms, privacy } = req
               .body;
4    const termsBool = terms === "true" || terms === true;
5    const privacyBool = privacy === "true" || privacy === true;
6
7    if (!firstname || !lastname || !dob || !email || !password || !
       termsBool || !privacyBool)
8      return res.status(400).json({ error: "Campos obrigatórios em
         falta." });
9
10   const existing = await User.findOne({ email });
11   if (existing) return res.status(400).json({ error: "Email já
       registado." });
12
13   const hashedPassword = await bcrypt.hash(password, 10);
14   const user = new User({
15     firstname, lastname, dob, email,
16     password: hashedPassword, country, city, phone, ethWallet,
17     acceptPayments: acceptPayments === "true", terms: termsBool,
         privacy: privacyBool
18   });
19
```

```
20    await user.save();
21    res.status(201).json({ message: "Utilizador registado com
        sucesso!" });
22  };
```

Listagem 4.23: Processo de registo de um novo utilizador.

Se, durante o registo, o utilizador submeter uma fotografia, esta é inserida no *bucket* `profilePics` do GridFS, sendo posteriormente referenciada pelo identificador `ObjectId` armazenado no campo `profilePic`.

Autenticação e Sessões

A autenticação é efetuada através do *endpoint* `/api/users/login`, que compara a palavra-passe introduzida com o *hash* armazenado e gera um JWT válido por sete dias. O *token* contém o identificador do utilizador e é devolvido ao cliente, sendo utilizado em todos os pedidos subsequentes que exijam autenticação. A Listagem 4.24 ilustra este processo.

```
1  const loginUser = async (req, res) => {
2    const { email, password } = req.body;
3    const user = await User.findOne({ email });
4    if (!user) return res.status(401).json({ error: "Credenciais inv
       álidas" });
5
6    const isMatch = await bcrypt.compare(password, user.password);
7    if (!isMatch) return res.status(401).json({ error: "Credenciais
       inválidas" });
8
9    const token = jwt.sign(
10     { id: user._id.toString(), firstname: user.firstname, email:
         user.email },
11     process.env.JWT_SECRET, { expiresIn: "7d" }
12   );
13
14   res.json({ message: "Login bem-sucedido!", token });
15 };
```

Listagem 4.24: Autenticação do utilizador e emissão do *token* JWT.

O *middleware* `auth.js` é responsável por validar o *token* recebido e anexar o identificador do utilizador ao objeto `req`, permitindo ao servidor identificar o autor de cada requisição. O utilizador autenticado pode consultar o seu perfil através do *endpoint* `/api/users/me`, que devolve os dados básicos e o URL da imagem de perfil.

Segurança e Conformidade

As palavras-passe são encriptadas no *backend* utilizando o algoritmo `bcrypt`, impedindo que sejam armazenadas ou transmitidas em texto simples. Durante o processo de autenticação, o sistema compara o *hash* armazenado com o resultado da encriptação da palavra-passe introduzida, garantindo a confidencialidade das credenciais.

Os endereços de carteiras Ethereum (`ethWallet`) são tratados como identificadores públicos, não sendo possível realizar operações financeiras sem a intervenção explícita do utilizador através da sua carteira pessoal (MetaMask).

Todos os ficheiros carregados (áudios, imagens e trechos) são armazenados em GridFS, o que permite controlar o acesso e garantir a integridade dos dados binários. Quando envolvem o acesso a conteúdos privados, as rotas de *download* e *streaming* incluem a verificação de permissões e requerem a autenticação via *token* JWT.

Atualização de Perfil

A função `updateUser` permite que o utilizador altere os seus dados pessoais e substitua a fotografia de perfil. A nova imagem é enviada para o GridFS e o campo `profilePic` é atualizado com o identificador do ficheiro. O sistema devolve a versão atualizada do perfil, conforme o exemplo da Listagem 4.25.

```
1  const updateUser = async (req, res) => {
2    const userId = req.user.id;
3    const updates = {};
4
5    if (req.body.firstname) updates.firstname = req.body.firstname;
6    if (req.body.phone) updates.phone = req.body.phone;
7    if (req.file) {
8      const gfs = new mongoose.mongo.GridFSBucket(mongoose.
           connection.db, {
9        bucketName: "profilePics",
10     });
11     const profilePicId = new mongoose.Types.ObjectId();
12     const uploadStream = gfs.openUploadStreamWithId(
13       profilePicId, req.file.originalname, { contentType: req.file
             .mimetype }
14     );
15     uploadStream.end(req.file.buffer);
16     updates.profilePic = profilePicId;
17   }
18
19   const updatedUser = await User.findByIdAndUpdate(userId, { $set:
         updates }, { new: true });
20   res.json({ message: "Perfil atualizado com sucesso!", user:
         updatedUser });
```

```
21 };
```

Listagem 4.25: Atualização do perfil de utilizador.

Pagamentos e Reutilizações

Para além da gestão de conta, o módulo do utilizador disponibiliza *endpoints* para consultar a atividade financeira e as reutilizações associadas às suas obras. A função `getUserPayments` analisa as faixas reutilizadas por terceiros, calcula a percentagem de reutilização e o valor correspondente em ETH, com base no preço base definido para cada música. Já a função `getUserReuses` devolve duas listas distintas: (*i*) `ReusedByOthers` — músicas originais do utilizador que foram reutilizadas por outros criadores, com o total de reutilizações; e (*ii*) `MyReuses` — músicas criadas pelo utilizador que reutilizam obras de terceiros, incluindo os respetivos autores originais e percentagens de reutilização.

Estes *endpoints* são essenciais para alimentar a secção de perfil do *frontend*, onde o utilizador pode visualizar os seus ganhos e as suas participações em obras colaborativas.

Rotas e *Middleware*

O ficheiro `userRoutes.js` associa todos os controladores aos *endpoints* públicos e privados. As operações que implicam autenticação utilizam o *middleware* `requireAuth`, enquanto o carregamento de imagens é tratado pelo `multer` em memória (ver Listagem 4.26).

```
1  const express = require('express');
2  const userController = require("../controllers/userController");
3  const requireAuth = require("../middleware/auth");
4  const multer = require("multer");
5  const upload = multer({ storage: multer.memoryStorage() });
6
7  const router = express.Router();
8
9  router.post('/register', upload.single("profilePic"),
       userController.registerUser);
10 router.post('/login', userController.loginUser);
11 router.get('/me', requireAuth, userController.me);
12 router.post('/logout', requireAuth, userController.logout);
13 router.put('/update', requireAuth, upload.single("profilePic"),
       userController.updateUser);
14 router.get('/payments', requireAuth, userController.
       getUserPayments);
15 router.get('/reuses', requireAuth, userController.getUserReuses);
```

```
16
17 module.exports = router;
```

Listagem 4.26: Estrutura das rotas de utilizador.

4.2.3 Gestão de Músicas

O módulo de gestão de músicas da plataforma SoundSlice é responsável por controlar o ciclo de vida completo das faixas musicais, ou seja desde o carregamento (*upload*) e extração de metadados, até à edição, *streaming*, reutilização e integração com contratos inteligentes. Esta componente interliga a base de dados MongoDB (com suporte GridFS) com o sistema de ficheiros, o motor de processamento FFmpeg, e a camada *on-chain* implementada em Ethereum/Hardhat.

Modelos de Dados das Faixas Musicais

Cada faixa é representada pelo modelo `Track.js`, cuja estrutura foi planeada para suportar tanto músicas originais como reutilizações e *mixes*. A Listagem 4.27 apresenta o esquema de dados, com destaque para os campos `metadata` (armazenando o JSON SC4M/IMAF) e `contractAddress`, que regista o contrato inteligente associado a cada obra.

```
1  const trackSchema = new mongoose.Schema({
2    title: { type: String, required: true },
3    artist: { type: String, required: true },
4    visibility: { type: String, enum: ['public', 'private'], default
       : 'public' },
5    fileId: { type: mongoose.Schema.Types.ObjectId, ref: 'uploads.
       files' },
6    coverId: { type: mongoose.Schema.Types.ObjectId, ref: 'uploads.
       files' }, // GridFS
7    metadata: { type: Object }, // JSON SC4M/IMAF
8    contractAddress: { type: String }, // Ethereum
9    createdAt: { type: Date, default: Date.now },
10   user: { type: mongoose.Schema.Types.ObjectId, ref: 'User',
       required: true },
11   originalOwner: { type: mongoose.Schema.Types.ObjectId, ref: "
       User" },
12   coholders: [
13     {
14       name: String,
15       percentage: Number
16     }
17   ],
```

```
18    reusedFrom: { type: mongoose.Schema.Types.ObjectId, ref: 'Track'
          },
19    reusePercentage: { type: Number },
20    totalReuses: { type: Number, default: 0 },
21    type: {
22      type: String,
23      enum: ["original", "reuse", "mix"],
24      default: "original"
25    }
26  }, { timestamps: true });
27
28  module.exports = mongoose.models.Track || mongoose.model("Track",
      trackSchema);
```

Listagem 4.27: Modelo de dados da música.

Este modelo permite distinguir entre:

- músicas **originais**, carregadas pelo próprio criador;
- músicas **reutilizadas**, derivadas de faixas originais de outros autores, com percentagem e contrato específicos;
- e músicas do tipo ***mix***, resultantes da junção de múltiplos trechos.

Os ficheiros de áudio e de capa são armazenados em GridFS, identificados pelos campos `fileId` e `coverId`, garantindo escalabilidade e gestão eficiente de grandes volumes de dados binários.

Controlador de Músicas

O módulo `trackController.js` define as operações básicas de criação e consulta de músicas. O método `addTrack` permite adicionar novas faixas, enquanto `getMyTracks` devolve as músicas associadas ao utilizador autenticado.

```
1  const Track = require('../models/Track');
2  const addTrack = async (req, res) => {
3    try {
4      const { title, artist } = req.body;
5      const track = await Track.create({
6        title,
7        artist,
8        filePath: req.file ? req.file.path : undefined,
9        user: req.user.id
10     });
11
12     res.status(201).json({ message: 'Música adicionada!', track })
         ;
```

```
13    } catch (err) {
14      console.error(err);
15      res.status(500).json({ error: 'Erro ao adicionar música.' });
16    }
17  };
18
19  const getMyTracks = async (req, res) => {
20    try {
21      const tracks = await Track.find({ user: req.user.id }).sort({
          createdAt: -1 });
22      res.json(tracks);
23    } catch (err) {
24      console.error(err);
25      res.status(500).json({ error: 'Erro ao carregar músicas.' });
26    }
27  };
28
29  module.exports = { addTrack, getMyTracks };
```

Listagem 4.28: Controlador básico de criação e listagem de músicas.

Rotas e *Upload* de Ficheiros

O ficheiro `trackRoutes.js` implementa as rotas da API responsáveis pelo *upload*, extração de metadados, edição e gestão de reutilizações musicais. Cada *endpoint* tem uma função específica:

- `/metadata` — recebe um ficheiro temporário, extrai metadados técnicos (formato, duração, género, SHA-256) e gera o objeto SC4M/IMAF base;
- `/upload` — guarda o ficheiro e a capa em GridFS, regista o documento da música e define o preço base (`basePrice`) a ser usado em futuras reutilizações;
- `/mine` — lista todas as músicas originais do utilizador autenticado;
- `/:id/file` — realiza *streaming* do áudio diretamente a partir do GridFS, suportando `Range requests`;
- `/:id/reuse/*` — executa a lógica de reutilização, nomeadamente análise, corte de trechos, criação do contrato `MusicReuse` e emissão do PDF do contrato inteligente;
- `/feed` — devolve o catálogo público de faixas disponíveis para reutilização, filtrável por título, artista ou género.

Geração Automática do Contrato PDF

Após o carregamento e registo de uma música original, o sistema disponibiliza um *endpoint* dedicado à geração de um documento PDF que representa o contrato digital associado à obra. Este contrato reúne os principais dados técnicos e autorais da faixa, incluindo título, artista, formato, duração e o endereço do contrato inteligente. O documento segue o mesmo modelo visual dos restantes contratos da plataforma, apresentando um cabeçalho azul-escuro, corpo informativo e um código *Quick Response* (QR) com o identificador do contrato. Através deste PDF, o autor obtém uma representação formal e verificável da submissão da sua obra, assegurando a rastreabilidade e autenticidade do registo.

4.2.4 Gestão de Reutilizações

A gestão das reutilizações é implementada no módulo `trackRoutes.js` através de um conjunto de rotas encadeadas que implementam as diferentes fases do processo:

- Seleção e corte do trecho — o utilizador define o intervalo temporal (em segundos) que pretende reutilizar, sendo esse segmento extraído do ficheiro original e guardado no GridFS;
- Confirmação da reutilização — é criado um novo registo de faixa (`Track`) do tipo `reuse`, com metadados SC4M/IMAF que documentam a operação.

Seleção do Trecho

O *endpoint* `/api/tracks/:id/reuse/select` permite ao utilizador escolher um intervalo temporal da música original. O *backend* utiliza o FFmpeg para cortar o trecho de áudio entre os instantes definidos nos parâmetros `start` e `end`. O resultado é armazenado no GridFS como um novo ficheiro, cuja referência e *hash* são devolvidos ao *frontend*.

O código completo da função responsável pela seleção do trecho encontra-se no Apêndice A.

A resposta desta rota é enviada ao *frontend* com toda a informação necessária para a etapa seguinte: percentagem reutilizada, identificador do trecho e valor a pagar pela utilização parcial da obra.

Confirmação da Reutilização

Na segunda fase, o utilizador confirma a operação, sendo criado um novo registo na coleção `Track`. O novo documento é do tipo `reuse` e contém a referência à música original, bem como todos os metadados técnicos e autorais da reutilização – ver no Apêndice A.

Este *endpoint* consolida a reutilização: cria o novo registo, atualiza o contador de reutilizações da música original e associa o trecho previamente cortado. Os metadados SC4M e IMAF são preenchidos automaticamente, descrevendo:

- o identificador e a origem da obra original;
- o intervalo temporal reutilizado e a percentagem correspondente;
- o valor atribuído ao reuso;
- e as informações de autoria (criador e reutilizador).

As estruturas SC4M e IMAF foram aplicadas como modelos de representação dos metadados musicais e das relações de reutilização, permitindo compatibilidade futura com sistemas de licenciamento interoperáveis.

Geração Automática do Contrato PDF

Após a confirmação da reutilização, o sistema cria automaticamente um contrato PDF relativo à nova obra derivada. Este documento contém as informações principais sobre a reutilização, nomeadamente o título original, o reutilizador, a percentagem reutilizada, a duração do trecho e o valor pago pela licença. O contrato utiliza o mesmo modelo gráfico adotado nos restantes documentos da plataforma.

4.2.5 Criação e Exportação de *Mixes*

A criação de *mixes* é uma das funcionalidades mais distintivas da plataforma SoundSlice, permitindo ao utilizador combinar vários trechos de músicas reutilizadas numa nova composição. O *backend* implementa esta funcionalidade no módulo `mixRoutes.js`, que realiza o envio, armazenamento e registo das *mixes*. Cada *mix* é considerada uma nova entidade do tipo `Track`, com `type = "mix"`, contendo no campo `metadata.sc4m.reusedSnippets` a lista dos trechos musicais utilizados, a sua origem, percentagem de reutilização e valor atribuído. O processo compreende duas fases: a exportação e armazenamento do ficheiro final e a geração automática do contrato em formato PDF.

Exportação do Ficheiro *Mix*

O *endpoint* `POST /api/mix/export` recebe o ficheiro resultante da combinação de várias faixas e a lista dos *snippets* reutilizados. O ficheiro é armazenado no GridFS, e o sistema gera automaticamente os metadados SC4M e IMAF, descrevendo a composição final. O código completo da função responsável pela exportação e registo da *mix* encontra-se no Apêndice A.

Durante esta operação, o sistema calcula automaticamente o valor total das licenças (`totalValueEth`) com base nos preços base das faixas reutilizadas. Os

metadados SC4M documentam o uso de cada trecho e as respetivas percentagens, enquanto o bloco IMAF descreve a estrutura da *mix* e a ordem das faixas originais. O ficheiro resultante é armazenado no GridFS e associado ao utilizador autenticado, sendo classificado como uma *mix*. Este processo garante a rastreabilidade e a ligação entre as obras derivadas e as suas fontes originais.

Geração Automática do Contrato PDF

Após a criação da *mix*, o sistema disponibiliza um *endpoint* que gera automaticamente um documento PDF representando o contrato de composição. O documento apresenta os dados técnicos, autorais e financeiros da *mix*, incluindo a lista de trechos reutilizados, os respetivos autores e percentagens de reutilização. O contrato em PDF é gerado segundo o mesmo modelo visual dos outros contratos da plataforma. Inclui ainda o valor total das licenças e a lista detalhada de trechos utilizados, fornecendo ao utilizador uma representação documental e verificável da criação da *mix*.

4.3 Contratos Inteligentes

O código-fonte integral dos contratos inteligentes `MusicRights`, `MusicReuse` e `MusicMix` desenvolvidos encontra-se disponível no Anexo B, permitindo a consulta detalhada das estruturas, variáveis e funções implementadas em *Solidity*. Os contratos foram compilados com o compilador `solc` versão 0.8.20 e testados em ambiente local utilizando a rede `Hardhat`. Cada contrato corresponde a uma fase distinta do ciclo de vida musical,registo, reutilização e criação de composições derivadas, e foi implementado de modo a garantir a rastreabilidade e imutabilidade das transações armazenadas na *blockchain*.

4.3.1 *Deployment* dos Contratos Inteligentes

A camada *backend* do sistema SoundSlice foi concebida para permitir a interação direta com a *blockchain* através da biblioteca `ethers.js`. Esta integração assegura que, sempre que é registada uma nova música, reutilização ou *mix*, é criado de forma automática um contrato inteligente na rede, garantindo a imutabilidade e a rastreabilidade das transações.

Por um lado, o registo de uma nova música desencadeia a instanciação do contrato `MusicRights` no *frontend*, recorrendo à carteira digital pessoal do utilizador. Este modelo descentralizado foi adotado uma vez que representa o ato autoral original e deve ser controlado diretamente pelo criador.

Por outro lado, os contratos de reutilização e criação de composições derivadas são gerados automaticamente pelo *backend* da plataforma, promovendo: (*i*) segurança — evita a exposição de chaves privadas dos utilizadores, utilizando uma conta técnica controlada pelo servidor; (*ii*) automatização — a criação imediata dos contratos no momento em que uma reutilização ou uma *mix* é registada, dispensando a intervenção manual do utilizador; e (*iii*) consistência — todos os contratos derivados seguem o mesmo padrão estrutural e os metadados (SC4M e IMAF) são gerados e armazenados de forma uniforme. Nestes dois casos, o servidor Node.js utiliza a biblioteca `ethers.js` para interagir com a rede Ethereum local (Hardhat). Através da chave privada configurada como variável de ambiente `SERVER_PRIVATE_KEY`, é criada uma instância da carteira técnica do servidor, que vai ser responsável por assinar e enviar as transações de criação dos contratos. Cada operação de *deployment* segue a seguinte sequência:

1. Carregamento do *artifact* compilado (ABI e *bytecode*) a partir da pasta `/smart-contracts/artifacts`;
2. Criação de uma instância `ContractFactory`;
3. Preparação dos parâmetros com base nos metadados gerados durante o *upload*, reutilização ou exportação da *mix*;
4. Execução do método `deploy()` e espera pela confirmação do bloco;
5. Registo do endereço do contrato e *hash* da transação no MongoDB.

Configuração e Ligação à Rede

A ligação à *blockchain* é estabelecida a partir de variáveis de ambiente definidas no ficheiro `.env`, incluindo a chave privada do servidor e o URL do nó *Remote Procedure Call* (RPC) da rede simulada Hardhat. Esta configuração permite que o servidor Node.js execute transações autenticadas em nome da aplicação.

```
1 const provider = new ethers.JsonRpcProvider(process.env.
      BLOCKCHAIN_RPC);
2 const wallet = new ethers.Wallet(process.env.SERVER_PRIVATE_KEY,
      provider);
```

Listagem 4.29: Configuração da ligação à *blockchain*.

O `provider` define o ponto de comunicação com a rede, enquanto a variável `wallet` representa a conta que efetua o *deployment* e assina todas as transações.

Contrato `MusicRights`

O contrato `MusicRights`, responsável pelo registo original das obras, é criado diretamente no *frontend*, através da carteira digital MetaMask do utilizador. Esta decisão baseia-se no princípio de que o registo autoral deve ser efetuado diretamente pelo titular dos direitos, assegurando a transparência, autenticidade e propriedade efetiva da transação. Evita ainda que o servidor central assuma a identidade do autor ou tenha acesso direto a chaves privadas. Assim, o *frontend* é responsável por assinar e enviar a transação do *deployment* do contrato `MusicRights`, enquanto o *backend* apenas recebe e armazena os dados resultantes (endereço do contrato e *hash* da transação) para referência futura. A Listagem 4.30 exemplifica este processo.

```
1  const factory = new ethers.ContractFactory(abi, bytecode, signer);
2  const contract = await factory.deploy(title, artist, genre, format
       );
3  await contract.waitForDeployment();
4
5  await fetch("/api/tracks/update-contract", {
6      method: "POST",
7      body: JSON.stringify({
8          contractAddress: await contract.getAddress(),
9          txHash: contract.deploymentTransaction().hash
10     })
11 });
```

Listagem 4.30: *Deployment* do contrato `MusicRights` no *frontend*.

Contrato `MusicReuse`

O contrato `MusicReuse` é adicionado sempre que um utilizador confirma a reutilização de uma obra, após a seleção de um trecho e o cálculo do valor do pagamento. O *backend* recolhe os dados da obra original, do reutilizador, da percentagem de reutilização e do valor em ETH, executando o processo da Listagem 4.31.

```
1  const provider = new ethers.JsonRpcProvider(process.env.
       BLOCKCHAIN_RPC);
2  const wallet = new ethers.Wallet(process.env.SERVER_PRIVATE_KEY,
       provider);
3  const factory = new ethers.ContractFactory(artifact.abi, artifact.
       bytecode, wallet);
4
5  const contract = await factory.deploy({
6      reuseId: params.reuseId,
7      originalId: params.originalId,
8      originalTitle: params.originalTitle,
```

```
9       creatorWallet: params.creatorWallet,
10      reuserWallet: params.reuserWallet,
11      reusePercent: BigInt(params.reusePercent),
12      valuePaid: ethers.parseEther(params.valueEth.toString()),
13      originalFileHash: params.originalFileHash,
14      snippetHash: params.snippetHash,
15      genre: params.genre,
16      snippetDuration: BigInt(params.snippetDuration)
17  });
18  await contract.waitForDeployment();
19  const address = await contract.getAddress();
```

Listagem 4.31: *Deployment* do contrato `MusicReuse`.

Durante o processo, são ainda realizadas operações de validação interna para garantir a conformidade dos dados, nomeadamente através da função auxiliar `ensureBytes32()`, que assegura que os valores de *hash* fornecidos possuem o formato correto em hexadecimal (66 caracteres). O contrato é então assinado pela conta técnica do servidor e armazenado na rede Hardhat local, retornando os dados necessários para atualizar o documento da reutilização na base de dados. Esta abordagem permite que o contrato de reutilização seja criado sem qualquer intervenção do utilizador, automatizando o registo das condições autorais e financeiras associadas à reutilização de um trecho musical.

Contrato `MusicMix`

O contrato `MusicMix` é responsável por representar na *blockchain* as composições resultantes da combinação de múltiplos trechos reutilizados. Após o utilizador exportar uma *mix* no editor de áudio, o *backend* calcula a duração total, a soma dos valores de licenciamento e as percentagens de reutilização de cada trecho incluído. Com base nestes dados, o servidor executa automaticamente o *deployment* do contrato, seguindo uma lógica semelhante à utilizada no `MusicReuse` (ver Listagem 4.32).

```
1  const wallet = new ethers.Wallet(process.env.SERVER_PRIVATE_KEY,
       provider);
2  const factory = new ethers.ContractFactory(artifact.abi, artifact.
       bytecode, wallet);
3
4  const sources = params.sources.map(s => ({
5      originalId: s.originalId,
6      title: s.title,
7      creatorWallet: s.creatorWallet,
8      reusePercent: BigInt(s.reusePercent),
9      valueShare: ethers.parseEther(s.valueShare.toString())
```

```
10 }));
11
12 const contract = await factory.deploy(
13     params.mixId,
14     params.mixTitle,
15     params.mixCreator,
16     params.mixWallet,
17     params.format || "mp3",
18     params.genre || "unknown",
19     sources
20 );
21 await contract.waitForDeployment();
22 const address = await contract.getAddress();
```

Listagem 4.32: *Deployment* do contrato `MusicMix`.

O contrato armazena, de forma imutável, as referências de todos os trechos utilizados, incluindo os identificadores das obras originais, as percentagens de contribuição e o valor proporcional pago a cada autor. Após o *deployment*, o endereço do contrato e o *transaction hash* são registados no campo `metadata.sc4m. contractAddress` do documento da faixa correspondente no MongoDB.

4.3.2 Execução do `MusicRights`

O contrato `MusicRights` constitui o ponto de partida de todo o ciclo de vida de uma obra musical na plataforma SoundSlice, representando o ato de criação e registo autoral inicial, servindo de base jurídica e técnica para os contratos seguintes. Quando o autor regista uma nova faixa, este tipo de contrato é criado e armazenado na *blockchain*, contendo a informação autoral e os metadados técnicos e financeiros associados ao ficheiro de áudio. O contrato define as propriedades fundamentais da obra:

- `title`, `artist`, `genre`, `format` e `duration` da música original;
- `fileHash` SHA-256 do ficheiro armazenado no MongoDB/GridFS, garantindo a integridade dos dados;
- `basePrice`, que representa o valor base para eventuais reutilizações, em wei;
- vetores `holders` e `shares` de registo de coautores e das respetivas percentagens de propriedade.

O construtor verifica que a soma das percentagens dos coautores é igual a 100, assegurando a correta distribuição dos direitos e evitando incoerências na partilha dos rendimentos. A função pública `registerReuse()` permite registar uma reutilização

da faixa original. O utilizador que pretenda reutilizar um trecho envia uma transação com o valor correspondente à percentagem pretendida segundo a Equação 4.1. O montante é automaticamente distribuído entre os titulares de acordo com as suas quotas (`shares[i]`), garantindo a execução automática do pagamento proporcional. Além disso, todas as reutilizações são registadas localmente no vetor `reuses`, com indicação do utilizador, percentagem reutilizada, valor pago e *timestamp*. As funções `getReuse()` e `getHolders()` permitem consultar os registos e a estrutura de titularidade de forma transparente. Este contrato é implantado diretamente pelo utilizador no *frontend*, através da carteira MetaMask, garantindo a autenticação e a posse legítima da obra.

Embora o contrato `MusicRights` esteja preparado para distribuir automaticamente os valores de reutilização pelos coautores definidos no vetor `holders`, esta funcionalidade ainda não se encontra refletida na interface atual da plataforma. Atualmente, o registo dos titulares é feito de forma centralizada na base de dados MongoDB, sendo apenas o criador principal identificado no contrato implantado. Consequentemente, a totalidade do valor pago em caso de reutilização é transferida para a carteira do criador original, não ocorrendo a divisão proporcional em *on-chain*. Esta opção foi adotada por motivos de simplificação e de coerência com a primeira fase de testes. A funcionalidade de partilha descentralizada encontra-se prevista como melhoria futura, a ser integrada numa versão posterior da plataforma.

4.3.3 Execução do `MusicReuse`

O contrato `MusicReuse` é utilizado sempre que um utilizador seleciona e reutiliza um trecho de uma obra já registada na plataforma. Tem por função documentar e proteger juridicamente a operação de reutilização, associando o trecho à obra original e ao seu titular de direitos. Durante o processo de reutilização, o *backend* recolhe os metadados gerados pela análise de similaridade (percentagem de reutilização, duração do trecho, género, *hashes* de integridade) e executa o *deployment* do contrato. O construtor recebe uma estrutura `ReuseData` com os seguintes campos principais:

- `originalId`, `originalTitle`, `creatorWallet` — identificam a obra original e o respetivo autor;
- `reuserName`, `reuserWallet`, `reusePercent`, `valuePaid` — descrevem o utilizador que reutiliza o conteúdo, a percentagem usada e o montante pago;
- `originalFileHash` e `snippetHash` — asseguram a integridade dos ficheiros e a rastreabilidade entre original e trecho;
- `genre`, `format` e `snippetDuration` — caracterizam tecnicamente o segmento reutilizado.

Uma vez o contrato armazenado na *blockchain*, é emitido o evento `ReuseRegistered`, que contém os dados essenciais da transação. Este evento é posteriormente interpretado pelo *backend* e armazenado no MongoDB, dentro do campo `metadata.sc4m.contractAddress` da reutilização. O `timestamp` e os `hashes` permitem auditar o processo e comprovar, de forma imutável, a correspondência entre a reutilização e a obra original. Desta forma, o contrato `MusicReuse` atua como uma camada de certificação digital entre a música original e o seu trecho reutilizado, formalizando a operação de licenciamento.

4.3.4 Execução do `MusicMix`

O contrato `MusicMix` foi concebido para registar composições derivadas, *mixes*, que combinam múltiplos trechos provenientes de diferentes obras registadas. O seu objetivo é consolidar, num único registo *on-chain*, a autoria, os direitos e as proporções de cada contribuinte da *mix*. O construtor recebe, como parâmetros, os dados da *mix* criada pelo utilizador e um vetor de estruturas `Source`, que representam cada música utilizada. Cada `Source` inclui:

- `originalId`, `title`, `creatorName`, `creatorWallet`;
- `reusePercent` — percentagem do trecho original utilizada no *mix*;
- `valueShare` — valor correspondente em wei, calculado de forma proporcional à reutilização.

Durante a implantação, o contrato copia todos os elementos do vetor `_sources` para a variável `sources`, e emite o evento `MixRegistered`, que contém o identificador da *mix*, o número de faixas de origem e a data de criação.

A função `getSources()` permite consultar a composição de uma *mix*, devolvendo a lista de músicas utilizadas e respetivas percentagens. Assim, a *blockchain* mantém um registo completo, imutável e auditável das relações de derivação entre obras musicais, preservando os direitos de todos os intervenientes.

O contrato `MusicMix` representa, portanto, a camada final do ciclo criativo na plataforma, a materialização descentralizada do conceito de "música colaborativa", na qual múltiplos autores participam na criação de novos conteúdos, sendo automaticamente remunerados em função da sua contribuição.

4.4 Disponibilização do Código-Fonte

De forma a assegurar a reprodutibilidade e transparência do trabalho desenvolvido, todo o código-fonte da plataforma SoundSlice, incluindo o *frontend*, o *backend* e os contratos inteligentes, foi disponibilizado num repositório público no GitHub[2].

[2]Disponível em: `https://github.com/mariana-pinheiro/sound-slice.git`

4.5 Sumário

Este capítulo apresentou a implementação integral da plataforma SoundSlice, desde o *frontend* desenvolvido em HTML, Tailwind CSS e JavaScript, até ao *backend* em Node.js com MongoDB e integração com a *blockchain* Ethereum.

Foram descritos os principais módulos da aplicação, registo e autenticação de utilizadores, *upload* e gestão de músicas, reutilização de trechos e criação de *mixes*, e demonstrada a sua articulação com os contratos inteligentes `MusicRights`, `MusicReuse` e `MusicMix`.

Esta implementação materializa a proposta apresentada no Capítulo 3, constituindo a base para a avaliação experimental descrita no Capítulo 5.

Capítulo 5

Avaliação e Resultados

Este capítulo apresenta o processo de validação experimental da solução desenvolvida, complementando a componente teórica e de implementação descrita nos capítulos anteriores. O objetivo principal é demonstrar o correto funcionamento da plataforma SoundSlice e avaliar o seu desempenho e usabilidade em diferentes contextos de utilização, garantindo que os requisitos definidos no início do projeto foram efetivamente cumpridos.

5.1 Introdução

Inicialmente, são descritos os testes funcionais, que visam confirmar a integridade e coerência das operações implementadas, assegurando que cada módulo do sistema — autenticação, *upload* de músicas, análise de reutilização, geração de contratos e interação *on-chain* — executa as suas funções conforme o esperado. De seguida, são apresentados os testes de desempenho, onde se avalia a latência média, mínima e máxima das principais rotas da API, bem como a estabilidade do sistema após múltiplas iterações. Por fim, é incluída uma análise de usabilidade, que procura compreender a perceção do utilizador final quanto à clareza da interface, facilidade de navegação e adequação da experiência global.

Este capítulo estabelece, assim, a ligação entre o desenvolvimento técnico descrito anteriormente e a validação prática da solução. Os resultados obtidos servirão de base para a discussão crítica e para as conclusões apresentadas no capítulo seguinte,

onde são sintetizados os contributos do trabalho e identificadas possíveis melhorias e extensões futuras.

5.2 Configuração Experimental

Esta secção descreve a configuração do ambiente experimental utilizado para a execução e validação dos testes funcionais e de desempenho da aplicação SoundSlice. O objetivo é garantir a reprodutibilidade dos resultados obtidos e a coerência entre as diferentes camadas do sistema (*frontend*, *backend*, base de dados e *blockchain* local).

As experiências foram realizadas num ambiente local com as seguintes características:

- **Sistema Operativo:** Windows 11 Home 64 bits;
- **Processador:** Intel Core i7, 1,8 GHz;
- **Memória RAM:** 8 GB;
- **Node.js:** versão 22.18.2;
- **Hardhat:** versão 3.0.7 (rede local de testes);

As ferramentas selecionadas para a execução dos diferentes tipos de teste foram:

- **Postman:** caracterização do funcionamento da API do *backend* em termos de funcionalidade e desempenho (latência e volume de dados);
- **Hardhat Console:** monitorização das transações *on-chain* e *gas usage*;
- **MongoDB Compass:** inspeção dos dados e validação de persistência;
- **Chrome Browser Developer Tools:** inspeção de desempenho do *frontend*;
- **Formulário *System Usability Scale*:** avaliação da experiência de utilização (aplicado a um grupo de utilizadores).

Todos os testes foram realizados numa instância local do servidor `Node.js` (porta 3000), comunicando com o MongoDB e a rede Hardhat em execução. Cada caso de uso foi testado de forma isolada e integrada, com registo dos tempos médios de resposta, latência de rede, volume de dados processados e validação dos contratos inteligentes criados. Os testes de usabilidade foram conduzidos após a estabilidade funcional do sistema, com foco na clareza da interface e na facilidade de execução das tarefas principais. Este ambiente experimental serviu de base para as avaliações subsequentes apresentadas nas secções seguintes.

5.3 Métricas de Avaliação

Esta secção apresenta as métricas utilizadas na avaliação da aplicação SoundSlice, abrangendo os testes funcionais, de desempenho e de usabilidade. O objetivo é definir os critérios que permitiram analisar o comportamento do sistema, tanto ao nível técnico como da experiência do utilizador.

Testes Funcionais — As métricas aplicadas centraram-se na verificação da fiabilidade e consistência das operações da API. Foram analisadas a taxa de sucesso das respostas (`200 OK` e `201 Created`), a coerência entre os dados enviados e os registados na base de dados, a integridade dos ficheiros guardados no GridFS e a correspondência entre os contratos inteligentes criados e os dados registados no MongoDB. A correta geração de contratos, com identificação do `contractAddress`, `txHash` e `chainId`, foi também considerada uma métrica essencial para a validação da integração com a *blockchain*.

Testes de Desempenho — As métricas de desempenho visaram medir a capacidade de resposta e a eficiência do sistema em diferentes operações. Foram recolhidos o tempo médio de resposta aos pedidos HTTP, a latência de rede entre cliente e servidor, o tempo de processamento dos *uploads* e o tempo médio de *deployment* dos contratos inteligentes `MusicReuse.sol` e `MusicMix.sol`. Analisou-se igualmente o volume total de dados processados e a utilização média da *Central Processing Unit* (CPU) e memória durante as operações de maior carga. Cada caso de uso foi repetido dez vezes, permitindo calcular valores médios e garantir a consistência dos resultados.

Testes de Usabilidade — Aplicou-se a metodologia *System Usability Scale* (SUS) a um grupo de utilizadores que interagiu com a interface *web* da aplicação. Foram consideradas as pontuações atribuídas no questionário SUS (escala de 0 a 100), o tempo médio de execução das principais tarefas (como registo, *upload* e criação de *mixes*), a taxa de erros durante a navegação e o nível de satisfação declarado pelos participantes. Estas métricas permitiram avaliar a eficácia, eficiência e aceitação global da aplicação.

A Tabela 5.1 lista as métricas adotadas para cada tipo de teste.

5.4 Cenários de Teste

Os cenários de teste foram elaborados com base nos principais casos de uso do sistema, simulando o ciclo completo de utilização da aplicação, desde o registo do utilizador até à criação e validação de contratos inteligentes. Cada cenário foi executado através de pedidos na ferramenta Postman, garantindo a consistência dos resultados e a rastreabilidade entre as operações efetuadas.

Tabela 5.1: Resumo das métricas de avaliação.

Tipo de Teste	**Métrica**	**Objetivo de Avaliação**
Funcional	Taxa de sucesso	Confirmar o correto funcionamento dos *endpoints*
Funcional	Validação de dados	Garantir coerência entre API, MongoDB e *blockchain*
Funcional	Geração de contratos	Verificar criação e integridade dos contratos inteligentes
Desempenho	Tempo médio de resposta	Avaliar a rapidez de processamento do servidor
Desempenho	Tempo de *deployment* do contrato	Medir eficiência da integração com a *blockchain*
Desempenho	Volume de dados processados	Quantificar carga de ficheiros e metadados
Usabilidade	Pontuação SUS	Avaliar perceção global de usabilidade
Usabilidade	Tempo médio por tarefa	Medir eficiência das interações do utilizador
Usabilidade	Taxa de erros	Identificar dificuldades na interface e navegação

Cenário 1 — Registo e Autenticação consiste na criação de uma conta de utilizador, autenticação e consulta do perfil. Os testes envolveram os seguintes *endpoints*:

- `POST /api/users/register` — registo de um novo utilizador com dados pessoais, país, cidade e *wallet* Ethereum;
- `POST /api/users/login` — autenticação do utilizador e geração de *token* JWT;
- `GET /api/users/me` — consulta do perfil autenticado e verificação de dados armazenados.
- `POST /api/users/logout` — término da sessão e invalidação do *token* de autenticação.

Os testes confirmaram a criação correta do utilizador na base de dados MongoDB, a geração do *token* de sessão e o acesso autorizado às rotas protegidas.

Cenário 2 — *Upload* e Armazenamento de Músicas avalia o processo de envio e armazenamento de faixas musicais, verificando a integração entre o servidor Node.js, o GridFS e a base de dados. Foram testados os seguintes *endpoints*:

- `POST /api/tracks/metadata` — extração automática dos metadados de uma faixa (formato, duração, tamanho e `hash`);
- `POST /api/tracks/upload` — envio da música e da capa para o servidor, com validação da estrutura JSON e do campo `basePrice`;
- `GET /api/tracks/mine` — listagem das músicas do utilizador autenticado;
- `GET /api/tracks/:id` — consulta detalhada de uma música armazenada;
- `PUT /api/tracks/:id` — edição das informações de uma música existente;

- `DELETE /api/tracks/:id` — eliminação de uma música;
- `GET /api/tracks/:id/file` — *streaming* autenticado do ficheiro de áudio;
- `GET /api/tracks/:id/file/public` — *streaming* público do ficheiro de áudio;
- `GET /api/tracks/:id/cover` — obtenção da capa associada à música;
- `GET /api/tracks/:id/contract-pdf` — geração automática do contrato em formato PDF;
- `GET /api/tracks/feed` — *feed* público de músicas disponíveis na plataforma.

Os dados recolhidos incluíram o tempo médio de resposta e o tamanho dos ficheiros armazenados. Foi ainda confirmada a associação dos ficheiros de áudio e imagem a identificadores `ObjectId` no GridFS.

Cenário 3 — Análise e Reutilização de Faixas Musicais simula a reutilização parcial de uma faixa por outro utilizador. O objetivo foi validar o cálculo automático da percentagem de reutilização e o valor de compensação. Os *endpoints* testados foram:

- `POST /api/tracks/:id/reuse/select` — seleção do trecho (início e fim) e geração de um *snippet*;
- `POST /api/tracks/:id/reuse/confirm` — confirmação da reutilização e criação automática do contrato inteligente.

Durante este teste foram registados o tempo de processamento, o tamanho do excerto gerado e o tempo de *deployment* do contrato `MusicReuse.sol` na *blockchain* local Hardhat.

Cenário 4 — Geração e Validação do Contrato *On-Chain* contempla os seguintes casos:

- Criação automática do contrato `MusicRights` aquando do *upload* de uma nova música (`POST /api/tracks/upload`);
- Criação automática do contrato `MusicReuse` aquando da reutilização ou criação de um mix;
- *Download* do contrato PDF da música original (`GET /api/tracks/:id/contract-pdf`);
- *Download* do contrato PDF da reutilização (`GET /api/tracks/:id/reuse/contract-pdf`);

- Validação dos dados *on-chain* através da biblioteca `ethers.js`.

Foram recolhidos os endereços dos contratos, identificadores das transações e tempos médios de execução das operações *on-chain*.

Cenário 5 — Criação de *Mix* e Contrato `MusicMix` avalia a criação de uma *mix* a partir de vários excertos já licenciados, resultando num novo contrato inteligente `MusicMix.sol`. O *endpoint* principal testado foi:

- `POST /api/mix/export` — envio do ficheiro combinado, com metadados e `snippets` reutilizados.

Foi registado o tempo médio de processamento e o endereço do contrato `MusicMix` gerado automaticamente. Confirmou-se ainda a persistência dos metadados SC4M e IMAF no registo da faixa.

5.4.1 Testes de Desempenho

Os testes de desempenho foram executados sobre os cenários anteriores, medindo a latência e o tempo de resposta médio dos *endpoints* mais críticos. Cada operação foi repetida dez vezes consecutivas, com recolha automática dos tempos no Postman.

5.4.2 Testes de Usabilidade

Os testes de usabilidade foram conduzidos com um grupo de cinco utilizadores, através do método SUS. Cada participante realizou um conjunto de tarefas representativas: registo, autenticação, *upload* de uma música, análise de reutilização, criação de *mixes* e geração de contratos. Após a conclusão, responderam ao questionário SUS composto por dez afirmações avaliadas numa escala de 1 (discordo totalmente) a 5 (concordo totalmente).

5.5 Resultados e Análise

Esta secção apresenta e analisa os resultados obtidos nos testes funcionais, de desempenho e de usabilidade realizados, com o objetivo de validar a conformidade, eficiência e adequação da solução desenvolvida.

5.5.1 Testes Funcionais

Os testes funcionais têm como objetivo avaliar o correto funcionamento das API implementadas no sistema e assegurar a coerência entre a interface *web*, o *backend* desenvolvido em Node.js/Express, a base de dados MongoDB com GridFS e a *blockchain* local Hardhat, onde são registados os contratos inteligentes correspondentes às operações de *upload*, reutilização e mistura musical.

Os testes foram realizados na plataforma Postman, permitindo a obtenção dos resultados por módulo (utilizadores, músicas e *mixes*), bem como a rastreabilidade de cada *endpoint* e o controlo rigoroso dos parâmetros de entrada e de resposta. Todas as chamadas foram realizadas com sucesso, retornando os resultados esperados. A Tabela 5.2 apresenta os casos de uso testados e os *endpoints* correspondentes.

Tabela 5.2: Resultados dos Testes Funcionais.

Caso de Uso	***Endpoint***	**Método**	**Resultado**
Registo de Utilizador	`/api/users/register`	POST	Sucesso
Autenticação	`/api/users/login`	POST	Sucesso
Logout	`/api/users/logout`	POST	Sucesso
Obter Perfil do Utilizador	`/api/users/me`	GET	Sucesso
Atualizar Perfil do Utilizador	`/api/users/update`	PUT	Sucesso
Consultar Reutilizações do Utilizador	`/api/users/reuses`	GET	Sucesso
Consultar Pagamentos do Utilizador	`/api/users/payments`	GET	Sucesso
Extração de Metadados da Música	`/api/tracks/metadata`	POST	Sucesso
Upload de Música (Ficheiro + Capa)	`/api/tracks/upload`	POST	Sucesso
Apagar Música	`/api/tracks/:id`	DELETE	Sucesso
Editar Música	`/api/tracks/:id`	PUT	Sucesso
Listagem das Minhas Músicas	`/api/tracks/mine`	GET	Sucesso
Consulta Detalhada de uma Música	`/api/tracks/:id`	GET	Sucesso
Streaming de Ficheiro Autenticado	`/api/tracks/:id/file`	GET	Sucesso
Streaming Público de Ficheiro	`/api/tracks/:id/file/public`	GET	Sucesso
Obtenção da Capa	`/api/tracks/:id/cover`	GET	Sucesso
Geração do Contrato PDF da Música	`/api/tracks/:id/contract-pdf`	GET	Sucesso
Feed Público de Músicas	`/api/tracks/feed`	GET	Sucesso
Seleção de Excerto de Reutilização	`/api/tracks/:id/reuse/select`	POST	Sucesso
Confirmação e *Deployment* do Contrato de Reuso	`/api/tracks/:id/reuse/confirm`	POST	Sucesso
Geração do Contrato PDF de Reutilização	`/api/tracks/:id/reuse/contract-pdf`	GET	Sucesso
Exportação de *Mix* e *Deployment* de Contrato	`/api/mix/export`	POST	Sucesso
Geração do Contrato PDF da *Mix*	`/api/mix/:id/contract-pdf`	GET	Sucesso

Os testes funcionais confirmaram o correto funcionamento de todas as componentes da aplicação SoundSlice, incluindo a integração entre o *backend*, a base de dados, o sistema de armazenamento distribuído e a *blockchain* local.

Todos os *endpoints* responderam com status de sucesso (`200-201`) e os contratos inteligentes foram adicionados e consultados com sucesso na rede local Hardhat, demonstrando a automação completa do ciclo de gestão de direitos musicais: (*i*) *Upload*; (*ii*) Análise; (*iii*) Reutilização; (*iv*) Geração de Contrato; (*v*) Criação de *Mixes*; e (*vi*) Validação *on-chain*.

É importante salientar que os testes de desempenho e de latência foram realizados num ambiente de simulação local, utilizando a rede Hardhat. Deste modo, os tempos registados representam um cenário idealizado, uma vez que não incluem fatores inerentes a uma rede pública, como a propagação de blocos, o congestionamento de transações ou a variabilidade na comunicação entre nós. Assim, as latências medidas subestimam os valores reais que seriam expectáveis numa *realnet*.

Este conjunto de testes valida que a solução proposta cumpre os requisitos funcionais definidos, garantindo a integridade, rastreabilidade e automatização do processo de gestão de direitos autorais em ambiente descentralizado.

5.5.2 Testes de Desempenho

Os testes de desempenho tiveram como objetivo avaliar a eficiência e a estabilidade da aplicação SoundSlice durante a execução das principais operações do sistema. Esta fase experimental permitiu medir o tempo de resposta médio dos diferentes *endpoints* da API e o volume de dados transferido em operações críticas, como o envio e leitura de ficheiros multimédia, a geração de contratos e as interações *on-chain* com os contratos inteligentes.

Cada caso de uso foi testado individualmente, replicando cenários reais de utilização e executando dez iterações por *endpoint*. Os testes foram realizados em ambiente local, com a *blockchain* simulada através do Hardhat e o armazenamento de dados efetuado em MongoDB com GridFS. Para cada iteração, foram recolhidos o tempo mínimo, máximo e médio de resposta, bem como o volume de dados processado.

Os resultados apresentados nas tabelas seguintes encontram-se organizados por cenário funcional, Registo e Autenticação de Utilizador, *Upload* e Armazenamento de Músicas, Análise de *Snippets*, Reutilização de Músicas e Criação de *Mixes*, permitindo uma visão comparativa do desempenho global da solução.

API do *Backend*

Os resultados da latência e volume de dados apresentados foram obtidos através da ferramenta Postman.

A Tabela 5.3 contém os valores obtidos através da uma execução um único pedido por *endpoint*. Para cada pedido, foi medido o tempo total de resposta (latência) e o volume de dados transferido em bytes (B). Esta medição teve como objetivo obter uma primeira visão do desempenho das principais operações do sistema.

Tabela 5.3: Resultados da latência e volume de dados da API (1 pedido por *endpoint*).

Cenário	***Endpoint***	**Latência** (ms)	**Dados** (B)
Registo e Autenticação	`POST /api/users/register`	886	$3{,}01 \times 10^2$
	`POST /api/users/login`	364	$5{,}75 \times 10^2$
	`GET /api/users/me`	37	$5{,}29 \times 10^2$
	`PUT /api/users/update`	130	$5{,}45 \times 10^2$
	`GET /api/users/payment`	120	$3{,}99 \times 10^2$
	`GET /api/users/reuses`	57	$3{,}80 \times 10^2$
	`POST /api/users/logout`	70	$3{,}71 \times 10^2$
Upload e Armazenamento	`POST /api/tracks/upload`	323	$7{,}22 \times 10^2$
	`POST /api/tracks/metadata`	2760	$4{,}92 \times 10^2$
	`GET /api/tracks/mine`	40	$5{,}32 \times 10^3$
	`PUT /api/tracks/:id`	26	$7{,}37 \times 10^2$
	`GET /api/tracks/:id`	110	$8{,}50 \times 10^2$
	`GET /api/tracks/feed?genre=pop&sort=popular`	20	$2{,}49 \times 10^2$
	`GET /api/tracks/:id/file`	71	$3{,}56 \times 10^6$
	`GET /api/tracks/:id/cover`	35	$110{,}11 \times 10^3$

	`GET /api/tracks/:id/contract-pdf`	291	$100{,}68 \times 10^3$
Análise e Snippet	`GET /api/tracks/snippet/:id`	29	$313{,}57 \times 10^3$
Reutilização de Música	`GET /api/tracks/:id/reuse/select`	3560	$5{,}64 \times 10^2$
	`POST /api/tracks/:id/reuse/confirm`	430	$1{,}45 \times 10^3$
	`GET /api/tracks/:id/reuse/contract-pdf`	475	$102{,}83 \times 10^3$
Criação e Exportação de *Mixes*	`GET /api/mix/export`	779	$3{,}02 \times 10^2$
	`GET /api/mix/:id/contract-pdf`	327	$102{,}46 \times 10^3$

De forma geral, verificou-se que:

- Os tempos médios de resposta situaram-se maioritariamente abaixo de 500 ms, com exceção dos *endpoints* de extração de metadados e de seleção de reutilização, que implicam maior processamento;
- O volume de dados transferido variou conforme o tipo de conteúdo, desde respostas de algumas centenas de B até ficheiros multimédia de vários MB;
- Os *endpoints* relacionados com *streaming*, *upload* e geração de contratos PDF apresentaram desempenho estável, sem falhas ou erros de servidor;
- As operações *on-chain* mostraram tempos médios compatíveis com o ambiente local *Hardhat*, validando a integração eficiente entre o *backend* e a *blockchain*.

A Tabela 5.4 apresenta os resultados da latência obtidos através da execução de dez pedidos por operação. Durante estes testes, determinou-se a latência média e o desvio padrão ($\mu \pm \sigma$), o valor mínimo, máximo e o coeficiente de variação (CV), permitindo aferir a variabilidade do desempenho em condições repetidas de execução. Os resultados encontram-se na Tabela 5.4.

De forma geral, os resultados confirmaram uma latência reduzida e estável na maioria dos *endpoints* da API, com tempos médios de resposta abaixo dos 200 ms para as operações mais frequentes. As rotas de autenticação (`/users/register` e `/users/login`) apresentaram tempos médios entre 160 ms e 170 ms, refletindo o processamento adicional associado à validação de credenciais e criação de *tokens*. As operações de consulta, como `/users/me` e `/tracks/mine`, mantiveram latências muito baixas, geralmente inferiores a 30 ms, evidenciando a eficiência das leituras diretas sobre a base de dados MongoDB.

Os *endpoints* relacionados com o envio e manipulação de ficheiros (`/tracks/upload` e `/tracks/metadata`) apresentaram tempos médios superiores, entre 90 ms e 130 ms, devido à necessidade de escrita no GridFS e à análise de metadados dos ficheiros de áudio. A geração de contratos em formato PDF situou-se em torno dos 200 ms, valor consistente com a criação dinâmica de documentos a partir dos dados armazenados.

As operações com maior tempo de execução corresponderam aos cenários que envolvem processamento intensivo e interação com os contratos inteligentes. A rota

Tabela 5.4: Resultados consolidados da latência da API (10 pedidos por *endpoint*).

Endpoint	**Latência** (ms)			
	$\mu \pm \sigma$	**Mín.**	**Máx**	**CV**
`POST /api/users/register`	167,8 ± 18,7	119	308	0,1
`POST /api/users/login`	161,0 ± 14,5	118	254	0,2
`GET /api/users/me`	8,1 ± 1,2	6	10	0,2
`PUT /api/users/update`	15,2 ± 2,5	11	26	0,2
`GET /api/users/payment`	21,1 ± 4,8	13	50	0,2
`GET /api/users/reuses`	26,2 ± 9,6	14	100	0,4
`POST /api/tracks/metadata`	94,7 ± 11,9	64	129	0,1
`POST /api/tracks/upload`	128,0 ± 15,3	92	183	0,1
`GET /api/tracks/mine`	20,7 ± 2,7	15	29	0,1
`GET /api/tracks/:id`	12,0 ± 2,1	8	29	0,2
`GET /api/tracks/:id/file`	35,2 ± 4,3	28	60	0,1
`GET /api/tracks/:id/file/public`	45,4 ± 6,9	32	83	0,2
`GET /api/tracks/:id/cover`	30,7 ± 10,4	6	120	0,3
`GET /api/tracks/:id/contract-pdf`	200,0 ± 22,8	104	365	0,1
`GET /api/tracks/:id/reuse/select`	2521,0 ± 412,6	1941	4008	0,2
`POST /api/tracks/:id/reuse/confirm`	264,0 ± 25,2	229	340	0,1
`GET /api/tracks/:id/reuse/contract-pdf`	206,0 ± 18,9	155	303	0,1
`GET /api/mix/extract-mix`	347,0 ± 31,4	296	465	0,1
`GET /api/mix/:id/contract-pdf`	124,0 ± 14,1	84	209	0,1

`/tracks/:id/reuse/select` apresentou o maior tempo médio registado (cerca de 2,5 s), resultado do cálculo de percentagens de reutilização e da comunicação com a *blockchain*. Ainda assim, as restantes interações *on-chain*, nomeadamente a confirmação de reutilização e a exportação de *mixes*, mantiveram-se dentro de valores aceitáveis para um ambiente de testes local, com médias entre 260 ms e 350 ms.

Contratos Inteligentes

Os testes foram conduzidos em ambiente local, utilizando a rede simulada Hardhat e o perfilador interno do Hardhat para medir o tempo de execução e o consumo de *gas*. A avaliação foi baseada nas métricas propostas por Javed and Mangues-Bafalluy [53], nomeadamente:

- **Latência da transação** — tempo entre o envio (t_{submit}) e a mineração (t_{mined}) de uma transação;
- ***Overhead* da troca de dados** — número e tamanho das comunicações com a rede *blockchain*;
- **Tempo de execução** — tempo necessário à execução interna do contrato na Ethereum *Virtual Machine* (EVM);
- **Taxa de erro** — proporção de pedidos falhados por tentativas totais;
- **Frequência de pedidos** — número de chamadas externas por execução de contrato.

Foram realizadas dez pedidos consecutivas do mesmo cenário de criação de contratos, sendo recolhidos os tempos de *deployment*, execução de função e latência de transação. A latência foi medida a partir dos *logs* do *backend* (através do método `measureTransaction`), correspondendo ao intervalo entre o envio e a confirmação da transação ($t_{mined} - t_{submit}$), conforme proposto por Javed and Mangues-Bafalluy [53]. O tempo de execução e o consumo de *gas* foram obtidos diretamente através do Hardhat Profiler.

Contrato `MusicRights`

Os resultados consolidados são apresentados na Tabela 5.5. Os valores temporais correspondem ao valor médio (μ) e desvio padrão (σ) das observações.

Tabela 5.5: Métricas de desempenho da execução dos contratos inteligentes `MusicRights`.

Métrica	$\mu \pm \sigma$	**Unidade**
Latência da transação	$0{,}013 \pm 0{,}008$	s
Tempo de *deployment* interno	$0{,}007 \pm 0{,}004$	s
Tempo de execução da função	$0{,}002 \pm 0{,}002$	s
Gas usado	1 008 812	gas
Taxa de erro	0	%
Frequência de pedidos	1	chamada/execução

O contrato `MusicRights` apresentou um desempenho estável e altamente eficiente. A latência média de apenas 13 ms evidencia a rapidez com que as transações são processadas e confirmadas na rede local. O tempo médio de implantação de 7 ms confirma a leveza do contrato e a eficiência da sua estrutura, enquanto o tempo de execução das funções de leitura foi praticamente instantâneo, situando-se em torno de 2 ms.

O consumo de *gas* manteve-se constante em todas as execuções, com uma média de aproximadamente 1 008 812 unidades, refletindo a consistência da lógica interna e a ausência de dependências externas que pudessem introduzir variabilidade. A taxa de erro nula demonstra a fiabilidade do processo de implantação, enquanto a frequência unitária de pedidos confirma que o contrato requer apenas uma transação por ciclo completo de criação.

Contrato `MusicReuse`

Os resultados consolidados das dez execuções encontram-se na Tabela 5.6. Os valores temporais apresentam o valor médio (μ) e o desvio padrão (σ) das observações.

Os resultados demonstram um comportamento estável e eficiente do contrato `MusicReuse`. A latência média de 273 ms reflete o tempo total necessário para que uma transação seja processada e incluída num bloco, valor adequado para um ambiente de testes local e compatível com medições reportadas em estudos semelhantes [53].

Tabela 5.6: Métricas de desempenho da execução dos contratos inteligentes `MusicReuse`.

Métrica	$\mu \pm \sigma$	**Unidade**
Latência da transação	0.273 ± 0.061	s
Tempo de *deployment* interno	0.052 ± 0.009	s
Tempo de execução da função	0.006 ± 0.001	s
Gas usado	92 638	gas
Taxa de erro	0	%
Frequência de pedidos	1	chamada/execução

O tempo de *deployment* situou-se em torno de 52 ms, enquanto o tempo de execução de funções de leitura foi praticamente instantâneo (6 ms).

O consumo de *gas* manteve-se constante nas dez execuções (926 388 unidades), indicando consistência na complexidade computacional do contrato. A taxa de erro foi nula, confirmando a estabilidade e fiabilidade do processo de implantação e interação com o contrato inteligente.

Contrato `MusicMix`

Após a realização das dez iterações, obtiveram-se os resultados consolidados apresentados na Tabela 5.7. Os valores temporais representam a média (μ) e o desvio padrão (σ) das observações.

Tabela 5.7: Métricas de desempenho da execução dos contratos inteligentes `MusicMix`.

Métrica	$\mu \pm \sigma$	**Unidade**
Latência da transação	0.021 ± 0.007	s
Tempo de *deployment* interno	0.032 ± 0.004	s
Tempo de execução da função	0.007 ± 0.001	s
Gas usado	1 022 143	gas
Taxa de erro	0	%
Frequência de pedidos	1	chamada/execução

Os resultados confirmam a eficiência do contrato inteligente `MusicMix` e da integração com a *blockchain* local:

- A latência média da transação, de aproximadamente 21 ms, evidencia o baixo tempo de propagação e mineração num ambiente controlado Hardhat;
- O consumo de *gas* manteve-se estável ($\approx 1 \times 10^6$ unidades), refletindo a previsibilidade do custo computacional da execução;
- O tempo de execução interno (32 ms) confirma uma execução determinística e rápida do contrato;
- Não foram registadas falhas (taxa de erro nula) nas dez execuções, garantindo fiabilidade do processo;

- Cada iteração implicou uma única chamada RPC (`factory.deploy()`), representando frequência unitária e *overhead* mínimo.

Como não foram utilizados oráculos nem APIs externas, o *overhead* de comunicação é exclusivamente o tráfego gerado pelas chamadas RPC ao nó local, estimado em menos de 100 kB por transação. Esta eficiência demonstra que a camada *on-chain* não representa um gargalo de desempenho na aplicação SoundSlice, validando a integração entre o *backend* e a EVM como solução viável para a gestão automatizada de direitos musicais.

Apesar dos resultados obtidos demonstrarem tempos de execução e latências bastante reduzidos, importa referir que estes foram recolhidos num ambiente controlado de simulação. Consequentemente, as latências observadas minoram as que ocorreriam numa rede pública real (*realnet*), onde a propagação e a confirmação das transações dependem de múltiplos fatores externos.

5.5.3 Testes de Usabilidade

O teste de usabilidade foi realizado utilizando o método SUS para avaliar a perceção dos utilizadores em relação à usabilidade do sistema. Cinco participantes responderam ao questionário composto por 10 questões (Q_1, ..., Q_{10}), avaliadas através de uma escala de Likert. Os resultados foram processados seguindo o método estabelecido, resultando nas classificações SUS ($\text{Score}_{\text{SUS}}$) apresentadas na Tabela 5.8.

Tabela 5.8: Resultados do teste SUS.

Participante	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	$\text{Score}_{\text{SUS}}$
1	5	1	5	1	5	1	5	1	5	1	100,0
2	4	1	4	2	5	1	5	1	4	2	87,5
3	5	5	2	4	3	4	3	4	2	2	40,0
4	5	1	4	2	5	1	4	2	4	1	87,5
5	5	2	4	2	5	2	4	2	5	1	85,0

Estas classificações, quando interpretadas segundo o método, indicam:

- Participante 1 – excelente usabilidade (100.0).
- Participante 2 – excelente usabilidade (87.5).
- Participante 3 – usabilidade fraca (40.0).
- Participante 4 – excelente usabilidade (87.5).
- Participante 5 – excelente usabilidade (85.0).

Para determinar a usabilidade global do sistema foi calculada a média das classificações SUS ($\overline{\text{Score}_{\text{SUS}}}$) através da Equação 5.1.

$$\overline{\text{Score}_{\text{SUS}}} = \frac{100.0 + 87.5 + 40 + 87.5 + 85}{5} = 80.0 \tag{5.1}$$

De acordo com a escala SUS, este valor médio indica que o sistema apresenta uma usabilidade **boa a excelente**.

5.6 Sumário

Este capítulo apresentou a avaliação experimental da plataforma SoundSlice, abrangendo os testes funcionais, de desempenho e de usabilidade. Os testes funcionais confirmaram o correto funcionamento das componentes da aplicação e a coerência entre o *frontend*, o *backend*, a base de dados e a *blockchain* local. Através dos testes de desempenho, verificou-se que a aplicação apresenta tempos de resposta reduzidos e estáveis, com latências médias inferiores a 500 ms na maioria das operações, e uma integração eficiente entre o servidor, o sistema de armazenamento distribuído e a camada *on-chain*. A análise de desempenho dos contratos inteligentes demonstrou um comportamento previsível e de baixo consumo de *gas*, confirmando a viabilidade técnica da automação de processos de licenciamento e remuneração através da EVM.

Por fim, os testes de usabilidade, realizados com base no método SUS, revelaram uma perceção global positiva dos utilizadores, classificando a interface como **boa a excelente** em termos de clareza, facilidade de navegação e adequação às tarefas propostas.

Em síntese, os resultados obtidos validam o cumprimento dos objetivos funcionais e técnicos definidos para a plataforma, comprovando a eficácia da solução desenvolvida na gestão automatizada de direitos musicais.

Capítulo 6

Conclusão

O presente capítulo apresenta as conclusões finais da dissertação, sintetizando os principais resultados obtidos ao longo do desenvolvimento da plataforma SoundSlice. Procura-se refletir sobre o contributo do trabalho realizado para a área da gestão automatizada de direitos musicais, evidenciando as inovações introduzidas e os objetivos alcançados. São ainda discutidas as principais limitações identificadas durante o processo de investigação e implementação, bem como as recomendações e perspetivas de evolução futura que poderão sustentar o aperfeiçoamento e a expansão da solução proposta.

6.1 Contribuições

O trabalho desenvolvido permitiu conceber, implementar e validar uma solução inovadora para a gestão automatizada de direitos musicais, baseada na integração de contratos inteligentes e tecnologias de análise de conteúdo audiovisual. A plataforma SoundSlice materializa o conceito de fragmentação e reutilização musical, possibilitando o registo, a partilha e a criação de novos conteúdos — incluindo *mixes* — de forma transparente e auditável.

Entre as principais contribuições destaca-se a definição de uma arquitetura funcional composta por uma camada de armazenamento centralizado, suportada por MongoDB e GridFS, uma camada de processamento e gestão de lógica de negócio desenvolvida em Node.js e Express, e uma camada descentralizada de execução contratual baseada em *blockchain*. Foi também implementado um conjunto de módulos

e interfaces que suportam o ciclo completo de gestão musical, abrangendo as etapas de *upload*, análise de reutilização, geração e validação de contratos inteligentes.

Adicionalmente, foram adotadas as especificações *Smart Contracts for Media* (SC4M) e *Interactive Music Application Format* (IMAF), assegurando a conformidade com padrões internacionais de descrição e gestão de obras multimédia. Por fim, a solução foi submetida a um processo de validação experimental que incluiu testes funcionais, de desempenho e de usabilidade, comprovando a consistência do modelo e a sua aplicabilidade prática em cenários de reutilização musical.

A dissertação contribui, assim, para o avanço do estudo da aplicação de tecnologias de *blockchain* e contratos inteligentes na indústria criativa, demonstrando o seu potencial para promover uma gestão mais justa, automática e transparente dos direitos de autor.

6.2 Reflexões Críticas

O processo de conceção e desenvolvimento da plataforma SoundSlice revelou-se desafiante e enriquecedor, permitindo a consolidação de competências técnicas e científicas em áreas interdisciplinares, como a engenharia de software, a análise de áudio e as tecnologias de registo distribuído.

Durante a fase de implementação, tornou-se evidente a importância de equilibrar a componente descentralizada da solução com mecanismos centralizados que garantissem eficiência e escalabilidade. A opção por um modelo híbrido, combinando uma base de dados centralizada com contratos inteligentes executados em rede local, mostrou-se adequada para a fase de prototipagem, embora implique desafios adicionais na futura transição para ambientes públicos.

A integração dos conceitos de SC4M e IMAF constituiu igualmente uma mais-valia conceptual e reforçou a compreensão da complexidade associada à interoperabilidade de padrões multimédia e à necessidade de normalização no domínio da gestão digital de direitos.

Por outro lado, a realização dos testes de usabilidade proporcionou uma visão empírica sobre a perceção dos utilizadores, destacando a importância da simplicidade de interação, da clareza informativa e da confiança nas operações automatizadas.

6.3 Limitações

Apesar dos resultados obtidos, o trabalho desenvolvido apresenta algumas limitações que importa reconhecer. Em primeiro lugar, o sistema foi testado num ambiente controlado e em rede local, o que leva a que estes valores sejam minorados relativamente às latências reais expectáveis em redes públicas, onde intervêm fatores como a propagação de blocos, a competição de transações e a variabilidade da carga na

rede. Esta limitação será endereçada em trabalhos futuros, através da realização de testes em redes de teste públicas (*testnets*) ou mesmo na *mainnet* Ethereum, de forma a obter medições mais representativas do comportamento real.

Adicionalmente, a integração dos pacotes SC4M e IMAF foi explorada sobretudo ao nível teórico e estrutural, não tendo sido possível implementar todas as suas funcionalidades devido a limitações de documentação e compatibilidade técnica.

Por fim, o sistema de contratos inteligentes foi desenvolvido num contexto de prototipagem e em rede simulada, o que, embora adequado para validação conceptual, não reflete integralmente as condições de operação em redes públicas com custos de transação e restrições de escalabilidade.

6.4 Trabalho Futuro e Recomendações

Em primeiro lugar, seria pertinente explorar a migração da infraestrutura *on-chain* para uma rede pública de *blockchain*, permitindo validar a interoperabilidade e o desempenho do sistema em ambientes descentralizados e de maior escala. Essa transição possibilitaria também o estudo de estratégias de otimização de *gas*, mecanismos de autenticação descentralizada e integração com carteiras digitais reais.

Outra vertente relevante seria o aprofundamento da integração do pacote *Smart Contracts for Media* (SC4M), através da incorporação dos seus modelos contratuais completos e da extensão das funcionalidades de licenciamento. Adicionalmente, a plataforma poderá beneficiar da criação de um sistema de reputação e validação comunitária, onde utilizadores e titulares de direitos possam confirmar manualmente as correspondências sugeridas pela análise automática. Uma melhoria bastante importante é, também, a atualização da interface para permitir a distribuição de *royalties* pelos vários titulares, indo de encontro ao contrato já criado. Por fim, recomenda-se a realização de testes de usabilidade e desempenho em contexto real, com músicos, produtores e entidades de gestão coletiva, de modo a recolher contributos empíricos que validem a aplicabilidade prática e comercial da solução.

Referências

[1] Equipa TOTVS, “Smart contracts: agilidade e segurança com tecnologia de ponta,” Available at https://www.totvs.com/blog/gestao-juridica/smart-contracts/, 2024, (Last accessed in 26/03/2025). [Citado na página 7]

[2] Iberdrola, “Smart contracts,” Available at https://www.iberdrola.com/inovacao/smart-contracts/, (Last accessed in 26/03/2025). [Citado na página 7]

[3] N. Szabo, “Formalizing and securing relationships on public networks,” *First Monday*, vol. 2, no. 9, 1997, consultado em abril de 2025. [Online]. Available: https://firstmonday.org/ojs/index.php/fm/article/view/548 [Citado na página 7]

[4] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008, consultado em abril de 2025. [Online]. Available: https://bitcoin.org/bitcoin.pdf [Citado na página 7]

[5] V. Buterin, “Ethereum white paper: A next-generation smart contract and decentralized application platform,” 2013, consultado em abril de 2025. [Online]. Available: https://ethereum.org/en/whitepaper/ [Citado na página 7]

[6] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, 2014, consultado em abril de 2025. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf [Citado na página 8]

[7] J. Schmidt, G. Wagner, and J. vom Brocke, “Smart contracts in financial services: A systematic literature review,” *Journal of Information Technology*, 2022. [Citado na página 8]

[8] T. Min, H. Wang, Y. Guo, and W. Cai, “Blockchain games: A survey,” *arXiv preprint arXiv:1906.05558*, 2019. [Online]. Available: https://arxiv.org/abs/1906.05558 [Citado na página 8]

[9] G. Zyskind, O. Nathan, and A. S. Pentland, “Decentralizing privacy: Using blockchain to protect personal data,” in *2015 IEEE Security and Privacy Workshops*, May 2015, pp. 180–184. [Citado na página 9]

[10] X. Xu, X. Chen *et al.*, "The application of blockchain in logistics and supply chain management," *International Journal of Production Research*, vol. 57, no. 7, pp. 2117–2135, 2019. [Citado na página 9]

[11] M. M. d. S. Florbela Galvão Cunha, "A systematic literature review on blockchain for real estate transactions: Benefits, challenges, enablers, and inhibitors," 2023. [Citado na página 9]

[12] Taş, Ruhi and Tanrıöver, Ömer Özgür, "A systematic review of challenges and opportunities of blockchain for e-voting," *Symmetry*, vol. 12, no. 8, 2020. [Online]. Available: https://www.mdpi.com/2073-8994/12/8/1328 [Citado na página 9]

[13] ISO/IEC JTC1/SC29/WG03 (MPEG), "Reference software for ISO/IEC 21000-23 smart contracts for media," https://standards.iso.org/iso-iec/21000/-23/ed-1/en/, 2018, acedido em abril de 2025. [Citado na página 9]

[14] ISO/IEC JTC1/SC 29/WG MPEG, "ISO/IEC 23000-12: Interactive music application format (MPEG-A part 12)," https://www.iso.org/standard/53644.html, 2010, acedido em abril de 2025. [Citado na página 9]

[15] M. O'Dair, Z. Beaven, J. Neilson, and P. Pacifico, "Music on the blockchain," 2019. [Online]. Available: https://doi.org/10.1002/jsc.2240 [Citado na página 10]

[16] MPEG, "About MPEG," https://www.mpeg.org/about-mpeg, acedido em abril de 2025. [Citado na página 10]

[17] J. Corral García, P. Kudumakis, I. Barbancho, L. J. Tardón, and M. Sandler, *Enabling Interactive and Interoperable Semantic Music Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 911–921. [Online]. Available: https://doi.org/10.1007/978-3-662-55004-5_45 [Citado na página 10]

[18] I. Jang, P. Kudumakis, M. Sandler, and K. Kang, "The MPEG interactive music application format standard [standards in a nutshell]," *IEEE Signal Processing Magazine*, vol. 28, no. 1, pp. 150–154, Jan 2011. [Citado na página 10]

[19] Sound Software / Queen Mary University of London, "IMAF encoder reference software," https://code.soundsoftware.ac.uk/projects/sv-imaf-enc/files, 2010, acedido em abril de 2025. [Citado na página 12]

[20] I. Heap, "Tiny human and the future of music licensing on Ethereum," *Journal of Music Technology*, vol. 9, no. 2, pp. 55–62, 2016. [Citado na página 13]

[21] I. De Leon and R. Gupta, "The impact of digital innovation and blockchain on the music industry," Inter-American

Development Bank, Tech. Rep. IDB-DP-549, 2017. [Online]. Available: https://publications.iadb.org/publications/english/document/The-Impact-of-Digital-Innovation-and-Blockchain-on-the-Music-Industry.pdf [Citado na página 13]

[22] B. Bodó, D. Gervais, and J. P. Quintais, "Blockchain and smart contracts: the missing link in copyright licensing?" *International Journal of Law and Information Technology*, vol. 26, no. 4, pp. 311–336, 2018. [Online]. Available: https://academic.oup.com/ijlit/article/26/4/311/5106727 [Citado na página 14]

[23] N. Kapsoulis, A. Psychas, G. Palaiokrassas, A. Marinakis, A. Litke, T. Varvarigou, C. Bouchlis, A. Raouzaiou, G. Calvo, and J. E. Subirana, "Consortium blockchain smart contracts for musical rights governance in a collective management organizations (CMOs) use case," *Future Internet*, vol. 12, no. 8, p. 134, 2020. [Online]. Available: https://www.mdpi.com/1999-5903/12/8/134 [Citado na página 14]

[24] M. N. Halgamuge and D. Guruge, "Fair rewarding mechanism in music industry using smart contracts on public-permissionless blockchain," *Multimedia Tools and Applications*, vol. 81, pp. 1523–1544, 2022. [Online]. Available: https://link.springer.com/article/10.1007/s11042-021-11078-6 [Citado na página 14]

[25] A. J. Sharp and O. Lobel, "Smart royalties: Blockchain solutions for music metadata and copyright," *IDEA: The Law Review of the Franklin Pierce Center for Intellectual Property*, vol. 64, no. 1, pp. 1–45, 2023. [Citado na página 15]

[26] Q. Shi and Y. Zhou, "Application of blockchain technology in digital music copyright management: a case study of VNT chain platform," *Frontiers in Blockchain*, vol. 7, 2024. [Online]. Available: https://www.frontiersin.org/journals/blockchain/articles/10.3389/fbloc.2024.1388832/full [Citado na página 15]

[27] V. Mendonca, A. Pillai, E. T. Abraham, K. Crasto, and A. Minu, "Decentralized music marketplace: A blockchain-based rights management approach," in *International Journal of Future Music Research*, vol. 6, no. 2, 2024. [Online]. Available: https://www.ijfmr.com/papers/2024/2/19080.pdf [Citado na página 15]

[28] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," in *Ethereum White Paper*, 2014. [Online]. Available: https://ethereum.org/en/whitepaper/ [Citado na página 23]

[29] K. Dwyer, "Ethereum's testnets explained: Holešky, goerli, sepolia, and more," Blog Ankr, 2023, acedido em setembro 2025. [Online]. Available: https://www.ankr.com/blog/ethereum-testnets-ultimate-guide/ [Citado na página 23]

[30] Solidity Team, "Solidity compiler releases," 2025, acedido em setembro de 2025. [Online]. Available: https://soliditylang.org/blog/category/releases/ [Citado na página 24]

[31] ——, "Solidity documentation," 2025, acedido em setembro de 2025. [Online]. Available: https://docs.soliditylang.org/ [Citado na página 24]

[32] G. Tebrean, "Solidity smart contracts: Gas optimization techniques," 2024, acedido em setembro de 2025. [Online]. Available: https://blog.web3labs.com/solidity-smart-contracts-gas-optimization-techniques/ [Citado na página 24]

[33] Solidity Team, "Solidity developer survey 2024 results," 2025, acedido em setembro de 2025. [Online]. Available: https://soliditylang.org/blog/2025/04/25/solidity-developer-survey-2024-results/ [Citado na página 24]

[34] Nomic Foundation, "Hardhat documentation," 2025, acedido em setembro de 2025. [Online]. Available: https://hardhat.org/docs [Citado nas páginas 24 e 25]

[35] ——, "Guide to testing contracts with Hardhat," 2025, acedido em setembro de 2025. [Online]. Available: https://hardhat.org/hardhat-runner/docs/guides/test-contracts [Citado na página 25]

[36] cgewecke / hardhat-gas-reporter authors, "hardhat-gas-reporter," Disponível no repositório NPM / GitHub, 2025, acedido em setembro de 2025. [Online]. Available: https://www.npmjs.com/package/hardhat-gas-reporter [Citado na página 25]

[37] Nomic Foundation, "Using console.log for Solidity debugging," 2025, acedido em setembro de 2025. [Online]. Available: https://hardhat.org/hardhat-network/docs/console-log [Citado na página 25]

[38] Node.js Foundation / OpenJS Foundation, "About node.js," 2025, acedido em setembro de 2025. [Online]. Available: https://nodejs.org/en/about [Citado na página 26]

[39] I. de Souza, "About npm," 2023, acedido em setembro de 2025. [Online]. Available: https://www.npmjs.com/about [Citado na página 27]

[40] MongoDB Inc., "MongoDB manual," 2025, acedido em setembro de 2025. [Online]. Available: https://www.mongodb.com/docs/ [Citado na página 27]

[41] ——, "GridFS specification," 2025, acedido em setembro de 2025. [Online]. Available: https://www.mongodb.com/docs/manual/core/gridfs/ [Citado na página 27]

[42] FFmpeg Developers, "Ffmpeg – a complete, cross-platform solution to record, convert and stream audio and video," https://ffmpeg.org/documentation.html, 2025, acedido em: 16 de Outubro de 2025. [Citado na página 27]

[43] L. Andrei, "O que é HTML: O guia definitivo para iniciantes," Available at https://www.hostinger.com.br/tutoriais/o-que-e-html-conceitos-basicos, 2023, acedido em setembro de 2025. [Citado na página 28]

[44] W3C, "HTML5 specification," 2014, acedido em setembro de 2025. [Online]. Available: https://www.w3.org/TR/html5/ [Citado na página 28]

[45] J. Alberico, "HTML: o que é, qual a sua importância para a web," 2024, acedido em setembro de 2025. [Citado na página 29]

[46] G. Ariane, "O que é CSS? Guia Básico para Iniciantes," Available at https://www.hostinger.com.br/tutoriais/o-que-e-css-guia-basico-de-css, Dec. 2022, acedido em setembro de 2025. [Citado na página 29]

[47] W3C, "CSS snapshot 2023," 2023, acedido em setembro de 2025. [Online]. Available: https://www.w3.org/TR/css-2023/ [Citado na página 29]

[48] Mozilla Developer Network (MDN), "Javascript," Doc. Web — Mozilla, 2025, acedido em setembro de 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript [Citado na página 29]

[49] MDN Web Docs, "JavaScript guide," 2025, acedido em setembro de 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide [Citado na página 29]

[50] ECMA International, "ECMAScript 2023 language specification," 2023, acedido em setembro de 2025. [Online]. Available: https://262.ecma-international.org/ [Citado na página 29]

[51] Tailwind Labs, "Tailwind CSS: Utility-first CSS framework," 2024, acedido em setembro de 2025. [Online]. Available: https://tailwindcss.com/blog [Citado na página 30]

[52] ——, "Tailwind CSS documentation," 2025, acedido em setembro de 2025. [Online]. Available: https://tailwindcss.com/docs [Citado na página 30]

[53] F. Javed and J. Mangues-Bafalluy, "An empirical smart contracts latency analysis on Ethereum blockchain for trustworthy inter-provider agreements,"

2025. [Online]. Available: https://arxiv.org/abs/2503.01397 [Citado nas páginas 92 e 93]

Anexo A

Listagens de Código

Nesta secção são apresentadas algumas das listagens completas de código-fonte que suportam a implementação do sistema SoundSlice, incluindo controladores e rotas.

A.1 Exportação de *Mixes*

O código abaixo representa a implementação completa do *endpoint* `POST /api/-mix/export`, responsável por receber o ficheiro final da *mix*, gerar os metadados SC4M/IMAF e criar o respetivo registo no MongoDB.

```
1  router.post("/export", requireAuth, upload.single("file"), async (
       req, res) => {
2      try {
3          let snippets = [];
4          if (req.body.snippets) {
5              try {
6                  snippets = JSON.parse(req.body.snippets);
7              } catch (e) {
8                  console.warn("Snippets inválidos recebidos:", req.
                       body.snippets);
9                  snippets = [];
10             }
11         }
12
13         const title = req.body.title || "Mix sem nome";
```

```
14          if (!req.file) return res.status(400).send("Falta o
                ficheiro mix");
15
16          const currentUser = req.user;
17          const fullName = [currentUser.firstname, currentUser.
                lastname].filter(Boolean).join(" ");
18          const bucket = new GridFSBucket(mongoose.connection.db, {
                bucketName: "uploads" });
19          const uploadStream = bucket.openUploadStream(req.file.
                originalname, {
20              contentType: req.file.mimetype,
21              metadata: { type: "mix" },
22          });
23
24          uploadStream.end(req.file.buffer);
25
26          uploadStream.on("finish", async () => {
27              try {
28                  console.log("Upload concluído, ID do ficheiro:",
                        uploadStream.id);
29                  let totalValueEth = 0;
30                  const reusedSnippets = [];
31
32                  for (const s of snippets) {
33                      let valueEth = 0;
34                      try {
35                          const reusedTrack = await Track.findById(s
                                .id);
36                          if (reusedTrack?.metadata?.basePrice) {
37                              const wei = BigInt(reusedTrack.
                                    metadata.basePrice);
38                              valueEth = parseFloat(ethers.
                                    formatEther(wei));
39                          }
40                      } catch (e) {
41                          console.warn("Erro ao buscar preço base
                                para track", s.id);
42                      }
43
44                      totalValueEth += valueEth;
45                      reusedSnippets.push({
46                          trackId: s.id,
47                          title: s.title,
48                          artist: s.artist,
49                          creatorName: s.creatorName || s.artist ||
                                fullName,
50                          creatorWallet: s.creatorWallet || "0
                                x0000000000000000000000000000000000000000
                                ",
```

```
51                      start: s.start,
52                      end: s.end,
53                      percent: s.percent || 0,
54                      snippetId: s.snippetId,
55                      valueEth: valueEth.toFixed(6),
56                      totalValueWei: ethers.parseEther(
                            totalValueEth.toFixed(6)).toString(),
57                  });
58              }
59
60              const metadata = {
61                  format: req.file.mimetype,
62                  size: req.file.size,
63                  duration: snippets.reduce((acc, s) => acc + (s
                        .end - s.start), 0),
64                  hash: "0x" + crypto.createHash("sha256").
                        update(req.file.buffer).digest("hex"),
65                  basePrice: "0",
66                  totalValueEth: totalValueEth.toFixed(6),
67                  sc4m: {
68                      version: "1.0",
69                      usage: "music-mix",
70                      reusedSnippets
71                  },
72                  imaf: {
73                      version: "1.0",
74                      mix: {
75                          tracks: snippets.map(s => ({
76                              id: s.id,
77                              title: s.title,
78                              start: s.start,
79                              end: s.end,
80                              url: s.url,
81                          })),
82                      },
83                  },
84                  creator: currentUser.ethWallet || currentUser.
                        _id.toString(),
85              };
86
87              const track = await Track.create({
88                  title,
89                  artist: fullName || "Desconhecido",
90                  visibility: "private",
91                  fileId: uploadStream.id,
92                  metadata,
93                  user: currentUser._id,
94                  totalReuses: 0,
95                  type: "mix",
```

```
96                  });
97
98                  console.log("Track criada com sucesso:", track._id
                        );
99
100                 // (A integração blockchain é tratada no capítulo
                        seguinte)
101
102                 res.json({ success: true, trackId: track._id });
103             } catch (err) {
104                 console.error("Erro ao criar Track:", err);
105                 res.status(500).send("Erro ao criar track do mix")
                        ;
106             }
107         });
108
109         uploadStream.on("error", (err) => {
110             console.error("Erro upload GridFS:", err);
111             res.status(500).send("Erro ao guardar ficheiro mix");
112         });
113     } catch (err) {
114         console.error("Erro export mix:", err);
115         res.status(500).send("Erro ao exportar mix");
116     }
117 });
```

Listagem A.1: Exportação e registo de uma *mix*.

A.2 Seleção de Trechos

O código abaixo representa a implementação completa do *endpoint* `POST /api/reuse/select`, responsável por selecionar o excerto e cortar com a ferramenta FFMPEG.

```
1 router.post("/:id/reuse/select", requireAuth, async (req, res) =>
     {
2     try {
3         const { start, end } = req.body;
4         const track = await Track.findById(req.params.id).populate
              ("user");
5
6         if (!track || !track.fileId) {
7             return res.status(404).json({ error: "Música não
                  encontrada" });
8         }
9
```

```
10        const duration = track.metadata?.duration || 1;
11        const length = end - start;
12        const percent = Math.min(100, Math.max(1, Math.round((
              length / duration) * 100)));
13
14        const basePrice = BigInt(track.metadata?.basePrice ||
              1000000000000000n);
15        const value = (basePrice * BigInt(percent)) / 100n;
16
17        const payTo = track.user?.ethWallet || "0
              x0000000000000000000000000000000000000000";
18
19        const tempInput = path.join(os.tmpdir(), `${track._id}-${
              Date.now()}.mp3`);
20        const snippetPath = path.join(os.tmpdir(), `snippet-${Date
              .now()}.mp3`);
21
22        const bucket = new mongoose.mongo.GridFSBucket(mongoose.
              connection.db, { bucketName: "uploads" });
23        const downloadStream = bucket.openDownloadStream(track.
              fileId);
24        const writeStream = fs.createWriteStream(tempInput);
25
26        await new Promise((resolve, reject) => {
27            downloadStream.pipe(writeStream);
28            downloadStream.on("error", reject);
29            writeStream.on("finish", resolve);
30        });
31
32        ffmpeg(tempInput)
33            .setStartTime(start)
34            .setDuration(length)
35            .audioCodec("libmp3lame")
36            .on("end", async () => {
37                try {
38                    const snippetId = new mongoose.Types.ObjectId
                          ();
39                    const uploadStream = bucket.
                          openUploadStreamWithId(snippetId, `snippet-
                          ${Date.now()}.mp3`, {
40                        contentType: "audio/mpeg",
41                    });
42
43                    const readStream = fs.createReadStream(
                          snippetPath);
44                    readStream.pipe(uploadStream);
45
46                    uploadStream.on("finish", () => {
```

```
47                            const buffer = fs.readFileSync(snippetPath
                                 );
48                            const hash = "0x" + crypto.createHash("
                                 sha256").update(buffer).digest("hex");
49
50                            res.json({
51                                originalId: track._id,
52                                start,
53                                end,
54                                percent,
55                                snippetId: uploadStream.id.toString(),
56                                snippetName: uploadStream.filename,
57                                value: value.toString(),
58                                payTo,
59                                hash,
60                            });
61
62                            fs.unlinkSync(tempInput);
63                            fs.unlinkSync(snippetPath);
64                        });
65                    } catch (err) {
66                        console.error("Erro ao guardar snippet:", err)
                             ;
67                        res.status(500).json({ error: "Erro ao guardar
                              snippet" });
68                    }
69                })
70                .on("error", (err) => {
71                    console.error("Erro FFmpeg:", err);
72                    res.status(500).json({ error: "Erro ao gerar
                          snippet" });
73                })
74                .save(snippetPath);
75        } catch (err) {
76            console.error("Erro geral:", err);
77            res.status(500).json({ error: "Erro na seleção de trecho"
                 });
78        }
79    });
```

Listagem A.2: Seleção e armazenamento do trecho reutilizado.

A.3 Confirmação da Reutilização

O código abaixo representa a implementação completa do *endpoint* `POST /api/-reuse/confirm`, responsável por confirmar a operação e criar uma nova faixa na coleção `Track`.

```
router.post("/:id/reuse/confirm", requireAuth, async (req, res) =>
    {
    try {
        const { start, end, percent, snippetId, snippetName,
            snippetHash, paymentTxHash } = req.body;
        const track = await Track.findById(req.params.id).populate
            ("user");
        if (!track) return res.status(404).json({ error: "Música
            original não encontrada" });

        const basePrice = BigInt(track.metadata?.basePrice ||
            1000000000000000n);
        const value = (basePrice * BigInt(percent)) / 100n;

        const snippetObjectId = new mongoose.Types.ObjectId(
            snippetId);
        const reusedTrack = new Track({
            title: track.title + " (Reuso)",
            artist: track.artist || "Desconhecido",
            visibility: "private",
            reusedFrom: track._id,
            reusePercentage: percent,
            fileId: snippetObjectId,
            coverId: track.coverId || null,
            user: req.user.id,
            type: "reuse",
            metadata: {
                format: "audio/mpeg",
                duration: end - start,
                basePrice: value.toString(),
                originalBasePrice: track.metadata?.basePrice?.
                    toString() || "0",
                percent,
                sc4m: {
                    schema: "https://schema.soundslice.dev/sc4m.
                        json",
                    version: "1.0",
                    usage: "music-reuse",
                    originalId: track._id.toString(),
                    creatorWallet: track.user?.ethWallet || "0x0",
                    value: value.toString(),
                    paymentTxHash: paymentTxHash || null,
                },
                imaf: {
                    schema: "https://schema.soundslice.dev/imaf.
                        json",
                    version: "1.0",
                    reuseOf: track._id.toString(),
```

```
40                          track: snippetName || `snippet-${snippetId}.
                                mp3`,
41                          snippetId: snippetObjectId.toString(),
42                          start,
43                          end,
44                          percent,
45                          snippetHash,
46                      },
47                  },
48              });
49
50              await reusedTrack.save();
51              track.totalReuses = (track.totalReuses || 0) + 1;
52              await track.save();
53
54              res.json({ success: true, reusedTrack });
55          } catch (err) {
56              console.error(err);
57              res.status(500).json({ error: "Erro ao confirmar reutiliza
                    ção" });
58          }
59      });
```

Listagem A.3: Criação do registo de reutilização.

Anexo B

Contratos Digitais

Este anexo apresenta o código-fonte dos contratos inteligentes desenvolvidos em Solidity para a plataforma SoundSlice.

B.1 MusicRights.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  contract MusicRights {
5      address public owner;
6      string public title;
7      string public artist;
8      bytes32 public fileHash;   // hash SHA256 do ficheiro
9      uint256 public basePrice;  // preco base em wei
10     string public format;
11     string public genre;
12     uint256 public duration;   // segundos
13     string public extra;       // JSON de metadados adicionais
14
15     address[] public holders;  // co-titulares
16     uint256[] public shares;   // percentagens correspondentes
17
18     struct Reuse {
19         address user;
```

```
20          uint256 percentage;
21          uint256 timestamp;
22          uint256 valuePaid;
23      }
24
25      Reuse[] public reuses;
26
27      constructor(
28          string memory _title,
29          string memory _artist,
30          bytes32 _fileHash,
31          uint256 _basePrice,
32          string memory _format,
33          string memory _genre,
34          uint256 _duration,
35          string memory _extra,
36          address[] memory _holders,
37          uint256[] memory _shares
38      ) {
39          require(_holders.length == _shares.length, "Titulares e
                percentagens desalinhados");
40          uint256 totalShare = 0;
41          for (uint256 i = 0; i < _shares.length; i++) {
42              totalShare += _shares[i];
43          }
44          require(totalShare == 100, "As percentagens devem somar
                100");
45
46          owner = msg.sender;
47          title = _title;
48          artist = _artist;
49          fileHash = _fileHash;
50          basePrice = _basePrice;
51          format = _format;
52          genre = _genre;
53          duration = _duration;
54          extra = _extra;
55          holders = _holders;
56          shares = _shares;
57      }
58
59      function registerReuse(uint256 _percentage) public payable {
60          uint256 expectedPayment = (basePrice * _percentage) / 100;
61          require(msg.value >= expectedPayment, "Valor insuficiente
                enviado");
62
63          reuses.push(Reuse({
64              user: msg.sender,
65              percentage: _percentage,
```

```
66              timestamp: block.timestamp,
67              valuePaid: msg.value
68          }));
69
70          // distribuir pagamentos proporcionalmente pelos titulares
71          for (uint256 i = 0; i < holders.length; i++) {
72              uint256 part = (msg.value * shares[i]) / 100;
73              payable(holders[i]).transfer(part);
74          }
75      }
76
77      function getReuse(uint256 index) public view returns (address,
             uint256, uint256, uint256) {
78          Reuse memory r = reuses[index];
79          return (r.user, r.percentage, r.timestamp, r.valuePaid);
80      }
81
82      function totalReuses() public view returns (uint256) {
83          return reuses.length;
84      }
85
86      function getHolders() public view returns (address[] memory,
            uint256[] memory) {
87          return (holders, shares);
88      }
89  }
```

Listagem B.1: Contrato `MusicRights.sol`

B.2 MusicReuse.sol

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.20;
3
4   /// @title Music Reuse - Contrato inteligente para reutilizacao de
         faxas
5   /// @author
6   /// @notice Regista e protege os direitos de reutilizacao de
        excertos musicais
7
8   contract MusicReuse {
9       // Dados basicos
10      string public originalId;
11      string public originalTitle;
12      string public originalCreator;
13      address public creatorWallet;
```

```
14
15      // Reuso
16      string public reuseId;
17      string public reuserName;
18      address public reuserWallet;
19
20      uint256 public reusePercent;
21      uint256 public valuePaid;
22      uint256 public timestamp;
23
24      // Hashes e integridade
25      bytes32 public originalFileHash;
26      bytes32 public snippetHash;
27
28      // Blockchain info
29      string public format;
30      string public genre;
31      uint256 public snippetDuration;
32
33      struct ReuseData {
34          string reuseId;
35          string originalId;
36          string originalTitle;
37          string originalCreator;
38          address creatorWallet;
39          string reuserName;
40          address reuserWallet;
41          uint256 reusePercent;
42          uint256 valuePaid;
43          bytes32 originalFileHash;
44          bytes32 snippetHash;
45          string format;
46          string genre;
47          uint256 snippetDuration;
48      }
49
50      event ReuseRegistered(
51          string reuseId,
52          string originalId,
53          string originalTitle,
54          string reuserName,
55          uint256 reusePercent,
56          uint256 valuePaid,
57          bytes32 snippetHash,
58          uint256 timestamp
59      );
60
61      constructor(ReuseData memory data) {
62          reuseId = data.reuseId;
```

```
63          originalId = data.originalId;
64          originalTitle = data.originalTitle;
65          originalCreator = data.originalCreator;
66          creatorWallet = data.creatorWallet;
67
68          reuserName = data.reuserName;
69          reuserWallet = data.reuserWallet;
70
71          reusePercent = data.reusePercent;
72          valuePaid = data.valuePaid;
73          originalFileHash = data.originalFileHash;
74          snippetHash = data.snippetHash;
75          format = data.format;
76          genre = data.genre;
77          snippetDuration = data.snippetDuration;
78
79          timestamp = block.timestamp;
80
81          emit ReuseRegistered(
82              data.reuseId,
83              data.originalId,
84              data.originalTitle,
85              data.reuserName,
86              data.reusePercent,
87              data.valuePaid,
88              data.snippetHash,
89              timestamp
90          );
91      }
92  }
```

Listagem B.2: Contrato `MusicReuse.sol`

B.3 MusicMix.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  /// @title Music Mix - Contrato inteligente para mixagem de varias
       faixas musicais
5  /// @author
6  /// @notice Regista a autoria e os direitos de um mix criado a
       partir de varias musicas
7
8  contract MusicMix {
9      struct Source {
```

```
10          string originalId;        // ID da musica original (MongoDB
                ou MusicRights)
11          string title;             // Titulo da musica original
12          string creatorName;       // Nome do criador original
13          address creatorWallet;    // Wallet do criador original
14          uint256 reusePercent;     // Percentagem usada no mix
15          uint256 valueShare;       // Valor correspondente (em wei)
16      }
17
18      string public mixId;          // ID interno do mix
19      string public mixTitle;       // Titulo do mix
20      string public mixCreator;     // Nome do criador do mix
21      address public mixWallet;     // Wallet do criador do mix
22      string public format;         // Ex: mp3
23      string public genre;          // Ex: Pop, Jazz, etc.
24      uint256 public timestamp;     // Quando foi criado
25
26      Source[] public sources;      // Lista de musicas originais
            reutilizadas
27
28      event MixRegistered(
29          string mixId,
30          string mixTitle,
31          address mixWallet,
32          uint256 totalSources,
33          uint256 timestamp
34      );
35
36      constructor(
37          string memory _mixId,
38          string memory _mixTitle,
39          string memory _mixCreator,
40          address _mixWallet,
41          string memory _format,
42          string memory _genre,
43          Source[] memory _sources
44      ) {
45          mixId = _mixId;
46          mixTitle = _mixTitle;
47          mixCreator = _mixCreator;
48          mixWallet = _mixWallet;
49          format = _format;
50          genre = _genre;
51
52          for (uint256 i = 0; i < _sources.length; i++) {
53              sources.push(_sources[i]);
54          }
55
56          timestamp = block.timestamp;
```

```
57
58          emit MixRegistered(_mixId, _mixTitle, _mixWallet, _sources
                .length, timestamp);
59      }
60
61      /// @notice Devolve todas as musicas usadas no mix
62      function getSources() public view returns (Source[] memory) {
63          return sources;
64      }
65  }
```

Listagem B.3: Contrato `MusicMix.sol`