Università di Pisa
Department of Computer Science
Computer Science Master Degree

TRUST PROJECT

Peer to Peer & Blockchain 24-25

Github repo

Student: Luca Rizzo

ID Number: 598992

Introduction

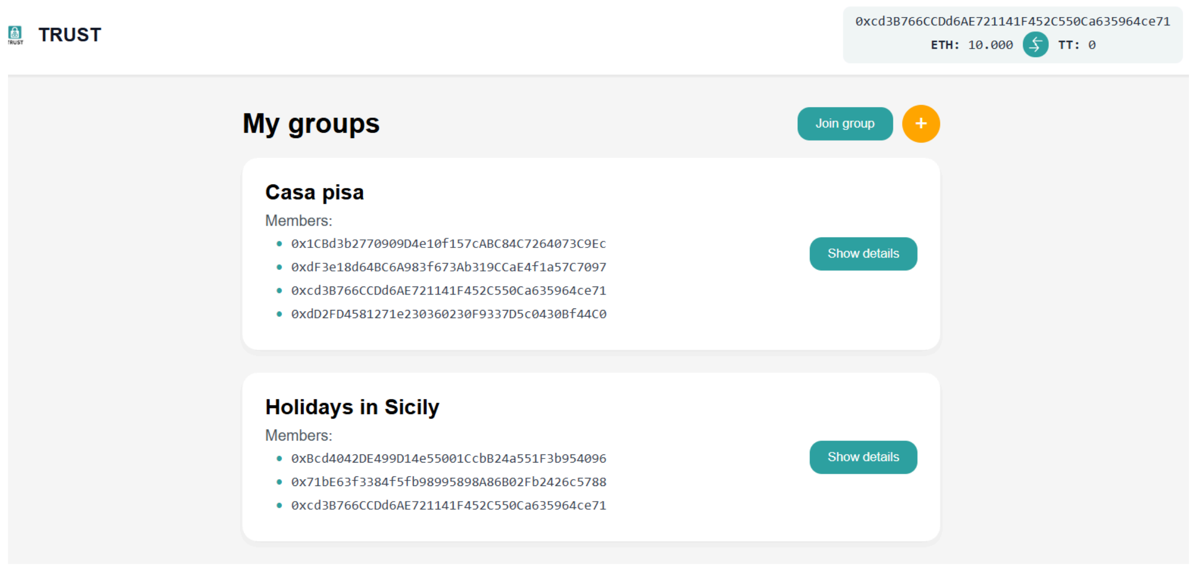
The goal of this project is to develop a decentralized application on the Ethereum blockchain that replicates and extends the core functionalities of Splitwise. The application, named **TRUST**, allows users to create groups, log shared expenses, track balances, and simplify debt relationships among participants. Unlike traditional systems, TRUST is entirely implemented through smart contracts, ensuring transparency, immutability, and automation of financial interactions.

TRUST

What sets TRUST apart is the integration of a native **fungible token**, called **Trust Token**, which is used to settle debts within the application. Conceptually, this token functions like a closed-circuit digital voucher or chip (“fish”) that users can acquire and then use to pay off shared expenses. Just like chips in a casino or tokens at a cooperative event, Trust Tokens circulate within the system: users receive them when others settle debts, and can in turn use them to settle future expenses.

This mechanism creates a self-contained economic environment where value is exchanged securely, traceably, and without relying on external payment systems.

In addition to the development of the smart contract system, a complete **frontend interface** was also implemented using Angular, a modern JavaScript framework. The frontend enables intuitive interaction with the deployed smart contracts, allowing users to manage groups, track expenses, and handle token payments through a user-friendly web interface. The frontend communicates directly with the contracts deployed on the blockchain using `ethers` library, ensuring real-time updates and direct execution of transactions without intermediaries.
In development mode, the user logs in by providing their Ethereum private key through a configuration file. In production, this step is replaced by a secure login via a wallet provider like MetaMask, which handles authentication and transaction signing directly in the browser. Once logged in, the user can view all the groups they belong to:



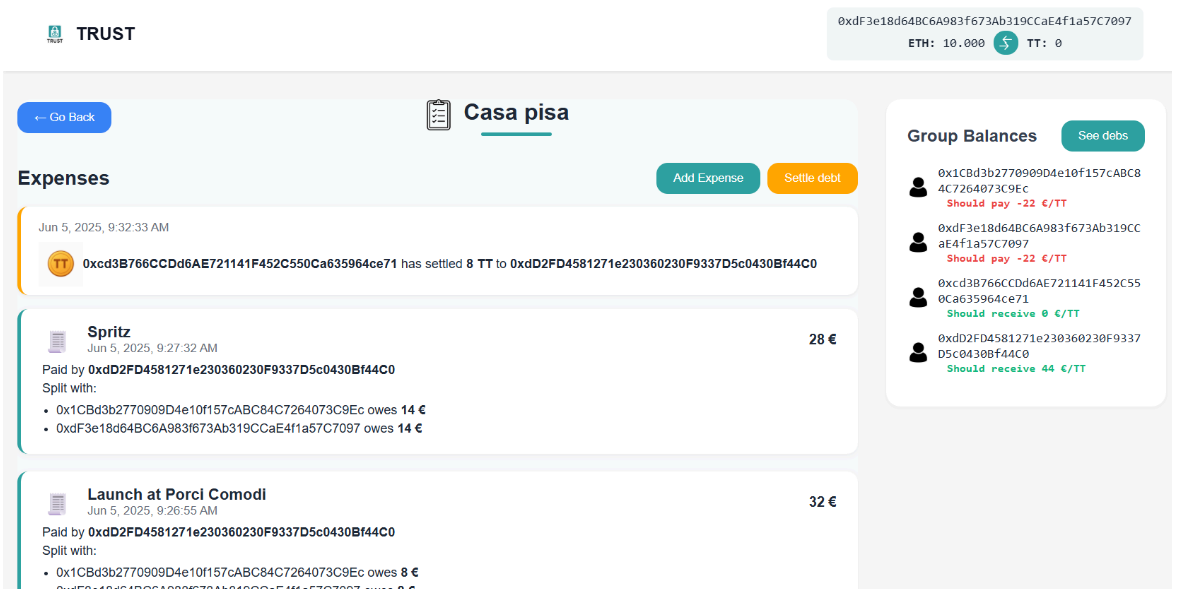
0xcd3B766CCDd6AE721141F452C550Ca635964ce71
ETH: 10.000 TT: 0

My groups

Casa pisa
Members:
- 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
- 0xcd3B766CCDd6AE721141F452C550Ca635964ce71
- 0xdD2FD4581271e230360230F9337D5c0430Bf44C0

Holidays in Sicily
Members:
- 0xBcd4042DE499D14e55001CcbB24a551F3b954096
- 0x71bE63f3384f5fb98995898A86B02Fb2426c5788
- 0xcd3B766CCDd6AE721141F452C550Ca635964ce71

By clicking on **"Show Details"**, the user can view detailed information about each group: the current debts and balances of each member, the full history of expenses (in Euros), and all the settlements (in Trust Tokens) that led to the current balances and debt situation.



0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
ETH: 10.000 TT: 0

Casa pisa

Expenses

Jun 5, 2025, 9:32:33 AM
0xcd3B766CCDd6AE721141F452C550Ca635964ce71 has settled **8 TT** to **0xdD2FD4581271e230360230F9337D5c0430Bf44C0**

Spritz — 28 €
Jun 5, 2025, 9:27:32 AM
Paid by **0xdD2FD4581271e230360230F9337D5c0430Bf44C0**
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec owes **14 €**
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097 owes **14 €**

Launch at Porci Comodi — 32 €
Jun 5, 2025, 9:26:55 AM
Paid by **0xdD2FD4581271e230360230F9337D5c0430Bf44C0**
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec owes **8 €**

Group Balances

0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
Should pay -22 €/TT

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
Should pay -22 €/TT

0xcd3B766CCDd6AE721141F452C550Ca635964ce71
Should receive 0 €/TT

0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Should receive 44 €/TT

At first glance, this mix of euro-denominated expenses and settlements in Trust Tokens (TT) might seem inconsistent. However, this design choice becomes clear in the implementation: **each TT corresponds to 1 euro**, maintaining a 1:1 ratio. Expenses are registered in euros for user familiarity, while settlements are executed on-chain using the Trust Token to ensure traceability, automation, and decentralization.
It can already be seen from the interface that the main features (such as group creation, expense registration, and debt settlement) are available. However, these will be explored in more detail in the next sections.

Implementation analysis

Architecture and Code Organization

The core logic of the TRUST application is centralized in the `TrustGroupManager.sol` contract. This contract orchestrates the entire expense management system and delegates specific functionalities to a set of supporting Solidity libraries and modules. The decision to split the application into multiple components was made to adhere to the principle of separation of concerns, ensuring that each module has a clearly defined responsibility. This modular design improves maintainability, readability, and reusability of the codebase.

Below is a brief description of the role of each module:

- `TrustGroupManager.sol`
  Acts as the main controller. It handles group creation, user registration, expense logging, debt simplification, and settlement. It delegates specific tasks to utility modules for clarity and encapsulation.

- `GroupRequestHandler.sol`
  Manages group membership logic. It handles join requests, invitation acceptance, and the logic of initialization of a group.

- `ExpenseHandler.sol`
  Handles the creation and storage of expense entries. It calculates each participant's share and updates the internal debt graph accordingly.
- `DebtSimplifier.sol`
  Implements the greedy debt simplification algorithm, reducing the number of edges in the debt graph without altering net balances.
- `DebtSettler.sol`
  Manages the actual settlement of debts using the ERC-20 Trust Token. Upon a token transfer, it updates the debt graph and confirms the payment.
- `BalanceHeap.sol`
  Implements a simple heap-based structure to assist in finding the maximum creditor and debtor efficiently during the simplification process.

Additionally, a custom ERC-20 compliant **Trust Token** contract was developed to enable on-chain debt settlement. This token leverages the standard implementation provided by **OpenZeppelin**, ensuring security and interoperability with the broader Ethereum ecosystem. The token contract allows minting (against Ether payments), transfers, and approvals, and is tightly integrated with the TRUST expense logic to update balances automatically upon settlement. Deployment of the system involves deploying **two main contracts**:

1. `TrustToken.sol` — the ERC-20 token used for settle debts
2. `TrustGroupManager.sol` — the core contract handling groups, expenses, and logic

(See the "Deployment and Module Integration" section below for more details on how these are managed using Hardhat Ignition and modular deployment scripts.)

Data structures

A fundamental component of the TRUST system is the management of groups and the relationships between users and their expenses. Each group is identified by a unique identifier and is stored in a mapping to easily access it through the id:

```
mapping(uint256 => Group) private groups;
```

Each group is represented by a `Group` struct, which stores all relevant information about its state, members, expenses, and financial relationships. Below is the full definition with inline comments for clarity:

```
struct Group {
    // Unique identifier for the group
    uint256 id;
    // Human-readable name of the group
    string name;
    // Address of the user who created the group
    address creator;
    // Set of addresses that are members of the group
    EnumerableSet.AddressSet members;
    // Pending requests from users who want to join the group
    EnumerableSet.AddressSet requestsToJoin;
```

```
        // Incremental counter for assigning unique IDs to expenses
        uint256 nextExpenseId;
        // Directed debt graph: debts[from][to] = amount owed
        mapping(address => mapping(address => uint256)) debts;
      // Net balance of each member: positive if they are owed, negative if
    they owe
        mapping(address => int256) balances;
        uint256 creationTimestamp;
    }
```

The state of a group is represented by **two core elements**: the **debt graph**, which tracks how much each participant owes to others, and the **individual balances**, which summarize the net credit or debit position of each user. These values are continuously updated as users interact with the system, primarily through the registration of new expenses and the settlement of debts.

An expense is a historical record: it logs who paid, how much, when, and for whom. These records are stored both for transparency and as a permanent reference to understand how each expense has impacted the group's financial state. By consulting the expense history, users can trace back the origin of each debt and balance adjustment, making it clear how the current situation was reached over time.

In the initial implementation, each expense recorded within a group was stored as a full `Expense` struct inside an on-chain array as a field of `Group struct`. This structure was designed to contain all the necessary information to reconstruct the expense and understand how it impacted the group's debt graph and balances. Each group maintained an `Expense[]` array, where every entry had the following format:

```
    struct Expense {
        uint256 id;
        string description;
        uint256 amount;
        address paidBy;
        ExpenseShare[] shares;
        uint256 timestamp;
    }
```

The `ExpenseShare` structure was used to specify how much of the registered expense each user had taken responsibility for and owed to the user who registered (and paid for) the expense:

```
    struct ExpenseShare {
        address participant;
        uint256 amount;
    }
```

While this approach ensured that every detail could be reconstructed directly from contract storage, it incurred a significant gas cost (especially for expenses involving many participants)making each registration potentially very expensive.

After observing that these details are primarily used for **display and historical tracking**, rather than on-chain computation, the system was refactored to emit an event `(ExpenseRegistered)` containing all relevant data, instead of storing the full `Expense` structure on chain.

To support this transition, the frontend was updated to rely entirely on event logs for retrieving expense data. When entering the details page of a group, it queries all past `ExpenseRegistered` events using `groupId` as an indexed parameter, enabling efficient reconstruction of the expense history of a group. It also subscribes in real time to new events to reflect state changes immediately, ensuring consistency between the blockchain state and the user interface without relying on stored structs.

```
event ExpenseRegistered(
  uint256 indexed groupId,
  uint256 indexed expenseId,
  address indexed payer,
  uint256 amount,
  string description,
  address[] splitWith,
  uint256[] amountForEach
);
```

The choice of data structures stored in the contracts was guided not only by functional requirements, but also by their impact on gas costs, favoring minimal and efficient storage patterns where possible (see Cost analysis section for more details).

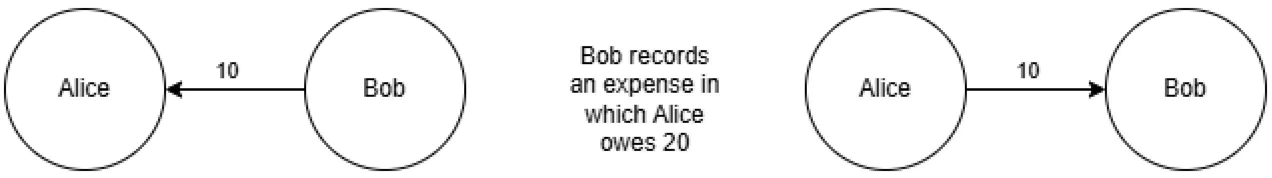
For example, with the initial on-chain approach to expense registration, where each `Expense` and its shares were fully stored in contract storage, the cost of recording a single expense could reach up to 3.98 €, a clearly unsustainable figure for real-world usage.

By switching to an event-based approach, where expense details are emitted via `ExpenseRegistered` events instead of being stored, the cost was reduced to approximately 1.89 €.

The conceptual role of an expense remains unchanged: it still affects the debt graph and individual balances by updating the internal state accordingly. It represents a financial interaction where the payer covers costs on behalf of others, generating obligations that are immediately reflected within the system. However, once this update has been applied, the expense itself no longer needs to persist in storage and is retained solely for logging and historical tracking purposes.

The financial obligation is captured in the system through a directed debt graph, which reflects who owes whom and how much.

The graph is not only directional but also **unidirectional per pair**: at any given time, a debt exists **only in one direction** between two users. For instance, if Bob owes 10 to Alice, and later Bob pays for an expense in which Alice owes him 20, the system does not store two separate debt entries. Instead, it updates the existing relationship, clearing Bob's previous debt and recording a single entry showing that Alice now owes 10 more to Bob (for a total of 20).



The debt graph is represented through a nested mapping structure:

```
mapping(address => mapping(address => uint256)) debts;
```

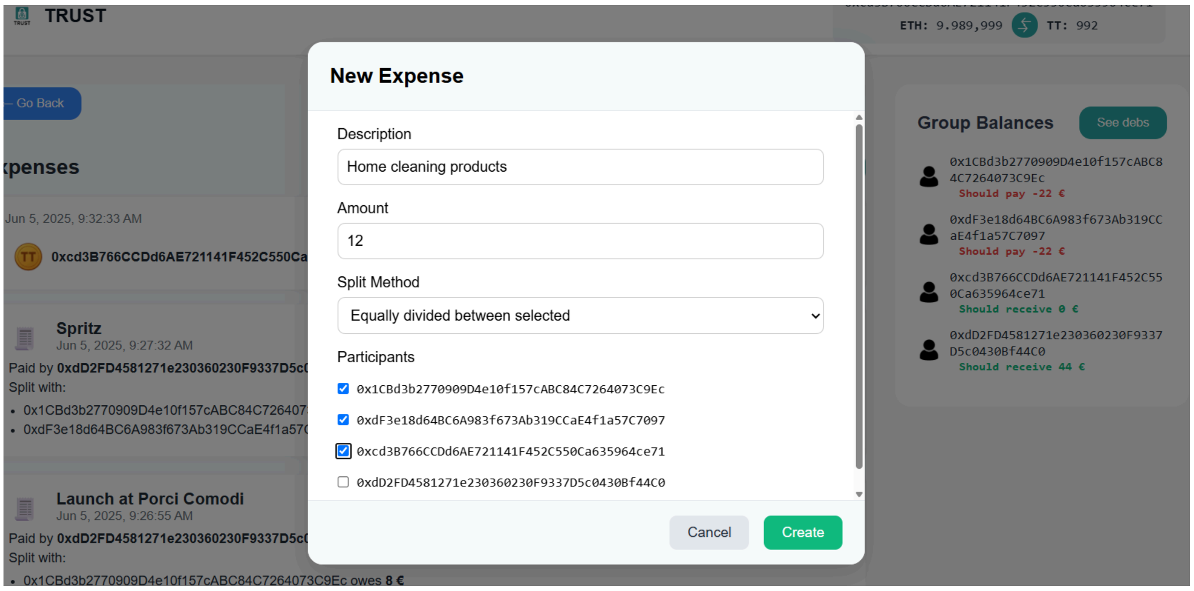
While not a true matrix, this structure conceptually behaves like a weighted adjacency matrix, where each entry `debts[from][to]` indicates how much `from` owes to `to`. It efficiently captures the directional and unidirectional nature of debts within the group, and allows fast lookup and updates as new expenses or settlements occur.

As new expenses are registered and debts are progressively settled, the debt graph and the individual balances evolve to represent the current financial obligations. To keep the structure efficient and readable, a member can trigger a simplification algorithm that reduces the number of edges in the graph without altering the net balances of any participant.

While the debt graph specifies who owes what to whom, it can often become unnecessarily complex due to indirect or circular relationships. In such cases, the net balances offer a clearer picture of each member's financial position within the group. For example, in a circular debt situation where A owes B, B owes C, and C owes A the same amount, no one effectively owes anything. The simplification algorithm detects these patterns and adjusts the debt structure accordingly, preserving correctness while improving clarity.

As previously mentioned, recording an expense causes the debt graph and individual balances to evolve, reflecting the new financial relationships created among the involved participants. Specifically, the amount paid is distributed among the selected users according to the chosen split method, and this distribution directly impacts both the debt graph and the net balances of the group members.

During expense registration, the payer can indicate which participants are involved and how the total amount should be split.



ETH: 9.989,999 TT: 992

Jun 5, 2025, 9:32:33 AM

0xcd3B766CCDd6AE721141F452C550Ca

Spritz
Jun 5, 2025, 9:27:32 AM
Paid by **0xdD2FD4581271e230360230F9337D5c0**
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C726407
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C

Launch at Porci Comodi
Jun 5, 2025, 9:26:55 AM
Paid by **0xdD2FD4581271e230360230F9337D5c0**
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec owes **8 €**

New Expense

Description: Home cleaning products

Amount: 12

Split Method: Equally divided between selected

Participants
- [✓] 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
- [✓] 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
- [✓] 0xcd3B766CCDd6AE721141F452C550Ca635964ce71
- [ ] 0xdD2FD4581271e230360230F9337D5c0430Bf44C0

Group Balances

0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
Should pay -22 €

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
Should pay -22 €

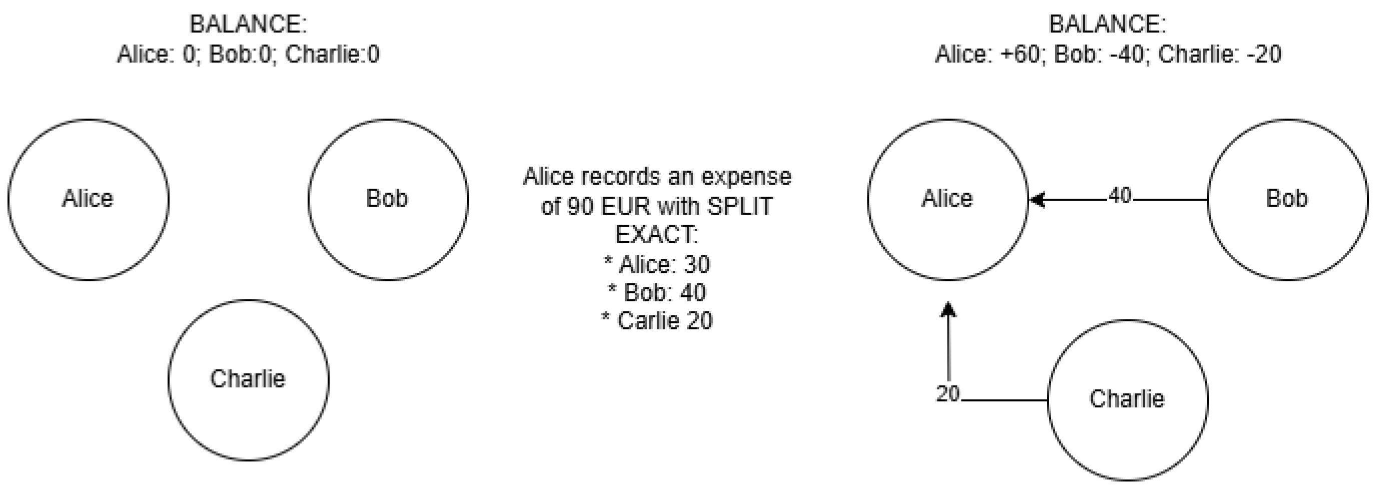
0xcd3B766CCDd6AE721141F452C550Ca635964ce71
Should receive 0 €

0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Should receive 44 €

The system supports different splitting strategies, such as:

- Equal split, where the amount is divided evenly among all selected participants
- Exact amounts, where the user manually assigns a specific value to each participant
- Percentage-based, where each share is calculated as a percentage of the total amount

This flexible input allows the system to accurately capture how the expense is distributed among participants. The recorded distribution is then used to update both the debt graph and the individual balances accordingly. Obviously **only the amounts lent to others are recorded as debts**. The share of the payer themselves is not included in the debt graph, as it does not represent an obligation.
For example, suppose Alice pays 90 EUR for an expense and chooses to split it using the EXACT method: Bob is assigned 40 EUR, Charlie 20 EUR, and Alice herself 30 EUR. In this case, the system will record that Alice has lent 40 EUR to Bob and 20 EUR to Charlie**.** Her own share of 30 EUR is not recorded in the debt graph or balances, since no one owes themselves money.


BALANCE:
Alice: 0; Bob:0; Charlie:0
Alice
Bob
Charlie
Alice records an expense of 90 EUR with SPLIT EXACT:
* Alice: 30
* Bob: 40
* Carlie 20
BALANCE:
Alice: +60; Bob: -40; Charlie: -20
Alice
40
Bob
20
Charlie

What matters for the balance and debts graph is not the total amount paid, but the portion that was covered on behalf of others. Debts are created **only** for the shares of the expenses that were paid in advance for other users.

The same strategy used for expenses is adopted for settlements: rather than storing them in a persistent `struct`, all relevant information is emitted through events. This allows the system to remain fully transparent while minimizing gas costs and on-chain storage. The contract emits a `DebtSettled` event, which is picked up by the frontend.

```
event DebtSettled(
    uint256 indexed groupId,
    address indexed from,
    address indexed to,
    uint256 amount
);
```

All **balance**, **debt, expense and settlement amount** values in the contract are stored as **18-decimal fixed-point numbers**, following the common practice in Ethereum of representing currency values using `uint256` with 18 decimal places. This approach ensures high precision in all financial computations and avoids rounding errors that could arise from floating-point arithmetic.

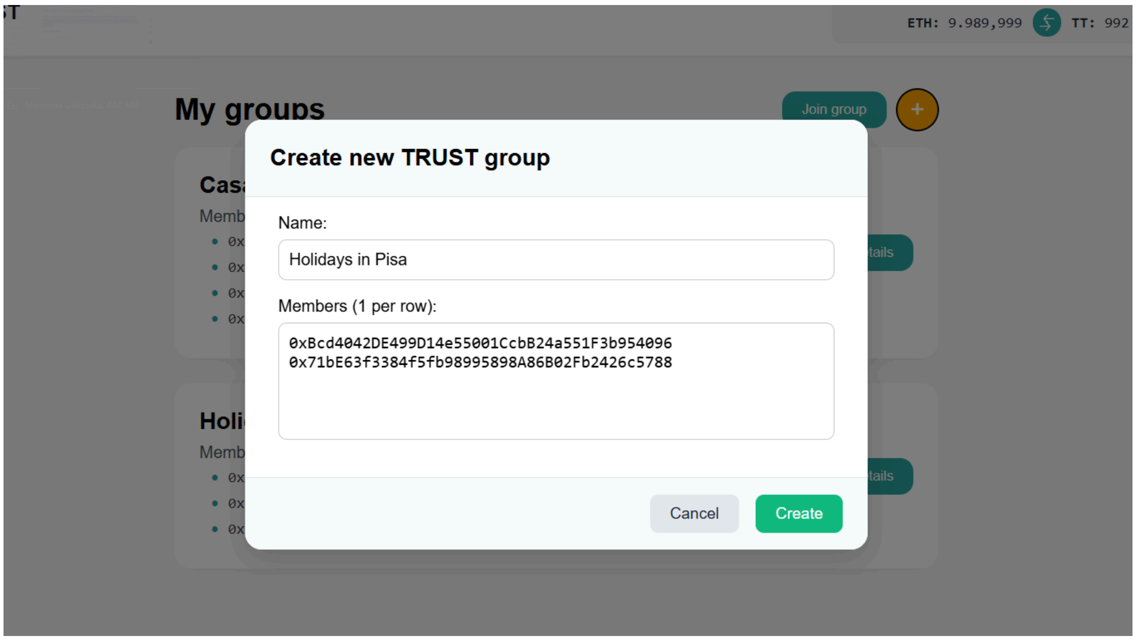
Although the frontend displays amounts using euros as the unit for user clarity, from the contract's perspective these values are simply quantities expressed in base-18 format. The choice of "euro" is purely conventional and visual; the underlying logic remains unit-agnostic and could support any currency or token denomination, as long as formatting is handled consistently at the interface level.

Functionalities of TrustGroupManager

The `TrustGroupManager` contract is the central logic layer of the TRUST application. It manages everything from group creation to expense tracking, debt settlement, and simplification. Below is a description of its main functionalities, highlighting the purpose of each method, the meaning of its parameters, and how each operation is triggered from the frontend interface. For every method described, a corresponding user interaction is implemented in the UI, allowing users to seamlessly invoke contract functions through intuitive actions.

Group Creation

```
function createGroup(string calldata name, address[] calldata otherMembers)
external returns (uint256);
```



Create new TRUST group

Name:

Holidays in Pisa

Members (1 per row):

```
0xBcd4042DE499D14e55001CcbB24a551F3b954096
0x71bE63f3384f5fb98995898A86B02Fb2426c5788
```

Cancel Create

To create a new group, a user calls the **`createGroup`** function, providing a human-readable `name` and an array of Ethereum addresses, `otherMembers`, representing the participants. The caller is automatically added to the group and marked as the creator. Internally, the group is assigned a unique ID and stored in the contract's `groups` mapping. The function emits a `UserApproved` event for each approved member, allowing the frontend to dynamically reconstruct all groups associated with a given user by indexing these events. This design ensures efficient retrieval without requiring additional on-chain storage.

Expense Registration

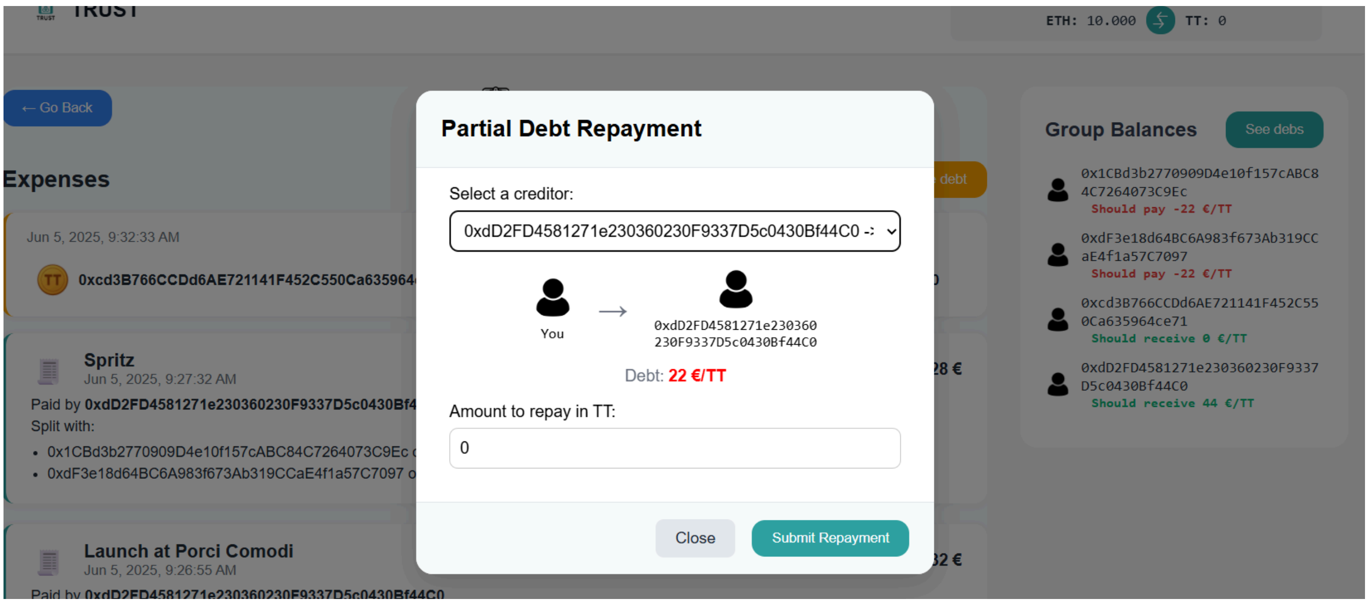
```
function registerExpenses(uint256 group_id, string calldata description, uint256
amount, address[] calldata splitWith, SplitMethod splitMethod, uint256[]
calldata values) external;
```

To register a shared expense, the `registerExpenses` function is used. The caller provides the `groupId` to which the expense belongs, a textual `description`, the `amount` of the expense (with 18 decimals precision), an array of `splitWith` participant addresses, and a corresponding `values` array, significant in the `SplitMethod.EXACT` as exact amounts or as percentage shares in the case of the `SplitMethod.PERCENTAGE`

The system treats this action as the caller having paid the full `amount` on behalf of the others. It calculates the difference between what each participant should pay and what they actually paid (usually zero), and updates the internal **debt graph** `(debts[from][to])` and **balances** accordingly. An `ExpenseRegistered` event is emitted for frontend tracking.

Debt Settlement

```
function settleDebt(uint256 group_id, uint256 amount, address to) external;
```



Jun 5, 2025, 9:32:33 AM
0xcd3B766CCDd6AE721141F452C550Ca635964

Spritz
Jun 5, 2025, 9:27:32 AM
Paid by 0xdD2FD4581271e230360230F9337D5c0430Bf4
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097

Launch at Porci Comodi
Jun 5, 2025, 9:26:55 AM
Paid by 0xdD2FD4581271e230360230F9337D5c0430Bf44C0

28 €
32 €

Partial Debt Repayment

Select a creditor: 0xdD2FD4581271e230360230F9337D5c0430Bf44C0

You → 0xdD2FD4581271e230360230F9337D5c0430Bf44C0

Debt: 22 €/TT

Amount to repay in TT: 0

Close | Submit Repayment

0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
Should pay -22 €/TT

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
Should pay -22 €/TT

0xcd3B766CCDd6AE721141F452C550Ca635964ce71
Should receive 0 €/TT

0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Should receive 44 €/TT

When a user wants to pay off their debt to another group member, they call `settleDebt`, specifying the `group_id`, the recipient's address (`to`), and the `amount` to be settled. The function first checks whether the sender has a sufficient token balance using `balanceOf`, and then verifies that the TrustGroupManager contract has been granted enough allowance through allowance. This requires the user to have previously approved the contract to spend tokens on their behalf by calling approve on the TrustToken contract. Once these checks pass, the function internally calls `transferFrom` to move the specified amount of tokens from the payer to the recipient. It then reduces the value in `debts[msg.sender][to]` by the settled amount and updates the balances of both users.

Although the function interacts with a trusted external account (the TrustToken contract) via `transferFrom`, it still follows the standard checks-effects-interactions pattern. All checks are performed first, then the internal state (debts and balances) is updated, and only at the end the external call to the token contract is executed, in order to avoid **reentrancy attacks** and ensure the consistency of the system's state.

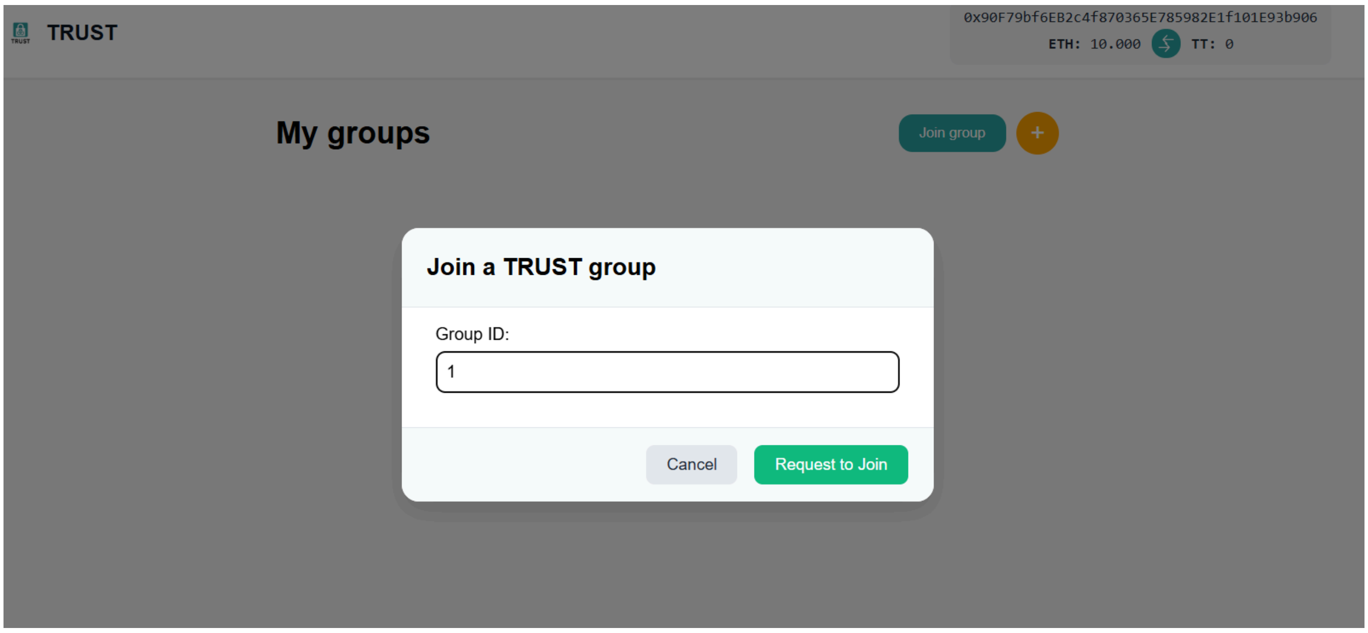
No persistent data structure is updated to store the settlement itself. Instead, a `DebtSettled` event is emitted and indexed by group ID so the frontend can process it efficiently.

Group Join Flow

- `function requestToJoin(uint256 group_id) external`
- `function approveAddress(uint256 group_id, address userToApprove) external`

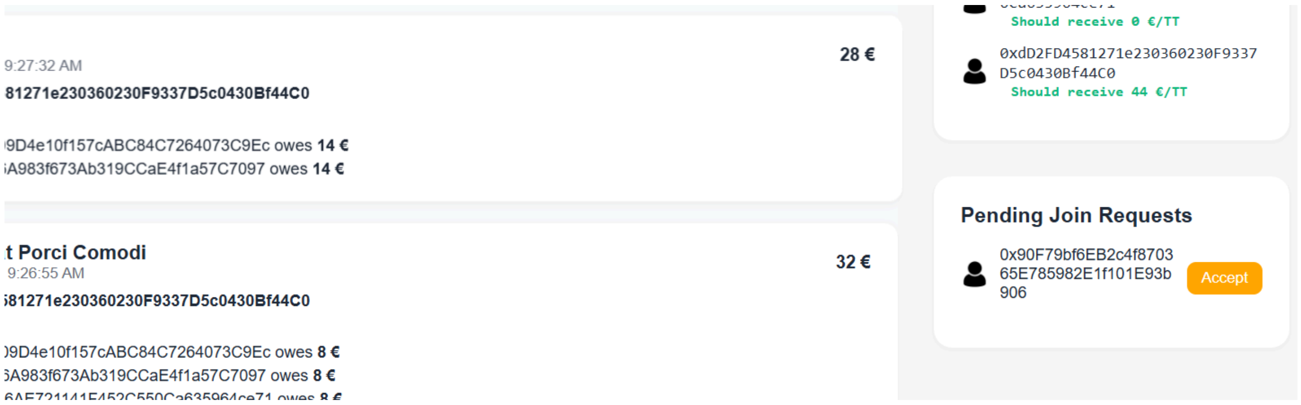
To prevent unauthorized access, the system adopts a **two-step group join mechanism** instead of allowing open enrollment. This was a deliberate design choice to improve **security and control**: only users explicitly approved by an existing group member (typically the creator) can join a group.

When a user wants to join a group, they must first call `requestToJoin`, specifying the `groupId`. This adds their address to the group's `requestsToJoin` set, indicating a pending request.



Group ID: 1

An existing group member can then call `approveRequestToJoin`, passing the `groupId` and the address of the requester.



9:27:32 AM
81271e230360230F9337D5c0430Bf44C0
9D4e10f157cABC84C7264073C9Ec owes **14 €**
A983f673Ab319CCaE4f1a57C7097 owes **14 €**
28 €

t Porci Comodi
9:26:55 AM
81271e230360230F9337D5c0430Bf44C0
9D4e10f157cABC84C7264073C9Ec owes **8 €**
A983f673Ab319CCaE4f1a57C7097 owes **8 €**
32 €

Should receive 0 €/TT
0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Should receive 44 €/TT

0x90F79bf6EB2c4f870365E785982E1f101E93b906

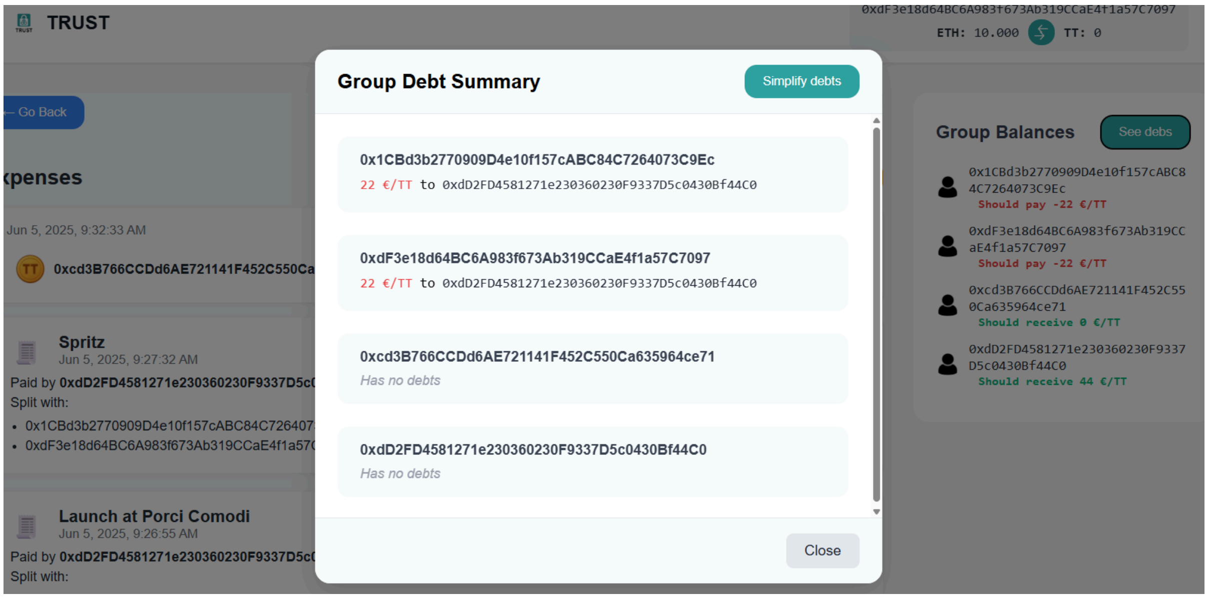
Upon approval:

- The requester is added to the `members` set of the group.
- The group ID is added to the requester's `groupsOfAddress` list
- The request is removed from the pending list.

This mechanism ensures that group membership remains under control and prevents spam or malicious users from joining groups without consent. It also allows the frontend to display pending join requests and manage invitations cleanly through user interaction.

Debt Simplification

```
function simplifyDebt(uint256 groupId) external;
```



Group Debt Summary

0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
22 €/TT to 0xdD2FD4581271e230360230F9337D5c0430Bf44C0

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
22 €/TT to 0xdD2FD4581271e230360230F9337D5c0430Bf44C0

0xcd3B766CCDd6AE721141F452C550Ca635964ce71
Has no debts

0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Has no debts

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
ETH: 10.000 TT: 0

Jun 5, 2025, 9:32:33 AM
0xcd3B766CCDd6AE721141F452C550Ca

Spritz
Jun 5, 2025, 9:27:32 AM
Paid by 0xdD2FD4581271e230360230F9337D5c0
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C726407
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57

Launch at Porci Comodi
Jun 5, 2025, 9:26:55 AM
Paid by 0xdD2FD4581271e230360230F9337D5c0
Split with:

Group Balances

0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
Should pay -22 €/TT

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
Should pay -22 €/TT

0xcd3B766CCDd6AE721141F452C550Ca635964ce71
Should receive 0 €/TT

0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Should receive 44 €/TT

Over time, the debt graph may become dense or contain circular dependencies. To simplify this, any member of a group can call `simplifyDebts`, providing the groupId. The function first computes the net balance of each group member (what they are owed minus what they owe), then applies a greedy algorithm to reduce the number of edges in the graph.

Before starting the simplification process, the function clears all existing debt edges by setting every `debts[from][to]` value to 0. This ensures that the new simplified graph is built from scratch.

The algorithm then proceeds by taking the already-stored net balances of each group member (previously maintained during expense registration and settlements). These balances are used to identify who owes and who is owed, without needing to recompute them. All users with a positive balance (creditors) are inserted into a max-heap, while users with a negative balance (debtors) are inserted into a min-heap. These heaps, implemented as simple array-based structures `(BalanceHeap)`, allow for efficient retrieval of the users with the highest credit and highest debt.

At each iteration, the algorithm extracts the top user from both heaps, the one who is owed the most and the one who owes the most, and creates a new debt link between them. The transferred amount is the minimum between their respective balances. If either user still has an outstanding amount after the transfer, they are reinserted into the heap for further processing.

This continues until both heaps are empty. The result is a simplified debt graph, with a greedy minimal number of edges and the original balances fully preserved, reducing the number of transactions needed for final settlement.

Read Functions (View Helpers)

To support the frontend and allow users to interactively explore group data, several read-only functions are provided:

- `function retrieveGroup(uint256 group_id) external view returns (GroupDetailsView memory)`: returns all the details associated with the group.

- `function allGroupDebts(uint256 group_id) external view returns (DebtNode[] memory)`: returns the current amount owed from one user to another.

These functions do not modify the blockchain state and are used by the frontend to display updated data after actions or when a user enters a group view. All returned structs are frontend-oriented representations of the actual on-chain data. For example, the debt graph cannot be returned as a nested mapping, since Solidity does not support returning mappings from public or external functions. Instead, it is exported as an array of `DebtNode[]`, where each `DebtNode` represents a user and contains a list of outgoing debt edges, effectively modeling the debt graph as an adjacency list. This structure is optimized for readability and integration with the frontend.

Functionalities of TrustToken

The Trust Token (TT) acts as a stable internal currency for the TRUST system. One TT symbolically represents 1 EUR, and is used exclusively to settle debts between users within the system. Users can receive TT from others (when they are paid back) and reuse them to settle future debts. This mechanism creates a closed-loop payment system, where tokens continuously circulate among participants.

The system is closed by design: ETH is used only to mint tokens, and there is no `withdraw` function. All ETH sent to the contract remains locked inside it. This guarantees that the TT maintains a stable symbolic value (1 TT = 1 EUR), avoids market speculation, and ensures that the token is used exclusively for its intended purpose: peer-to-peer expense settlement.

The fallback() and receive() functions of the TrustToken contract are both overridden to call buyTokens(). This ensures that any ETH sent directly to the contract (intentionally or by mistake) is always interpreted as a token purchase, using the same current exchange rate defined in the contract.

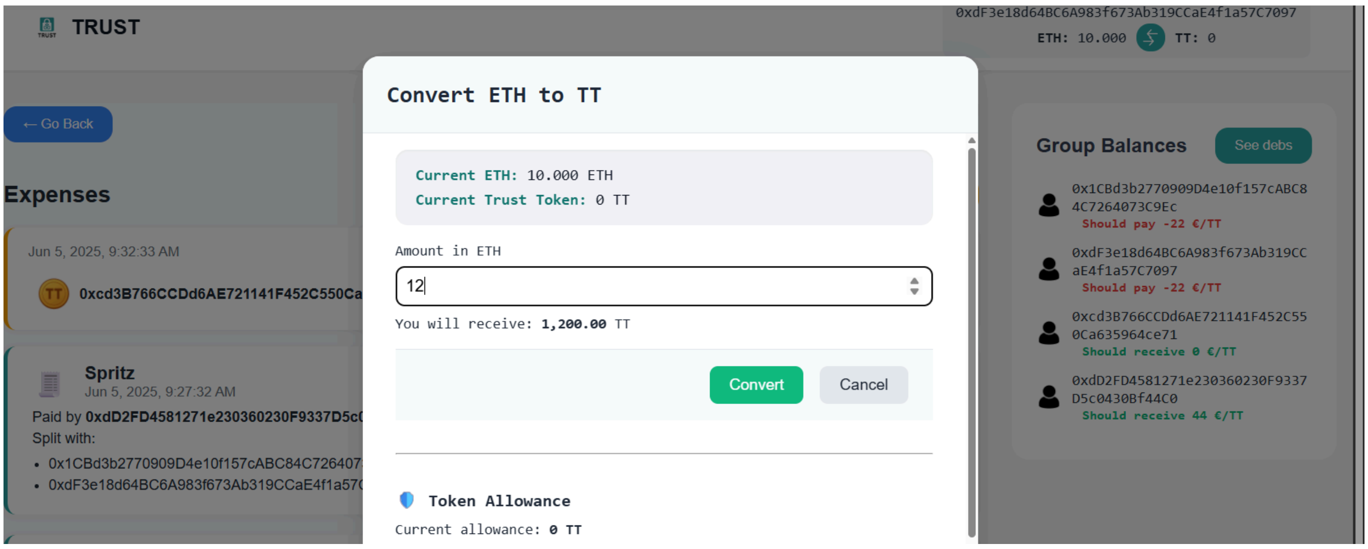
Why no `withdraw`?

Allowing users to withdraw ETH in exchange for Trust Tokens would break the internal logic of the system. The TT are meant to act solely as an **internal utility token**, used only for settling debts within the app. ETH is used exclusively to enable access to the system: it provides the initial weight to the tokens, like exchanging real currency for chips in a closed environment. If withdrawal were allowed, this would open the door to speculative behavior and external price manipulation, which would undermine the very purpose of the token.

For example, if ETH gains or loses significant value over time, settling a debt today versus tomorrow could result in different real-world values, distorting fairness. By **locking ETH within the system** and preventing withdrawal, we **maintain price neutrality and isolation from external financial dynamics**. If the ETH price changes substantially, the system administrator can manually adjust the exchange rate, ensuring that the symbolic equivalence **1 TT ≈ 1 EUR** is preserved. In this way, ETH remains only a gateway mechanism and not as a tradable or speculative asset inside the TRUST economy.

Buy token

```
function buyTokens() public payable;
```


TRUST

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
ETH: 10.000 TT: 0

← Go Back

Expenses

Jun 5, 2025, 9:32:33 AM
0xcd3B766CCDd6AE721141F452C550Ca

Spritz
Jun 5, 2025, 9:27:32 AM
Paid by 0xdD2FD4581271e230360230F9337D5c0
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C726407
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57

Convert ETH to TT

Current ETH: 10.000 ETH
Current Trust Token: 0 TT

Amount in ETH
12
You will receive: 1,200.00 TT

Convert Cancel

Token Allowance
Current allowance: 0 TT

Group Balances See debs

0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec
Should pay -22 €/TT

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
Should pay -22 €/TT

0xcd3B766CCDd6AE721141F452C550Ca635964ce71
Should receive 0 €/TT

0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Should receive 44 €/TT

Minting is the only entry point to obtain TT. Users can acquire tokens by calling the `buyTokens()` function and sending ETH along with the transaction. The number of TT minted is calculated based on the current rate, which defines how many tokens are issued per unit of ETH. The ETH sent is permanently locked in the contract and cannot be withdrawn.

The rate is manually adjustable by the contract owner via the `updateRate(uint256 newRate)` function. This mechanism ensures that TT maintains a symbolic and stable internal value (conceptually 1 TT = 1 EUR). If the value of ETH increases significantly, the owner can increase the rate so that users receive more TT per ETH, preserving the original idea that ETH is only used to access the system and assign initial weight to the tokens, not to influence their value through speculation.

The `TrustToken` contract implements all the standard ERC-20 functions deriving from the implementation of the OpenZeppelin library:

- `function totalSupply() external view returns (uint256);`
  Returns the total number of tokens in circulation.

- `function balanceOf(address account) external view returns (uint256);`
  Returns the balance of tokens held by the specified address.

- `function transfer(address to, uint256 amount) external returns (bool);`
  Transfers `amount` tokens from the caller to the `to` address.

- `function allowance(address owner, address spender) external view returns (uint256);`
  Returns the remaining number of tokens that `spender` is allowed to spend on behalf of `owner`.

- `function approve(address spender, uint256 amount) external returns (bool);`
  Authorizes `spender` to transfer up to `amount` tokens on behalf of the caller.

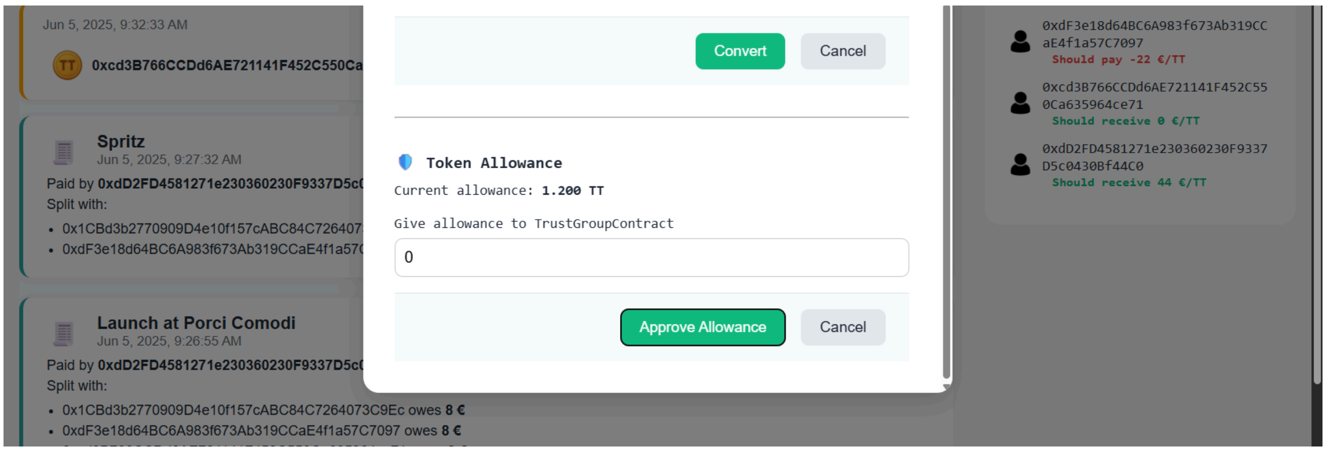
- `function transferFrom(address from, address to, uint256 amount) external returns (bool);`
  Transfers `amount` tokens from `from` to `to`, provided the caller has sufficient allowance

These functions are used directly or indirectly within the system, especially when settling debts.

One of the most important functions is `balanceOf`, which allows the contract to check how many tokens a user owns. This is essential, for example, during a `settleDebt` call, where the system must ensure that the payer has enough TT to cover the amount they wish to settle.

Closely related is the `allowance` function, which returns how many tokens the user has authorized the contract to spend on their behalf. Before any transfer can happen using `transferFrom`, the contract uses `allowance` to verify that it has sufficient authorization.

To grant this permission, the user must first call `approve`, which explicitly allows the `TrustGroupManager` contract to move a certain amount of tokens on their behalf. This step is typically performed from the front end before settling a debt.


Jun 5, 2025, 9:32:33 AM
0xcd3B766CCDd6AE721141F452C550Ca

Spritz
Jun 5, 2025, 9:27:32 AM
Paid by **0xdD2FD4581271e230360230F9337D5c0**
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C726407
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57

Launch at Porci Comodi
Jun 5, 2025, 9:26:55 AM
Paid by **0xdD2FD4581271e230360230F9337D5c0**
Split with:
- 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec owes **8 €**
- 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097 owes **8 €**

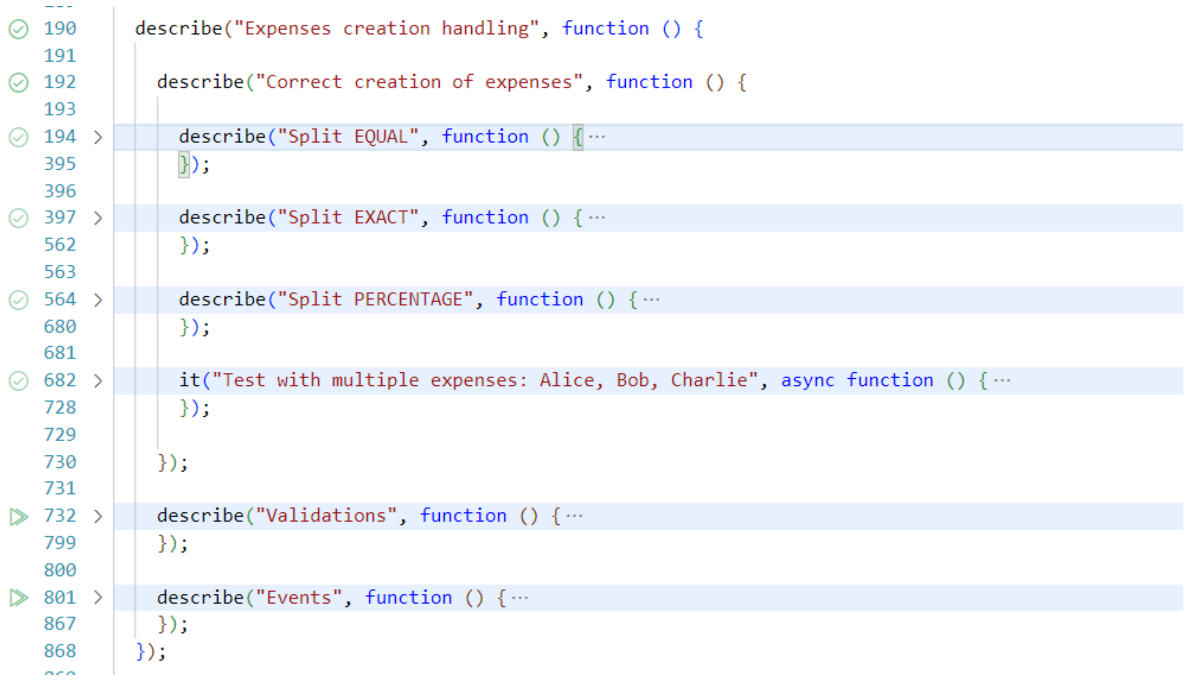
Token Allowance
Current allowance: **1.200 TT**
Give allowance to TrustGroupContract
0

0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097
Should pay -22 €/TT
0xcd3B766CCDd6AE721141F452C550Ca635964ce71
Should receive 0 €/TT
0xdD2FD4581271e230360230F9337D5c0430Bf44C0
Should receive 44 €/TT

The actual transfer of tokens is performed using `transferFrom`, which deducts the specified amount from the payer and sends it to the recipient. This function is at the core of the `settleDebt` process and enables trustless transfers directly between users under the control of the contract.

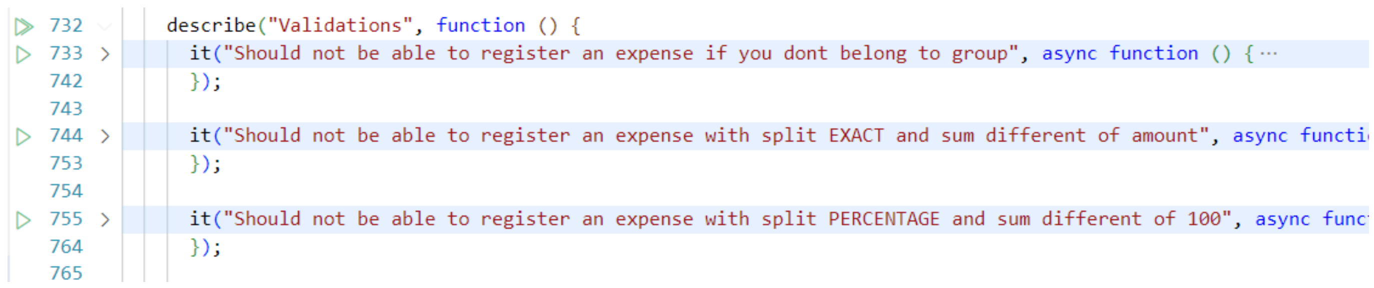
Functional tests

Extensive testing was carried out, primarily within the `TrustGroupManager.ts` file. The main objective was to achieve full coverage of all critical logic and corner cases, ensuring the system behaves correctly under both normal and edge conditions. Tests were written to validate all the core functionalities, including group creation, expense registration, debt settlement, and simplification. Special attention was given to scenarios involving unexpected inputs or failure paths, such as unauthorized access or invalid request paths. These tests help ensure that the contract does not just behave correctly when everything is well-formed, but also when users attempt to interact with it incorrectly or maliciously.
As an example of the testing depth, the logic for expense registration alone is covered through several dedicated test groups. In the file TrustGroupManager.test.ts, we can find specific blocks for each supported split mode: Split EQUAL, Split EXACT, and Split PERCENTAGE. Each of these groups contains multiple tests that verify the correct distribution of amounts, internal debt updates, and edge case behavior.



```
describe("Expenses creation handling", function () {

  describe("Correct creation of expenses", function () {

    describe("Split EQUAL", function () {⋯
    });

    describe("Split EXACT", function () {⋯
    });

    describe("Split PERCENTAGE", function () {⋯
    });

    it("Test with multiple expenses: Alice, Bob, Charlie", async function () {⋯
    });

  });

  describe("Validations", function () {⋯
  });

  describe("Events", function () {⋯
  });
});
```

In addition to the positive test cases, there is also a section labeled Validations, which focuses entirely on error handling and rejection scenarios. These tests ensure that invalid inputs or unauthorized operations are correctly rejected by the contract. Examples include checks such as:



```
  describe("Validations", function () {
    it("Should not be able to register an expense if you dont belong to group", async function () {⋯
    });

    it("Should not be able to register an expense with split EXACT and sum different of amount", async functi
    });

    it("Should not be able to register an expense with split PERCENTAGE and sum different of 100", async func
    });

```

Finally, the Events group verifies that the correct events are emitted during the registration process, ensuring that the frontend can rely on them for real-time updates.

This rigorous testing is particularly important in the context of blockchain development, where once a contract is deployed, its logic is immutable. Unlike traditional software systems that can be patched or updated, a smart contract must be considered final at the moment of deployment. For this reason, testing is not just a best practice but it is an absolute necessity to ensure the correctness and reliability of the system. By simulating both positive and negative cases, the test suite aims to anticipate real-world usage and misuse, significantly reducing the risk of unexpected behavior after deployment, which would require a costly and disruptive **redeployment of the entire contract**. To run the tests, it is sufficient to navigate to the `hardhat` folder and execute the command:

```
npm run functional-test
```

This script is defined in the `package.json` file and internally runs:

```
npx hardhat test ./test/TrustGroupManager.ts
```

Contracts deployment

The deployment process is managed using **Hardhat Ignition**, a modular deployment system provided by the Hardhat framework. In TRUST, deployment is handled through a custom TypeScript module located at: `ignition/modules/TrustModule.ts`

This module defines and deploys the two core contracts: `TrustToken` is deployed first, followed by `TrustGroupManager`, which receives the address of the token contract through its constructor. This ensures that the manager contract is correctly initialized with a valid reference to the token it will interact with for debt settlement. By explicitly deploying the token first and passing its address at construction time, we guarantee consistency and correctness in the setup.

Before running the application locally, make sure to install all dependencies by running inside the `hardhat` folder: `npm install`

To deploy the contracts, a running blockchain network is required. When working **in local**, this can be achieved by starting a local **Hardhat node**. The typical local deployment workflow is as follows:

1. Compile the contracts and start the local node:

   ```
   npm run compile-and-start
   ```

2. In a separate terminal, deploy the contracts using the TrustModule:

   ```
   npm run local-deploy
   ```

Alternatively, you can use the `npm run deploy-all-local` script, which does everything for you: it compiles the contracts, starts the local Hardhat node in the background and then automatically triggers the deployment via Ignition. This allows for a full local deployment with a single command, and when you close the terminal, both the node and the deployment process are terminated together.

Run frontend application

Once the contracts are deployed, you can move inside the `frontend` folder and run: `npm install` to install the frontend dependencies. Then, the application can be launched with: `npm run start`.

This will serve the application in development mode on `localhost:4200`, connecting to the contracts deployed on `localhost:8545`. The frontend communicates directly with the contracts using `ethers.js`, enabling group creation, expense tracking, and token-based settlements.

Costs analysis

A core objective throughout the development was to minimize gas costs in order to make the application practically usable. While storing all data on-chain simplifies logic and guarantees full traceability, it comes at the cost of significantly higher transaction fees, which can quickly become prohibitive in real-world scenarios. For this reason, special attention was given to **minimizing storage writes**, such as push() calls on arrays or updates to mappings, which are among the most expensive operations in the EVM. Reducing these writes wherever possible has a direct and measurable impact on lowering gas consumption. Several critical optimizations were introduced during development to reduce gas consumption, for example:

- Expense registration:
  - Before: storing full expense data (including participant shares) directly in storage.
    - Cost with 8 participants: ~4 €
  - After: switched to emitting a lightweight ExpenseRegistered event
    - New cost: ~1.89 €

  Same transparency but much cheaper expense registration

- Group creation:
  - Before: for each user, storing group membership in a `mapping(address => uint256[])`, requiring:
    - 1 new storage slot (≈ 20,000 gas) per member
    - 1 push() per member in the mapped dynamic array
    - Cost with 20 participants: 7.64 €
  - After: removed this mapping entirely, relying on events and inference from group state
    - Cost with 20 participants: 4.34 €

  Same traceability but much cheaper group creation

Gas usage in the TRUST system is evaluated using a custom test script located at `/test/GasUsage.ts`. This script automatically performs sequences of operations, such as group creation, expense registration, debt simplification, and settlement, while systematically varying key parameters.

The script logs the gas used and estimates the cost in USD using the average Ethereum gas price and the ETH/USD rate as of June 6, 2025.

Although the project includes the Hardhat Gas Reporter in `gas-report.txt` file, which provides a useful summary of gas usage per function, it does not reflect how gas consumption changes with different input scenarios. For this reason, the analysis presented here is based on the logs produced by `GasUsage.ts`, enabling a detailed study of the scalability and cost evolution of each operation under realistic and variable conditions.

Results are presented in the form of comparative tables and plots in the following sections.

Create group

In the group creation operation, the primary parameter affecting gas consumption is the number of participants. As the group size increases, the gas usage grows accordingly. This is expected, since each additional member requires more storage and computational steps within the smart contract.

members	gasUsed	estimated Cost EUR
2	304241	1.17€
4	353618	1.36€
8	546419	2.11€
20	1124811	4.34€

Settle debt

In the debt settlement operation, gas usage does not depend on the value being settled. Even when transferring extremely large amounts, the gas cost remains nearly constant. This is because the operation only updates the internal state for a single debt edge and performs a token transfer, both of which are unaffected by the specific amount involved.

amount settled	gasUsed	estimated Cost EUR
90	80029	0.31€
120	80029	0.31€
120	80017	0.31€
1000000	80041	0.31€

Register expense

In the expense registration operation, the key parameter influencing gas consumption is the number of participants involved in the expense. As this number increases, the gas cost grows very slow. This behavior is due to the fact that the contract must update the internal debt graph and the balances for all involved users. Moreover, the emitted event becomes increasingly large as the number of participants grows, since it must record all addresses involved in the expense along with their respective shares.

# members	gasUsed	estimated Cost EUR
2	183200	1.11€
4	285111	1.10€
8	488933	2.29€

Simplify debts

In the debt simplification operation, gas consumption primarily depends on the number of participants. The most expensive part of the process is clearing the debt graph, which involves iterating over the entire "debts matrix" (a nested mapping) to reset values. This operation cannot be performed directly in Solidity, as nested mappings cannot be cleared in a single instruction and require explicit iteration. As a result, the gas cost scales with the number of users, even if many entries are zero. A possible

optimization would be to maintain an explicit list of active debt edges (i.e., who owes whom). This would allow the simplification algorithm to skip zero entries and operate in time proportional to the number of actual edges, rather than the square of the number of participants. However, this change would increase the gas cost of other operations due to the need to manage and update that list consistently.
Overall, for common usage scenarios involving up to 8 participants, the cost of simplification remains acceptable and does not present a scalability issue.

description	# members	gasUsed	estimated Cost EUR
Single edge	2	96360	0.37 €
Multiple in-edges	3	102431	0.39 €
Multiple out-edges	3	108968	0.42 €
Cycle (3 nodes, one zero)	3	95160	0.37 €
Cycle (4 nodes, all zero)	4	82944	0.32 €
PDF example	4	129189	0.5 €
Complex case (20 members, cyclic)	20	1328498	5.12 €
All-to-all debts (20 members)	20	1615972	6.23 €
Random debts (8 members)	8	238761	0.92 €

Request to join and approve

For “join” operations, gas usage is fixed and does not vary with any input parameters, as each action (request and approval) involves a single address and does not depend on group size.

operation	gasUsed	estimated Cost EUR
Request to Join	95361	0.37€
Approve Join Request	80433	0.31€

Vulnerabilities analysis

Several common classes of vulnerabilities were considered during development, and specific strategies were adopted to mitigate or eliminate them. This section presents a review of those vulnerabilities and the reasoning behind the chosen defense mechanisms. While no system can claim absolute security, the precautions taken, including rigorous testing, inspire confidence in TRUST’s robustness against known vulnerabilities.

Reentrancy

In TRUST, the only external call occurs during debt settlement, when the system invokes the transferFrom function on the TrustToken contract. Although this token contract is a trusted component because it is deployed by the same system and passed directly via the constructor, the Checks-Effects-Interactions pattern is still strictly followed.

All validations and internal state updates (debts, balances) are performed before the external call to `transferFrom`. This is not strictly necessary in this case due to the trusted nature of the token, but it is maintained deliberately as a matter of conceptual correctness and best practice in smart contract design.

Integer Overflow and Underflow

Arithmetic bugs due to overflow or underflow were a common issue in earlier Solidity versions. However, TRUST uses Solidity version ^0.8.0, which includes built-in checks for arithmetic operations. Any overflow or underflow automatically causes a transaction to revert.

As such, no additional libraries like `SafeMath` are required because Solidity's native protections are sufficient, and all financial computations in TRUST (including balances, shares, and token transfers) are safely guarded by default.

Authorization and Access Control

Unprotected functions can lead to unauthorized access or manipulation of critical contract data. In TRUST, access control is carefully enforced across all contract functions:

- Only group members can register expenses or approve join requests.
- Only the original creator is marked as such.
- Settlement and simplification operations require valid membership.

These constraints are implemented via `require` statements and are thoroughly tested to ensure that malicious actors inject invalid data into a group.

Phishing Avoidance via `msg.sender` Enforcement

All identity-sensitive operations in TRUST rely exclusively on `msg.sender` to determine the caller's authority. No user addresses are ever passed as parameters to represent the initiator of an action. This design prevents phishing-style attacks via malicious frontends or calls, as only the account actually signing the transaction can perform operations.

Using TT for Speculation: Why It's Not Possible

As mentioned earlier, the TRUST token system is designed as a closed-loop economy. Users can mint TrustTokens (TT) by sending ETH, but there is no mechanism to withdraw ETH back. This simple but crucial design choice eliminates the possibility of using TT for speculation.