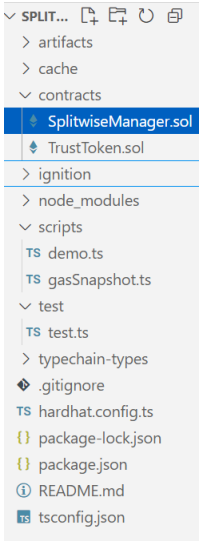
SplitWise

Introduzione

SplitWise è un'applicazione progettata per la suddivisione delle spese tra gruppi di persone tramite blockchain Ethereum.
SplitWise include le funzionalità di **creazione di un gruppo, unione ad un gruppo, registrazione di una spesa nel gruppo, semplificazione del debito** e **pagamento di un debito** tramite fungible token ERC-20.

Struttura del progetto

L'implementazione del progetto utilizza il framework **HardHat,** ed è stato strutturato nei seguenti moduli:



SplitwiseManager.sol: Smart contract scritto in Solidity che implementa le funzionalità principali di SplitWise .
TrustToken.sol: Modulo che implementa un token ERC-20. Emette token in cambio di Ether a parità di valore.
demo.ts: Script typescript che simula il comportamento di vari utenti che creano un gruppo, si uniscono ad un gruppo e registrano una spesa, stampando a video il comportamento del programma.
gasSnapshot.ts: Script typescript che si occupa di calcolare il costo del gas per ogni funzionalità, in base al costo attuale del gas, e lo stampa a video.
test.ts: Suite di test che verificano il funzionamento delle varie feature.

TrustToken.sol

TrustToken implementa un token ERC-20 che permette ad un utente di coniare nuovi gettoni pagando 1 wei di ETH per ottenere 1 unità di token.
TrustToken tramite la keyword "is ERC20" alla dichiarazione del contratto, eredita l'implementazione ERC-20 di OpenZeppelin, e tramite essa l'implementazione delle funzionalità standard ERC-20.

La costante TOKENPRICE indica quanti wei bisogna inviare per ottenere una singola unità di token. La costante è fissata ad 1, in modo da ottenere 1 token per ogni wei.
Il costruttore **constructor() ERC20("TrustToken", "TTK") {}** richiama il costruttore della classe base ERC20, passando il nome del token creato e il suo abbreviativo.
La funzione **mint()** permette ad un utente di acquistare nuovi token semplimente inviando valuta in ETH al contratto. E' dichiarata **external,** poichè può essere chiamata solo da indirizzi esterni e **payable** poichè consente alla funzione di ricevere Ether, accessibili dalla variabile msg.value. La funzione verifica che è necessario inviare una somma almeno pari al TOKENPRICE. Calcola il numero di token da coniare, verifica che esso sia positivo e richiama la funzione _mint() della classe base ERC20, richiedendo

l'ammontare di token richiesto in formato comprensivo di 18 decimali. Se amount = 1, in realtà si sta creando 1×10^{18} unità minime del token.

SplitwiseManager.sol

SplitwiseManager è il modulo che implementa le feature di **creazione di un gruppo, unione ad un gruppo, registrazione di un debito, semplificazione di un debito e pagamento tramite token.**

```
struct Group {
    string groupName;       // nome descrittivo del gruppo
    address[] memberList;       // elenco degli indirizzi dei membri
    mapping(address => bool) isMember;     // lookup O(1) membership
    mapping(address => int256) netBalance;  // saldo netto di ciascun membro
    mapping(address => mapping(address => uint256)) debtGraph; // debtGraph[debitor][creditor] = importo
}
```

La struttura dati di un gruppo è modellata dalla struct **Group**. Esse contiene una stringa che identifica il gruppo, una lista contenente gli indirizzi dei membri, un mapping da address a bool chiamato isMember che verifica se un membro fa parte del gruppo, un mapping che associa ogni indirizzo ad un intero e serve ad esprimere il saldo di ciascun membrom e un mapping da address ad un mapping che va da address ad un intero, che modella il grafo dei debitori, dove debtGraph[debitor][creditor] = importoDebito.
Per verificare se un indirizzo fa parte di un gruppo viene utilizzato un mapping perché computazionalmente il controllo isMember[addr] costa (O(1)), e consuma meno gas rispetto a scorrere un array e verificare per ogni elemento se l'indirizzo coincide (O(n)).

Alla definizione del contratto SplitwiseManager, si definiscono le variabili che delineano il suo stato, gli eventi e gli errori da lanciare. La variabile **trustToken** rappresenta un riferimento al contratto TrustToken ERC-20 usato per regolare i pagamenti tra membri. **NextGroupId** rappresenta un contatore autoincrementale che assegna un ID univoco a ogni nuovo gruppo creato. **Groups** rappresenta un mapping che associa a ciascun groupId la relativa struttura dati Group.

```
event GroupCreated(uint256 indexed groupId, string groupName);
event MemberJoined(uint256 indexed groupId, address member);
event ExpenseAdded(uint256 indexed groupId, string description, uint256 totalAmount);
event DebtsSimplified(uint256 indexed groupId);
event DebtSettled(uint256 indexed groupId, address payer, address payee, uint256 amount);
```

Ogni volta che nel contratto avviene un'azione rilevante, viene emesso un **evento** che permette esternamente di ascoltare e reagire ed essi. **GroupCreated** indica che un nuovo gruppo creato. **MemberJoined** indica che un utente entra in un gruppo. **ExpenseAdded** indica che è stata inserita una spesa da dividere. **DebtsSimplified** indica che i debiti di un gruppo sono stati semplificati. **DebtSettled** indica la registrazione di un pagamento effettivo tra due membri.

```
error NotGroupMember();
error InvalidParameters();
```

Vengono definiti due tipi di errore che possono essere **revertiti** con revert. **NotGroupMember** indica che chi chiama la funzione non è membro del gruppo. **InvalidParameters** indica che viene passata una combinazione di parametri non valida.

```
modifier onlyGroupMember(uint256 groupId) {
    if (!groups[groupId].isMember[msg.sender]) revert NotGroupMember();
    _;
}
```

onlyGroupMember è un decorator che garantisce che le funzioni decorate possano essere eseguite **solo** da un membro del gruppo specificato. Se msg.sender non compare in groups[groupId].isMember, la chiamata si blocca con NotGroupMember. Se il controllo ha successo, esegue la funzione decorata.

```
constructor(address tokenAddress) {
    trustToken = TrustToken(tokenAddress);
}
```

Il **costruttore** inizializza il riferimento a TrustToken assegnandogli l'indirizzo del contratto deployato TrustToken, e salvandolo nella variabile trustToken.
Sono presenti varie funzioni di utility chiamabili dall'esterno del contratto, come **getNetBalance** che restituisce il saldo di un utente di un dato gruppo; **getDebt** che restituisce il debito registrato da un dato debitore verso un dato creditore in un gruppo, e **getGroupMembers** che restituisce la lista degli indirizzi membri di un dato gruppo.
La funzione **_addMemberToGroup è** una funzione interna che controlla che un indirizzo newMember non sia gia presente in un dato gruppo, in tal caso lo aggiunge al gruppo.

La funzione **createGroup** crea un nuovo gruppo e ne restituisce l'id. Vengono passati in input il nome del gruppo e la lista di membri iniziale. Viene incrementato il contatore groupID, viene creato in storage un nuovo gruppo con id groupId, viene aggiunto l'indirizzo dell'utente chiamante come membro del gruppo, vengono aggiunti tutti gli altri membri forniti nel gruppo e viene emesso un evento GroupCreated. Si utilizza nei parametri di input la keyword **calldata** in modo da leggere direttamente l'input dai parametri del payload del messaggio inviato dall'utente senza copiarlo in memoria, risparmiando così gas.

joinGroup è una funzione esterna che abilita a chiamare la funzione interna addMemberToGroup al chiamante sul proprio indirizzo, in modo da unirsi ad un gruppo.

La funzione **addExpense** registra una spesa nel gruppo. Viene applicato il decorator onlyGroupMember. Viene effettuato un check se la spesa è diverso da 0, se sono presenti partecipanti e se l'indirizzo pagante è un membro del gruppo. Vengono calcolate (in memoria) le quote dovute di ciascun partecipante in base al tipo di split passato. Per ogni partecipante, la funzione diminuisce il saldo netto della quota da pagare, incrementa il saldo di chi ha pagato e aggiunge il debito nel grafo dei debiti tra l'utente e colui che ha pagato.

_computeShares è una funzione interna che in base al **tipo di split** e alle eventuali **percentuali** o **quote esatte,** calcola l'array di quote dovute da ogni utente.
Se il tipo di split è in parti uguali, divide l'importo per il numero di partecipanti. Se viene indicata divisione esatta, per ogni partecipante assegna dai dati ausiliari la quota esatta, e infine controlla che la somma delle quote sia uguale all'importo dovuto. Se il tipo di split è in percentuali, calcola la quota per utente come totale * percentuale/10000. Le percentuali dei vari utenti vengono sommate. Si controlla che la somma sia uguale a 10000. Si usa 10000 e non 100 in modo da rappresentare in scala anche i numeri decimali in solidity, come 100.00.

settleDebt è una funzione che registra il pagamento on-chain di un debito usando token ERC-20. Viene preso l'importo dal chiamante al creditore dal grafo dei debiti. Viene controllato se l'importo sia maggiore di 0 e inferiore uguale al debito registrato. Viene richiesto di trasferire l'importo dal debitore al creditore. Subito dopo viene effettutata la require:se il trasferimento è andato a buon fine continua, altrimenti revert. Viene aggiornato il grafo del debito e i rispettivi bilanci.

simplifyDebts è una funzione che semplifica il grafo del debito secondo l'algoritmo greedy. Vengono creati 4 array: un **array dei creditori** affiancato da un array con i corrispettivi saldi, e un array dei debitori affiancato da un array con i loro corrispettivi saldi.

Ogni membro viene inserito nell'array appropriato con corrispettivo bilancio. I due (4) array vengono **ordinati** in modo decrescente. L'ordinamento utilizza insertionSort.
Viene **azzerato** il debtGraph. Effettua un ciclo fino ad arrivare alla lunghezza dei due array, indicizzato da due indici per i rispettivi array, credIndex e debtIndex.
Ad ogni **iterazione**, viene scelto come importo il **minore** tra i saldi del maggiore creditore e il maggiore debitore, e viene **costruito un arco** nel grafo dei debiti tra essi.
Viene decrementato il saldo dei due array interni creditorBalance e debtorBalance (no storage) dell'importo registrato. Nel caso in cui il saldo del creditore o del debitore si azzeri, l'indice nell'array creditor o debitor viene incrementato, avanzando.
Il ciclo poi viene iterato, finchè non si esauriscono gli elementi di entrambi gli array.

Demo.ts

Demo.ts è uno script typescript che deploya il contratto e simula il comportamento degli utenti con lo smart contract Splitwise.
La funzione **await ethers.getSigners()** ritorna un array di oggetti Signer corrispondenti agli account di test messi a disposizione da Hardhat. Li chiamiamo alice, bob e carl.
ethers.deployContract("TrustToken") crea e inviare automaticamente la transazione di deploy del contratto TrustToken e restituisce immediatamente un oggetto Contract il cui deployment è in corso.
await token.waitForDeployment(); sospende l'esecuzione finché la transazione di deploy non viene confermata su un blocco. Vengono eseguite queste due istruzioni sia per TrustToken che Splitwisemanager.
Viene effettuata una transazione da Alice, che **crea un gruppo** chiamato Pizza Night, indicando come membri bob e carl.
Il metodo manager**.connect(alice)** crea una nuova connessione che imposta msg.sender = alice.address per tutte le chiamate successive. La Transaction Response tx contiene i dati della transazione. La **call tx.wait()** sospende l'esecuzione finché la rete non conferma la transazione.
Dalla transazione viene selezionato l'evento GroupCreated e da esso viene estratto il groupID del gruppo, per poi stampare a video l'avvenuta creazione del gruppo groupID.
Alice aggiunge una spesa di 90 divisa in parti uguali tra i membri tramite la funzione **addExpense.** Viene stampato, ciclando, il grafo dei debiti, chiamando la funzione **getDebt** per ogni creditore e debitore, e stampando a video.
Viene semplificato il grafo dei debiti tramite la chiamata **simplifyDebts**.
In modo del tutto simile, vengono aggiunte delle spese in parti uguali da carl e bob, mostrando la semplificazione del grafo. Vengono poi creati altri due gruppi, in cui si testa la divisione in quote esatte e in percentuali fissate.

Test.ts

Modulo typescript che contiene i test sugli smart contract implementati, eseguiti dal framework hardhat. Lo script dichiara due variabili **factory** per splitwisemanager e trustToken. Le factory sono oggetti che permettono di deployare nuove istanze dei contratti in modo tipizzato.
Vengono dichiarate due **variabili** token e manager che conterranno le **istanze deployate** dei due smart contract.
Vengono dichiarate quattro variabili dove successivamente verranno assegnati gli account di test. Simulano gli utenti nelle chiamate dei contratti.
La call **before** contiene istruzioni eseguite una sola volta, prima di qualsiasi blocco it() all'interno di un contesto di test racchiuso in un blocco describe().
Viene eseguito un blocco before che recupera 4 account di prova tramite ethers.getSigners() e recupera la factory di trustToken e SplitWiseManager tramite ethers.getContractFactory, assegnandoli alle variabili precedentemente dichiarate.

Viene inizializzato tramite describe la suite di test per TrustToken. Prima di ogni test di TrustToken, viene deployata una nuova istanza di TrustToken.

La keyword **it** definisce un singolo case test, descrivendo cosa deve fare. Il primo test asserisce che la transazione deve fallire se il valore inviato è sotto il prezzo minimo. Il test si aspetta che, se chiamata mint() con un importo di TOKENPRICE-1 = 0, viene effettuato un revert con il messaggio presente nella stringa.
Il secondo test testa l'invio a mint() di un valore corretto (3 token), aspettandosi che il balance in token dell'utente sia uguale al balance coniato.

Tramite il blocco describe() vengono raccolti tutti i test legati ai metodi di SplitwiseManager. Inizialmente, vengono coniati 1000 token per ogni account.
Prima di ogni test, viene deployato TrustToken, e vengono aggiunti 1000 token per ciascun account. Viene deployato splitwisemanager, passando l'indirizzo del TrustToken. Viene infine autorizzato lo splitwisemanager a spendere da qualsiasi account qualsiasi somma di token tramite **approve().**

Viene instanziato un test che effettua **l'intero flusso** create, join, settle, simplify.
Inizialmente crea un gruppo chiamato vacanze e si aspetta che venga emesso l'evento GroupCreated con groupId = 0 e nome = "Vacanze". Carol effettua il join al gruppo 0, e il test si aspetta che venga emesso l'evento MemberJoined con parametri (0, carol).
Viene aggiunta una spesa di 30000, divisa in modo uguale da owner, alice e bob. Il test si aspetta l'emisssione dell'evento ExpenseAdded con arg (0, “Cena”, 30000).
Viene poi asserito che alice saldi 5000 token a owner e emetta l'evento DebtSettled con (0, alice, owner, 5000). Il test controlla poi il debito residuo tra alice e owner.
Il test asserisce poi che la **semplificazione** del debito porta l'emissione dell'evento DebtsSimplified, e dopo la semplificazione rimanga un debito di 5000 tra alice e owner.

Viene poi creato un test che crea un gruppo chiamato ExactTest, divide la spesa in quote esatte e verifica i nuovi bilanci degli account, e un test che effettua la stessa cosa utilizzando le percentuali di spesa.

Viene poi creata una sezione di test di Error handling dei test, dove si verificano i revert. Nel primo test owner crea il gruppo 0, ma carol che non è membro prova a semplificare, aspettandosi un errore. Il secondo test crea un gruppo, e il membro owner aggiunge una spesa pari a zero, aspettandosi un errore InvalidParameters. Il terzo test crea un gruppo, e un account non membro aggiunge una spesa diversa da zero, aspettandosi un errore InvalidParameters. Nel quarto test Test crea un gruppo includendo un membro Alice. Il membro owner aggiunge una spesa pari a 200, e Alice prova a effettuare un pagamento ad owner pari a 0, aspettandosi InvalidPayment. Viene testato poi un pagamento superiore al debito tra i due.

gasSnapshot.ts

gasSnapshot.ts è uno script utilizzato per misurare e riportare i costi del gas per le operazioni definite da SplitwiseManager e TrustToken.
Lo script inizia con una funzione **report()** di utility che riceve in input un'etichetta, la ricevuta di transazione e il prezzo del gas, e calcola l'ammontare di gas usato, il prezzo del gas e il costo di gas speso dalla funzionalità, e stampa il risultato in console.
La funzione estrae la quantità di gas consumato dalla transazione tramite r**eceipt.gas used**, calcola il costo del gas moltiplicando **gasUsed*gasPrice**, e stampa il costo in wei in formato ETH leggibile, insieme alla quantità di gas usato e il prezzo del gas.

Nel main() lo script recupera il **prezzo corrente del gas** usando il metodo **ethers.provider.send("eth_gasPrice", []),** ottiene gli account di test preconfigurati (owner, alice, bob, carol) tramite getSigners(), **deploya** il contratto TrustToken e riporta i consumi del gas per il deploy.
Effettua il Mint di 1000 token TTK per ciascun account (owner, alice, bob, carol) e stampa i consumi del gas per ogni mint.
Deploy del contratto SplitwiseManager e stampa a video dei consumi del deploy.
Approva il manager a spendere un numero illimitato di token per ogni account e stampa a video del costo del gas.
Crea un gruppo con 3 membri e stampa a video del costo del gas. Successivamente, effettua una chiamata a join(), addExpense() con split equal, addExpense() con di quote esatte, addExpense() con percentuali fissate, saldo del debito da un utente all'altro e semplificazione del debito, e per ognuna di queste operazioni, stampa a video il costo del gas associato.

```
--- Gas snapshot SplitwiseManager ---

Gas price current: 1.11 gwei

Deploy TrustToken    →gas:   679980 | price: 1.11 gwei | cost: 0.00075246178676004 ETH
mint 1000 TTK        →gas:    67486 | price: 1.11 gwei | cost: 0.000074679602549028 ETH
mint 1000 TTK        →gas:    50386 | price: 1.11 gwei | cost: 0.000055756845183228 ETH
mint 1000 TTK        →gas:    50386 | price: 1.11 gwei | cost: 0.000055756845183228 ETH
mint 1000 TTK        →gas:    50386 | price: 1.11 gwei | cost: 0.000055756845183228 ETH
Deploy Manager       →gas:  1084106 | price: 1.11 gwei | cost: 0.001199665192795788 ETH
approve              →gas:    46331 | price: 1.11 gwei | cost: 0.000051269606521338 ETH
approve              →gas:    46331 | price: 1.11 gwei | cost: 0.000051269606521338 ETH
approve              →gas:    46331 | price: 1.11 gwei | cost: 0.000051269606521338 ETH
approve              →gas:    46331 | price: 1.11 gwei | cost: 0.000051269606521338 ETH
createGroup (4)      →gas:   272570 | price: 1.11 gwei | cost: 0.00030162432603486 ETH
joinGroup            →gas:    25273 | price: 1.11 gwei | cost: 0.000027966950111454 ETH
addExpense Equal     →gas:   143498 | price: 1.11 gwei | cost: 0.000158794025525004 ETH
addExpense Exact     →gas:    59221 | price: 1.11 gwei | cost: 0.000065533603155558 ETH
addExpense Percent   →gas:    59505 | price: 1.11 gwei | cost: 0.00006584787585099 ETH
settleDebt           →gas:    59947 | price: 1.11 gwei | cost: 0.000066336990398106 ETH
simplifyDebts        →gas:    97250 | price: 1.11 gwei | cost: 0.0001076162663055 ETH
```

Analisi Vulnerabilità

Nella funzione mint() l'uso di amount = msg.value / TOKENPRICE tronca sempre per difetto, bloccando nel contratto eventuali resti decimali.
Nel codice del metodo settleDebt è presente un segmente a rischio di re-entrancy:
require(trustToken.transferFrom(msg.sender, creditor, paymentAmount), "Token transfer failed");
groupData.debtGraph[msg.sender][creditor] -= paymentAmount;
groupData.netBalance[msg.sender] += int256(paymentAmount);
groupData.netBalance[creditor] -= int256(paymentAmount);
Lo scenario possibile è il seguente. Il transferFrom invoca il contratto esterno trustToken. Se trustToken è malevolo, può eseguire un fallback che richiama **di nuovo** settleDebt **prima** che il calcolo del debtGraph abbia effettivamente ridotto il debito.
Esempio: Alla prima chiamata, registeredDebt vale, ad esempio, 100 token. Il primo transferFrom passa, ma il contratto **non ha ancora** decrementato debtGraph[msg.sender][creditor]. TrustToken chiama settleDebt di nuovo con lo **stesso** paymentAmount. Il require(paymentAmount <= registeredDebt) passa di nuovo perché registeredDebt è ancora 100. Ottieni due trasferimenti da 100 token, mentre il debito rimane invariato fino al termine della prima chiamata.
Per risolvere il problema, è stato spostato il require(transferFrom(...)) dopo l'aggiornamento del grafo del debito. Inoltre, se require(transferFrom(...)) effettua un revert, i precedenti aggiornamenti effettuati (al grafo) nella funzione vengono annullati.

Manuale d’uso

Viene dato per assunto che venga eseguito su sistema linux e che hardhat sia installato.
I comandi da utilizzare per l’utilizzo dei contract sono i seguenti, dall’interno della cartella root del progetto.

npm install //Installazione dipendenze
npx hardhat node //Su altro terminale
npx hardhat compile //Compila gli smart contract
npx hardhat run scripts/demo.ts --network localhost //Esecuzione demo
npx hardhat test //Esecuzione test
npx hardhat run scripts/gasSnapshot.ts --network localhost //Calcolo gas