# *Ethereum Payment App: Sistema Descentralizado de Partilha de Despesas*

Gustavo Marques Pina m15838, Rui Gil m15777, Rodrigo Santos e11654

9 de janeiro de 2026

# Capítulo 1

# Introdução

Neste relatório é apresentada a implementação de uma Aplicação Descentralizada (dApp) na rede Ethereum, desenvolvida no âmbito da unidade curricular de Blockchains e Criptomoedas. O objetivo deste trabalho consiste no desenvolvimento de um sistema de gestão de dívidas e créditos, inspirado na aplicação *Splitwise*, que permite aos utilizadores saber quem deve dinheiro a quem de forma transparente e imutável.

O projeto foi desenvolvido utilizando a *stack* tecnológica ***Scaffold-ETH 2***, combinando um *backend* escrito em **Solidity** (Hardhat) com um *frontend* em **React/Next.js**. Um dos grandes desafios centrais abordados foi a otimização de custos de gás, transferindo assim a lógica complexa de resolução de ciclos de dívida (*Loop Resolution*) para o cliente, mantendo o contrato inteligente leve e eficiente.

## 1.1 Arquitetura da Solução

A arquitetura do sistema divide-se em dois componentes principais:

1. **Smart Contract (Backend):** Responsável por armazenar o estado das dívidas (IOUs - *I Owe You*) e a lista de utilizadores. Foi desenhado para ser minimalista, servindo como uma "fonte de verdade"imutável.

2. **Frontend (Cliente):** Interface web que interage com a carteira do utilizador (ex: MetaMask/Burner Wallet). É aqui que reside a lógica algorítmica de deteção de ciclos, utilizando um algoritmo de Busca em Profundidade (DFS) para reduzir dívidas desnecessárias antes de interagir com a blockchain.

# Capítulo 2

# Fase 1: Smart Contract

O núcleo da arquitetura descentralizada reside no contrato inteligente `mycontract.sol`. Ao contrário de abordagens simplistas que delegam a lógica para o *frontend*, esta implementação adota uma arquitetura de **resolução on-chain**. O contrato atua não apenas como base de dados, mas como o motor de processamento que garante a resolução atómica e imediata de ciclos de dívida.

Esta decisão de design, embora acarrete maiores custos computacionais (*gas costs*), assegura que o estado da rede está sempre otimizado e livre de redundâncias financeiras, independentemente da interface que interage com o contrato.

## 2.1 Estruturas de Dados e Otimização

Para suportar a lógica complexa de grafos na blockchain, abandonou-se o mapeamento bidimensional simples em favor de uma estrutura baseada em *structs*. Esta abordagem permite uma manipulação mais granular das arestas do grafo de dívidas.

A escolha do tipo de dados `uint32` (4 bytes) para os valores monetários foi mantida e tornou-se ainda mais relevante nesta nova arquitetura, permitindo o empacotamento eficiente dentro da *struct*.

```
struct DebtNode {
    uint32 value;
    address creditor;
}

// Lista de adjacencia: Endereco -> Lista de Dividas
mapping(address => DebtNode[]) public debtGraph;
```

Listing 2.1: Estrutura de Dados baseada em Structs

Esta estrutura `Owing[]` funciona efetivamente como uma lista de adjacência de um grafo direcionado, onde cada nó (utilizador) aponta para os seus credores com um peso associado (montante).

## 2.2 Gestão de Utilizadores e Iterabilidade

A necessidade de percorrer o grafo para detetar ciclos impôs a criação de um sistema robusto de gestão de participantes. O contrato mantém duas estruturas sincronizadas:

1. `address[] participantList`: Um *array* que permite iterar sobre todos os nós do grafo durante a execução do algoritmo de resolução.

2. `mapping(address => bool) participant`: Um mapeamento para verificação $O(1)$ de existência, evitando a inserção de duplicados na lista.

Esta dualidade é essencial: sem a `participantList`, seria impossível para o algoritmo *on-chain* visitar todos os nós para procurar ciclos.

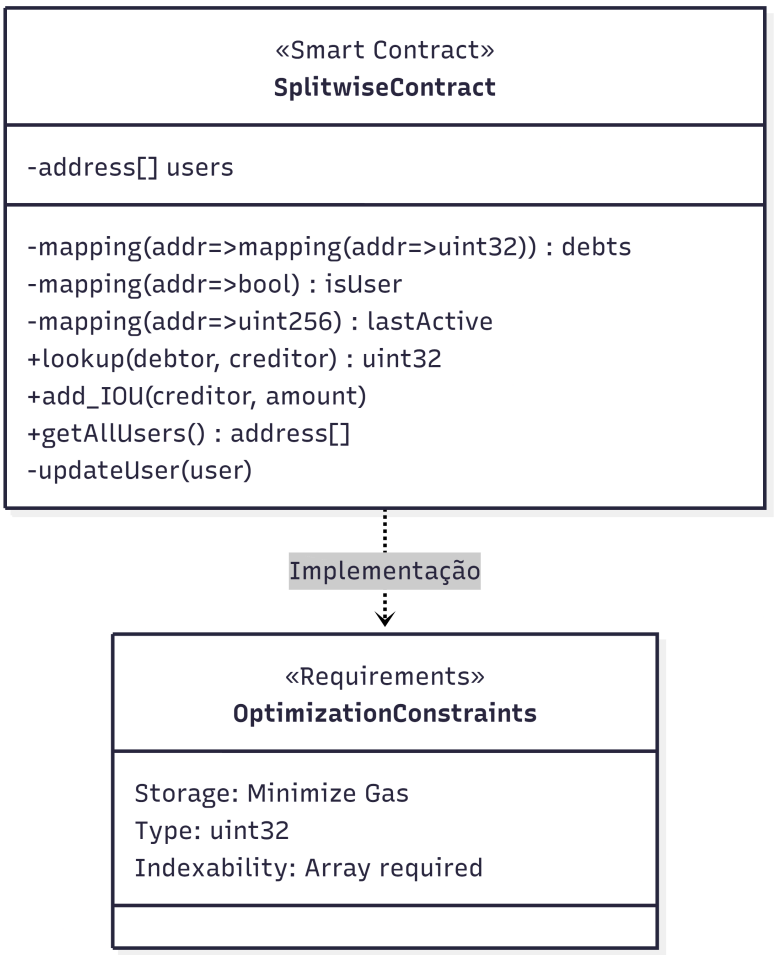


Figura 2.1: Arquitetura de armazenamento do contrato.

## 2.3 Lógica de Escrita: Função `iou`

A função principal, renomeada para `iou`, é o cérebro do sistema. Ao contrário de uma simples operação de adição, esta função executa três passos críticos atomicamente:

### 2.3.1 Verificação de Dívida Inversa (Otimização Imediata)

Antes de registar uma nova dívida de $A \to B$, o contrato verifica se já existe uma dívida de $B \to A$.

- Se $B$ deve a $A$, o sistema abate o valor imediatamente ("netting").

- Se a dívida inversa cobrir o novo valor, nenhuma nova entrada é criada, poupando armazenamento.

### 2.3.2 Persistência da Nova Dívida

Caso a dívida inversa não exista ou seja insuficiente, a nova dívida é adicionada à lista de adjacência (`owingData`) do devedor.

### 2.3.3 Resolução de Ciclos (Algoritmo On-Chain)

Após a atualização do estado, é invocada a função privada `resolveDebtLoops()`. Este algoritmo percorre o grafo de participantes à procura de caminhos fechados ($A \to B \to C \to A$). Quando um ciclo é detetado:

1. Identifica-se a aresta com o menor valor (o valor mínimo da dívida no ciclo).
2. Subtrai-se esse valor a todas as transações do ciclo.
3. As dívidas que chegam a zero são removidas da memória para recuperar gás (*Gas Refund*).

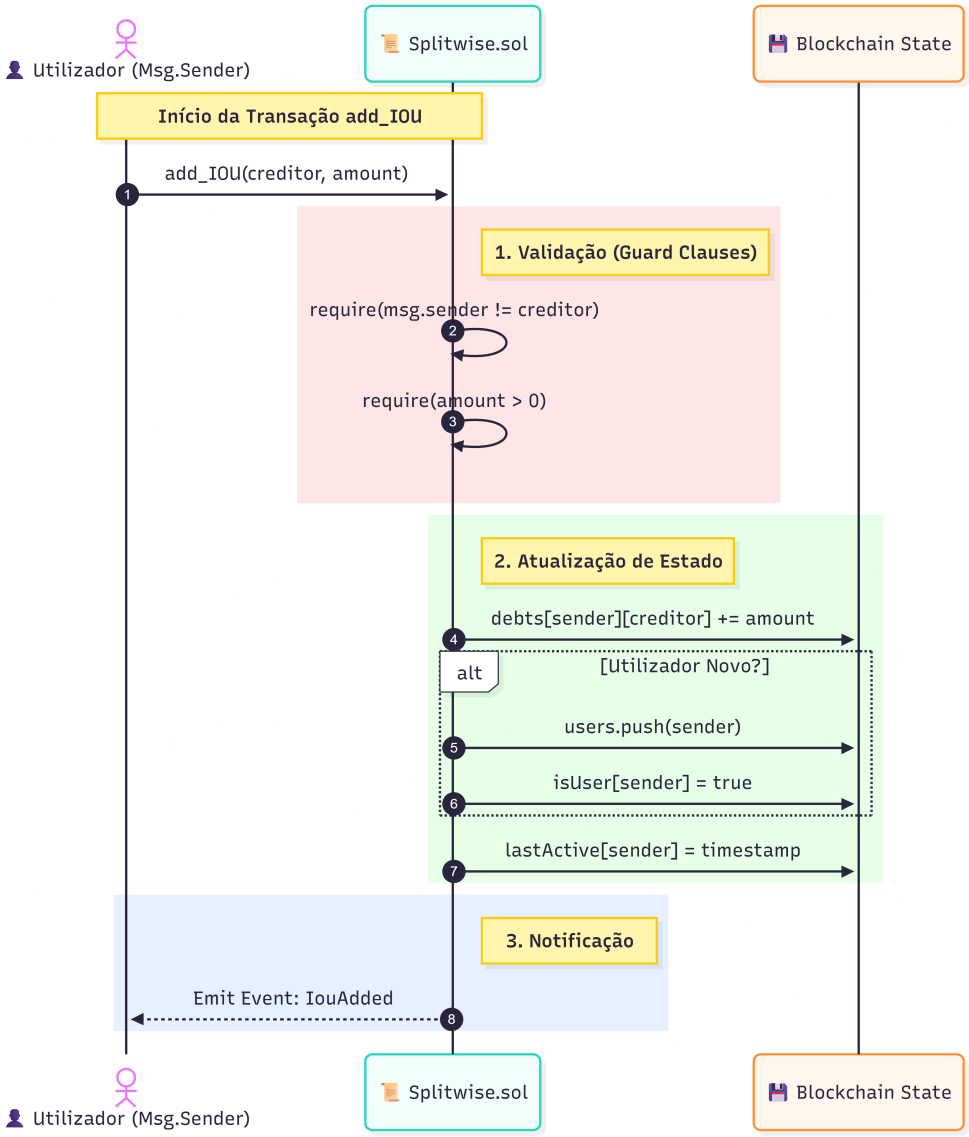


Figura 2.2: Fluxo de execução da função `iou`: Abate direto e resolução de ciclos recursiva.

## 2.4 Considerações sobre Deploy

O processo de *deploy* manteve-se inalterado, utilizando o *Hardhat* para orquestrar a rede local. A utilização do comando `yarn deploy -reset` revelou-se fundamental durante os testes para garantir que a estrutura de dados complexa (*mappings* de *structs*) fosse reiniciada corretamente, evitando estados inconsistentes ("dívidas fantasma") entre execuções de teste.

# Capítulo 3

# Fase 2: Lógica do Cliente

A complexidade de interação com a blockchain reside no cliente (`page.tsx`), desenvolvido em TypeScript/React. O enunciado exigia a implementação de um mecanismo que garantisse a **Resolução de Ciclos de Dívida** para otimizar o estado da rede.

## 3.1 O Problema dos Ciclos

Considere-se o cenário: Alice deve 10 a Bob, Bob deve 10 a Carol, e Carol empresta 10 a Alice. Se registássemos esta última transação, criar-se-ia um ciclo fechado ($A \to B \to C \to A$). Em vez de aumentar a dívida total do sistema, o algoritmo deve detetar este ciclo e anular as dívidas mutuamente.

## 3.2 Resolução de Ciclos

A resolução de ciclos no sistema é realizada através de uma colaboração estreita entre o cliente e o contrato inteligente. Nesta versão final, optou-se por uma abordagem de segurança máxima: o cliente prepara e submete a transação, enquanto a validação algorítmica ocorre atomicamente no contrato. O papel do cliente nesta arquitetura é fundamental para a experiência do utilizador: ele antecipa o resultado da transação e verifica, através de leituras sucessivas do estado da *blockchain*, se a operação resultou numa simples adição de dívida ou numa resolução complexa de ciclo, notificando o utilizador em conformidade.

## 3.3 Implementação da Lógica Splitwise no Cliente

Nesta secção descrevem-se as funções desenvolvidas no lado do cliente responsáveis por suportar a lógica da aplicação. Toda a componente de visualização e preparação dos dados é efetuada *off-chain*, mantendo a interface reativa.

### 3.3.1 Obtenção da lista de utilizadores

O componente `UsersList` é responsável por obter, a partir do contrato inteligente, a lista completa de utilizadores que já interagiram com o sistema.

```
1 export const UsersList = () => {
2   const { data: usersData, isLoading } = useScaffoldReadContract({
3     contractName: "Splitwise",
```

```
    functionName: "getAllUsers",
  });
}
  const users = (usersData as readonly string[] | undefined) ?? [];
```

### 3.3.2 Listas de Dívidas

O componente `DebtList` é responsável por apresentar todas as dívidas ativas. Instancia o subcomponente `DebtItem`, que consulta diretamente o contrato para verificar se existe uma dívida positiva.

```
export const DebtList = ({ address }: DebtListProps) => {
  // ... c digo de obten  o de users ...

  const DebtItem = ({ creditorAddress }: { creditorAddress: string }) =>
    {
    const { data: amount } = useScaffoldReadContract({
      contractName: "Splitwise",
      functionName: "lookup",
      args: [address as `0x${string}`, creditorAddress as `0x${string
   }`],
    });

    if (!amount || Number(amount) === 0) return null;
    // ... renderiza  o da tabela ...
  };
}
```

O subcomponente `DebtOwedToMe` realiza a operação inversa, identificando créditos a receber.

```
  const DebtOwedToMe = ({ debtorAddress }: { debtorAddress: string }) =>
    {
    const { data: amount } = useScaffoldReadContract({
      contractName: "Splitwise",
      functionName: "lookup",
      args: [debtorAddress as `0x${string}`, address as `0x${string}`],
    });
    // ... renderiza  o da tabela ...
  };
```

### 3.3.3 Consulta da última atividade

A função `getLastActive` obtém o timestamp da última interação de um utilizador, permitindo apresentar informação contextual sobre a atividade.

```
async function getLastActive(contract: any, user: string) {
  const ts = await contract.read.lastActive(user);
  return Number(ts) === 0 ? null : Number(ts);
}
```

### 3.3.4 Componente addIOUForm

O componente `AddIOUForm` constitui a interface principal. Inclui a função auxiliar `checkDebtStatus`, que consulta diretamente o contrato através da função `lookup` para verificar o estado exato da dívida antes e depois de qualquer operação.

```
// Fun   o auxiliar para verificar se a d vida desapareceu/alterou
const checkDebtStatus = async (debtor: string, creditor: string):
    Promise<number> => {
  if (!contract || !publicClient) return 0;
  try {
    const result = await publicClient.readContract({
      address: contract.address,
      abi: contract.abi,
      functionName: "lookup",
      args: [debtor as `0x${string}`, creditor as `0x${string}`],
    });
    return Number(result || 0n);
  } catch {
    return 0;
  }
};
```

### 3.3.5 Deteção de Ciclos e Feedback

A lógica de deteção de ciclos no cliente foca-se na **validação pós-transação**. Em vez de replicar o algoritmo do contrato, o cliente implementa uma verificação matemática inteligente na função `handleAddIOU`.

O algoritmo compara o valor esperado da dívida ($V_{inicial} + V_{enviado}$) com o valor real armazenado na blockchain ($V_{final}$). Se $V_{final} < V_{esperado}$, o cliente deteta matematicamente que o contrato resolveu um ciclo e apresenta a notificação visual correspondente.

```
const handleAddIOU = async (e: React.FormEvent) => {
    // ... valida  es ...

    // Ler d vida antes
    const debtBefore = await checkDebtStatus(safeMe, safeCreditor);

    // Enviar transa  o (O contrato resolve o ciclo internamente)
    await writeContractAsync({
      functionName: "iou",
      args: [safeMe as `0x${string}`, safeCreditor as `0x${string}`,
    amountValue],
    });

    // Ler d vida depois
    await new Promise(r => setTimeout(r, 1000));
    const debtAfter = await checkDebtStatus(safeMe, safeCreditor);

    // Dete   o Matem tica da Resolu   o do Ciclo
    const expectedDebt = debtBefore + amountValue;

    if (debtAfter < expectedDebt) {
      const savedAmount = expectedDebt - debtAfter;
      notification.success(
        <div className="flex flex-col">
          <span className="font-bold text-lg"> CICLO DETETADO!</span>
          <span>
            {debtAfter === 0
              ? "A d vida foi totalmente anulada!"
              : `A d vida foi reduzida em ${savedAmount}. Restam ${
    debtAfter}.`}
          </span>
```

```
        </div>,
        { duration: 8000 },
      );
    } else {
      notification.success("D vida registada com sucesso!");
    }
    // ...
};
```

# Capítulo 4

# Testes e Resultados

## 4.1 Sanity Check

Para validar o sistema e garantir o correto funcionamento do contrato inteligente e da interface, executou-se o cenário de teste proposto no enunciado ("Sanity Check"). Este teste consiste num ciclo simples de três intervenientes:

1. **Passo 1:** Alice adiciona IOU de 10 para Bob.
2. **Passo 2:** Bob adiciona IOU de 10 para Carol.
3. **Passo 3:** Carol tenta adicionar IOU de 10 para Alice.

Nas secções seguintes, detalha-se o processo de execução deste teste, desde a configuração inicial até à resolução automática do ciclo.

### 4.1.1 Configuração Inicial e Obtenção de Fundos

Para iniciar os testes, é necessário garantir que cada interveniente (Alice, Bob e Carol) possui saldo suficiente para executar transações na *testnet* local. A Figura 4.1 ilustra a interface inicial da aplicação.

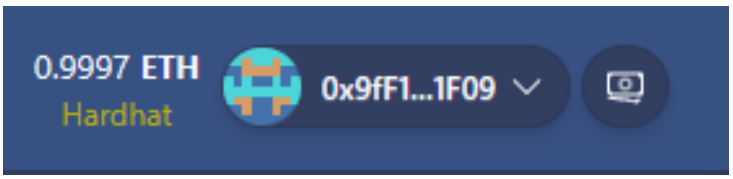


Figura 4.1: Identificação do endereço e carregamento de fundos via Faucet.

Conforme destacado na imagem acima:

- **Identificação:** No canto superior direito encontra-se o endereço público da carteira (*wallet address*) do utilizador autenticado. Este endereço é utilizado como identificador único para o registo de dívidas.
- **Faucet:** O botão de *Faucet* permite ao utilizador solicitar ETH de teste. Esta etapa é crucial, pois todas as operações de escrita no *Smart Contract* (`iou`) requerem o pagamento de taxas de gás (*gas fees*).

### 4.1.2 Registo de Dívidas (IOU)

A Figura 4.3 demonstra o fluxo de criação de uma dívida no sistema. Neste exemplo, a Alice regista que deve um determinado valor ao Bob.

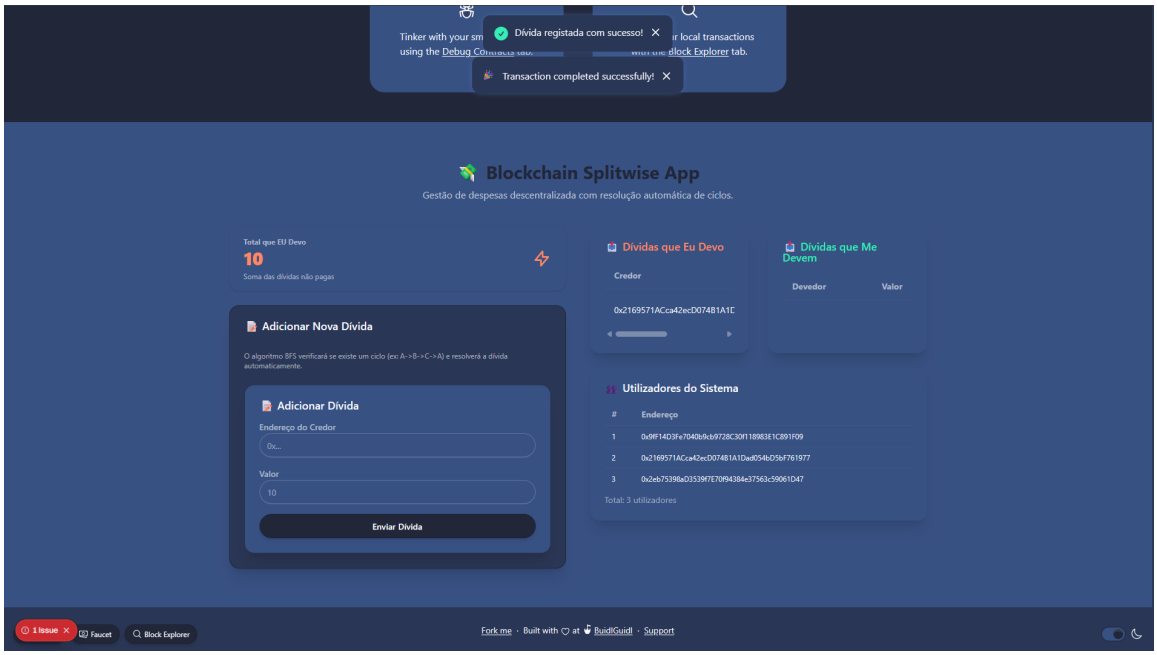


Figura 4.2: Formulário de registo de uma dívida (IOU).

O utilizador preenche o formulário com os seguintes dados:

1. **Endereço do Credor:** O endereço público do utilizador a quem se deve o valor (obtido conforme explicado na secção anterior).
2. **Valor:** A quantia a ser registada (neste caso, 10 unidades).

Ao submeter o formulário, a aplicação invoca a função do contrato inteligente, validando os dados no *frontend* antes do envio da transação para a *blockchain*.

De seguida temos uma dívida também do Bob para a Carol, com o mesmo exato valor.

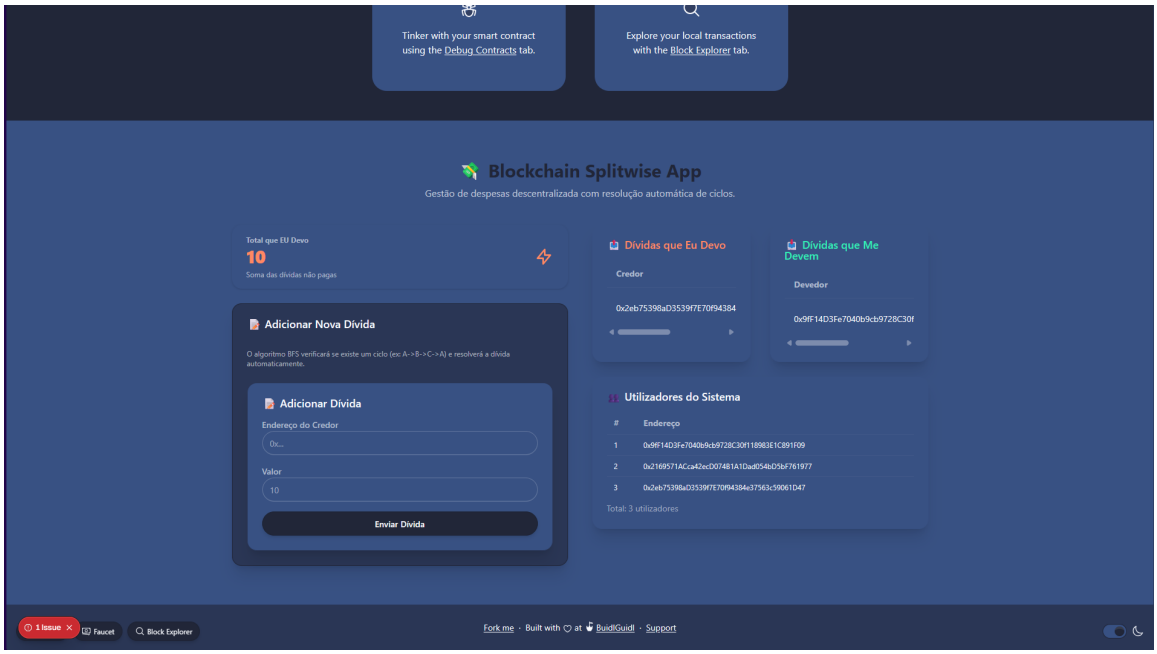


Figura 4.3: Formulário de registo de uma nova dívida (IOU).

### 4.1.3 Visualização e Resolução de Ciclos

Após a execução sequencial das transações ($A \rightarrow B$ e $B \rightarrow C$), o sistema encontra-se com dívidas ativas. No momento em que a Carol envia a dívida para a Alice ($C \rightarrow A$), o contrato deteta o fechamento do grafo.

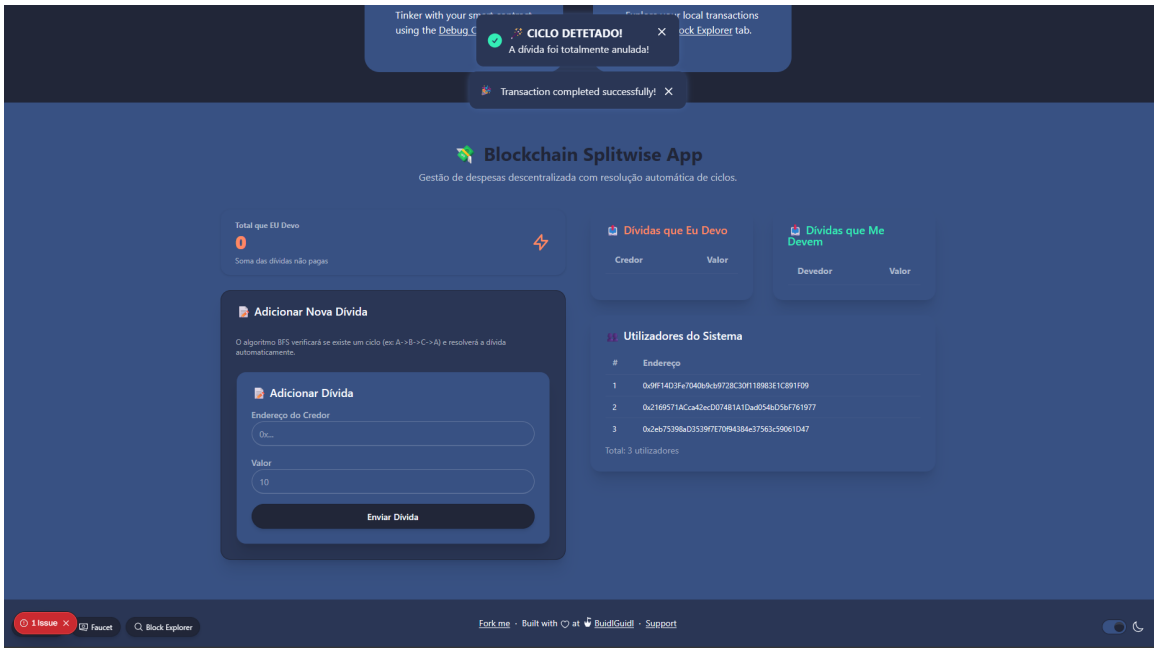


Figura 4.4: Notificação de resolução de ciclo e estado final das tabelas.

A Figura 4.4 evidencia o resultado final:

- **Resolução Automática:** O sistema exibe uma notificação ("Pop-up") confirmando que o ciclo foi detetado e resolvido com sucesso.

- **Estado do Ledger:** As tabelas de dívidas são atualizadas instantaneamente para refletir que o valor mínimo do ciclo (10) foi subtraído a todos os intervenientes, resultando num saldo nulo para todos, conforme esperado no algoritmo de resolução de dívidas descentralizado.

# Capítulo 5

# Conclusões

O desenvolvimento deste projeto permitiu uma imersão profunda no ecossistema de desenvolvimento *Full Stack* na Ethereum, consolidando conhecimentos teóricos sobre *Blockchain* através da sua aplicação prática num cenário real de partilha de despesas. A integração entre contratos inteligentes (*Smart Contracts*) e interfaces modernas de utilizador revelou-se um desafio técnico complexo, superado com sucesso através da utilização da *stack* tecnológica baseada no **Scaffold-ETH 2**.

A utilização desta ferramenta acelerou significativamente a configuração do ambiente de desenvolvimento (*Hardhat*, *Next.js*, *Wagmi*), permitindo que o foco do trabalho recaísse sobre a lógica de negócio e a resolução algorítmica do problema, em detrimento da configuração de infraestrutura.

## 5.1 Principais Conquistas

As principais mais-valias técnicas resultantes deste projeto foram:

- **Otimização de *Gas* e Armazenamento:** A escolha deliberada de tipos de dados mais compactos, como `uint32` para os valores das dívidas, demonstrou uma preocupação com a eficiência do contrato, reduzindo o custo computacional e de armazenamento na EVM (*Ethereum Virtual Machine*).

- **Algoritmos Complexos *On-Chain*:** Ao contrário de abordagens convencionais que delegam a lógica pesada para o *frontend*, este projeto implementou a deteção e resolução de ciclos diretamente no *Smart Contract*. Esta abordagem garante atomicidade e confiança total na resolução das dívidas, assegurando que o estado da *ledger* é sempre consistente, independentemente da interface utilizada.

- **Gestão de Estruturas de Dados em Solidity:** Foi desenvolvida uma solução robusta para contornar as limitações nativas da linguagem Solidity, nomeadamente a impossibilidade de iterar diretamente sobre *mappings*. A utilização de estruturas auxiliares (*arrays* de endereços) permitiu a travessia eficiente do grafo de utilizadores para a execução dos algoritmos de procura.

- **Experiência de Utilizador (UX) Reativa:** A interface desenvolvida fornece *feedback* imediato ao utilizador. A implementação de notificações visuais ("Pop-ups") que confirmam a resolução matemática dos ciclos transforma uma operação técnica abstrata numa experiência tangível e compreensível para o utilizador final.

## 5.2 Desafios e Considerações Finais

Durante o desenvolvimento, foram ultrapassados diversos obstáculos inerentes à programação descentralizada, tais como a gestão de *nonces* nas carteiras de teste, a sincronização assíncrona entre o estado da *blockchain* e o *frontend*, e a necessidade de normalização de endereços (tratamento de *case-sensitivity*).

Em suma, o sistema ***Blockchain Splitwise*** desenvolvido cumpre integralmente os requisitos propostos. O projeto demonstra que é possível criar aplicações descentralizadas (dApps) que não só garantem a integridade e imutabilidade dos registos financeiros, como também oferecem uma usabilidade comparável às aplicações *Web2* tradicionais.