ERC-792: Arbitration Standard

Release 1.0.0

Apr 28, 2022

Contents:

CHAPTER 1

Introduction

Note: This tutorial requires basic Solidity programming skills.

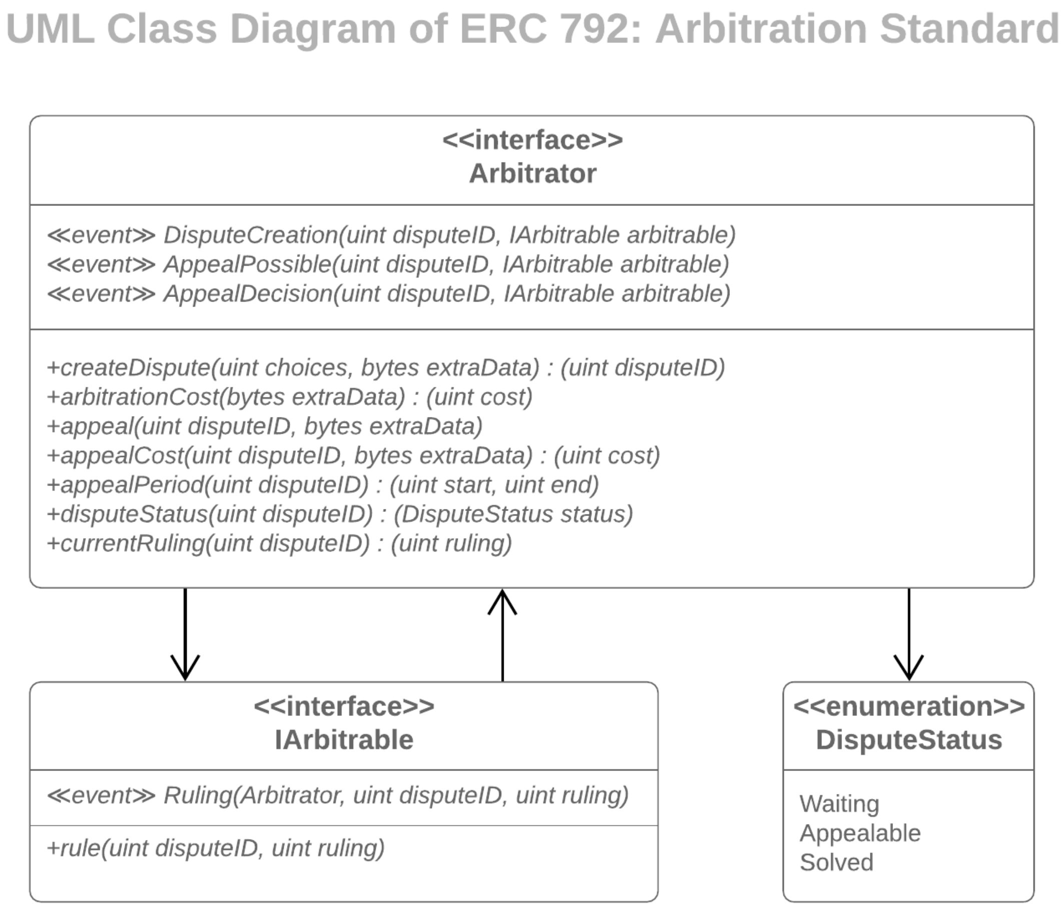
Note: See the original proposal of the standard here.

ERC-792: Arbitration Standard proposes a standard for arbitration on Ethereum. The standard has two types of smart contracts: `Arbitrable` and `Arbitrator`.

`Arbitrable` contracts are the contracts on which *rulings* of the authorized `Arbitrator` are enforceable.

`Arbitrator` contracts are the contracts which give *rulings* on disputes.

In other words, `Arbitrator` gives rulings, and `Arbitrable` enforces them.


UML Class Diagram of ERC 792: Arbitration Standard
<<interface>>
Arbitrator
≪event≫ DisputeCreation(uint disputeID, IArbitrable arbitrable)
≪event≫ AppealPossible(uint disputeID, IArbitrable arbitrable)
≪event≫ AppealDecision(uint disputeID, IArbitrable arbitrable)
+createDispute(uint choices, bytes extraData) : (uint disputeID)
+arbitrationCost(bytes extraData) : (uint cost)
+appeal(uint disputeID, bytes extraData)
+appealCost(uint disputeID, bytes extraData) : (uint cost)
+appealPeriod(uint disputeID) : (uint start, uint end)
+disputeStatus(uint disputeID) : (DisputeStatus status)
+currentRuling(uint disputeID) : (uint ruling)
<<interface>>
IArbitrable
≪event≫ Ruling(Arbitrator, uint disputeID, uint ruling)
+rule(uint disputeID, uint ruling)
<<enumeration>>
DisputeStatus
Waiting
Appealable
Solved

In the following topics, you will be guided through the usage of the standard. We will implement some examples for `Arbitrable` and `Arbitrator` contracts.

Note: Highlighted sections in source code examples indicate statements that are just modified.

Note: You can find the contracts used in this tutorial here.

We will also implement a very simple decentralized application on top of an `Arbitrable` we developed. You can see the live demo of the DAPP we will develop, here.

CHAPTER 2

Arbitrable Interface

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: [@remedcu]
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.0;

import "./IArbitrator.sol";

/**
 * @title IArbitrable
 * Arbitrable interface.
 * When developing arbitrable contracts, we need to:
 * - Define the action taken when a ruling is received by the contract.
 * - Allow dispute creation. For this a function must call arbitrator.createDispute
↪{value: _fee}(_choices,_extraData);
 */
interface IArbitrable {
    /**
     * @dev To be raised when a ruling is given.
     * @param _arbitrator The arbitrator giving the ruling.
     * @param _disputeID ID of the dispute in the Arbitrator contract.
     * @param _ruling The ruling which was given.
     */
    event Ruling(IArbitrator indexed _arbitrator, uint256 indexed _disputeID, uint256␣
↪_ruling);

    /**
     * @dev Give a ruling for a dispute. Must be called by the arbitrator.
     * The purpose of this function is to ensure that the address calling it has the␣
↪right to rule on the contract.
```

(continues on next page)

(continued from previous page)

```
     * @param _disputeID ID of the dispute in the Arbitrator contract.
     * @param _ruling Ruling given by the arbitrator. Note that 0 is reserved for
↪"Not able/wanting to make a decision".
     */
    function rule(uint256 _disputeID, uint256 _ruling) external;
}
```

`rule` is the function to be called by `Arbitrator` to enforce a *ruling* to a *dispute*.

`Ruling` is the event which has to be emitted whenever a *final ruling* is given. For example, inside `rule` function, where the ruling is final and gets enforced.

CHAPTER 3

Arbitrator Interface

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: [@remedcu]
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */

pragma solidity ^0.8.0;

import "./IArbitrable.sol";

/**
 * @title Arbitrator
 * Arbitrator abstract contract.
 * When developing arbitrator contracts we need to:
 * - Define the functions for dispute creation (createDispute) and appeal (appeal). Don't forget to store the arbitrated contract and the disputeID (which should be unique, may nbDisputes).
 * - Define the functions for cost display (arbitrationCost and appealCost).
 * - Allow giving rulings. For this a function must call arbitrable.rule(disputeID, ruling).
 */
interface IArbitrator {
    enum DisputeStatus {
        Waiting,
        Appealable,
        Solved
    }

    /**
     * @dev To be emitted when a dispute is created.
     * @param _disputeID ID of the dispute.
```

(continues on next page)

(continued from previous page)

```
     * @param _arbitrable The contract which created the dispute.
     */
    event DisputeCreation(uint256 indexed _disputeID, IArbitrable indexed _
↪arbitrable);

    /**
     * @dev To be emitted when a dispute can be appealed.
     * @param _disputeID ID of the dispute.
     * @param _arbitrable The contract which created the dispute.
     */
    event AppealPossible(uint256 indexed _disputeID, IArbitrable indexed _arbitrable);

    /**
     * @dev To be emitted when the current ruling is appealed.
     * @param _disputeID ID of the dispute.
     * @param _arbitrable The contract which created the dispute.
     */
    event AppealDecision(uint256 indexed _disputeID, IArbitrable indexed _arbitrable);

    /**
     * @dev Create a dispute. Must be called by the arbitrable contract.
     * Must be paid at least arbitrationCost(_extraData).
     * @param _choices Amount of choices the arbitrator can make in this dispute.
     * @param _extraData Can be used to give additional info on the dispute to be␣
↪created.
     * @return disputeID ID of the dispute created.
     */
    function createDispute(uint256 _choices, bytes calldata _extraData) external␣
↪payable returns (uint256 disputeID);

    /**
     * @dev Compute the cost of arbitration. It is recommended not to increase it␣
↪often, as it can be highly time and gas consuming for the arbitrated contracts to␣
↪cope with fee augmentation.
     * @param _extraData Can be used to give additional info on the dispute to be␣
↪created.
     * @return cost Amount to be paid.
     */
    function arbitrationCost(bytes calldata _extraData) external view returns␣
↪(uint256 cost);

    /**
     * @dev Appeal a ruling. Note that it has to be called before the arbitrator␣
↪contract calls rule.
     * @param _disputeID ID of the dispute to be appealed.
     * @param _extraData Can be used to give extra info on the appeal.
     */
    function appeal(uint256 _disputeID, bytes calldata _extraData) external payable;

    /**
     * @dev Compute the cost of appeal. It is recommended not to increase it often,␣
↪as it can be higly time and gas consuming for the arbitrated contracts to cope with␣
↪fee augmentation.
     * @param _disputeID ID of the dispute to be appealed.
     * @param _extraData Can be used to give additional info on the dispute to be␣
↪created.
     * @return cost Amount to be paid.
```

(continues on next page)

(continued from previous page)

```
     */
    function appealCost(uint256 _disputeID, bytes calldata _extraData) external view␣
↪returns (uint256 cost);

    /**
     * @dev Compute the start and end of the dispute's current or next appeal period,␣
↪if possible. If not known or appeal is impossible: should return (0, 0).
     * @param _disputeID ID of the dispute.
     * @return start The start of the period.
     * @return end The end of the period.
     */
    function appealPeriod(uint256 _disputeID) external view returns (uint256 start,␣
↪uint256 end);

    /**
     * @dev Return the status of a dispute.
     * @param _disputeID ID of the dispute to rule.
     * @return status The status of the dispute.
     */
    function disputeStatus(uint256 _disputeID) external view returns (DisputeStatus␣
↪status);

    /**
     * @dev Return the current ruling of a dispute. This is useful for parties to␣
↪know if they should appeal.
     * @param _disputeID ID of the dispute.
     * @return ruling The ruling which has been given or the one which will be given␣
↪if there is no appeal.
     */
    function currentRuling(uint256 _disputeID) external view returns (uint256 ruling);
}
```

3.1 Dispute Status

There are three statuses that the function `disputeStatus` can return; `Waiting`, `Appealable` and `Solved`:

- A *dispute* is in `Waiting` state when it arises (gets created, by `createDispute` function).
- Is in `Appealable` state when it got a *ruling* and the `Arbitrator` allows to *appeal* it. When the `Arbitrator` allows to appeal, it often gives a time period to do so. If a dispute is not appealed within that time, `disputeStatus` should return `Solved`.
- Is in `Solved` state when it got a *ruling* and the *ruling* is final. Note that this doesn't imply `rule` function on the `Arbitrable` has been called to enforce (execute) the *ruling*. It means that the decision on the *dispute* is final and to be executed.

3.2 Events

There are three events to be emitted:

- `DisputeCreation` when a *dispute* gets created by `createDispute` function.
- `AppealPossible` when appealing a *dispute* becomes possible.

- `AppealDecision` when *current ruling* is *appealed*.

3.3 Functions

And seven functions:

- `createDispute` should create a dispute with given number of possible `_choices` for decisions. `_extraData` is for passing any extra information for any kind of custom handling. While calling `createDispute`, caller has to pass required *arbitration fee*, otherwise `createDispute` should revert. `createDispute` should be called by an `Arbitrable`. Lastly, it should emit `DisputeCreation` event.
- `arbitrationCost` should return the *arbitration cost* that is required to *create a dispute*, in weis.
- `appeal` should appeal a dispute and should require the caller to pass the required *appeal fee*. `appeal` should be called by an `Arbitrable` and should emit the `AppealDecision` event.
- `appealCost` should return the *appeal fee* that is required to *appeal*, in weis.
- `appealPeriod` should return the time window, in `(start, end)` format, for appealing a ruling, if known in advance. If not known or appeal is impossible: should return `(0, 0)`.
- `disputeStatus` should return the status of dispute; `Waiting`, `Appealable` or `Solved`.
- `currentRuling` should return the current ruling of a dispute.

CHAPTER 4

Implementing an Arbitrable

Warning: Smart contracts in this tutorial are not intended for production but educational purposes. Beware of using them on main network.

When developing arbitrable contracts, we need to:

- Implement `rule` function to define an action to be taken when a ruling is received by the contract.
- Develop a logic to create disputes (via calling `createDispute` on Arbitrable)

Consider a case where two parties trade ether for goods. Payer wants to pay only if payee provides promised goods. So payer deposits payment amount into an escrow and if a dispute arises an arbitrator will resolve it.

There will be two scenarios:

1. No dispute arises, `payee` withdraws the funds.
2. **`payer` reclaims funds by depositing arbitration fee…**

    a. `payee` fails to deposit arbitration fee in `arbitrationFeeDepositPeriod` and `payer` wins by default. The arbitration fee deposit paid by `payer` refunded.

    b. `payee` deposits arbitration fee in time. Dispute gets created. `arbitrator` rules. Winner gets the arbitration fee refunded.

Notice that only in scenario 2b `arbitrator` intervenes. In other scenarios we don't create a dispute thus don't await for a ruling. Also, notice that in case of a dispute, the winning side gets reimbursed for the arbitration fee deposit. So in effect, the loser will be paying for the arbitration.

Let's start:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
```

(continues on next page)

(continued from previous page)

```
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";

contract SimpleEscrow is IArbitrable {
    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
```

`payer` deploys the contract depositing the payment amount and specifying `payee` address, `arbitrator` that is authorized to rule and `agreement` string. Notice that `payer = msg.sender`.

We made `reclamationPeriod` and `arbitrationFeeDepositPeriod` constant for sake of simplicity, they could be set by `payer` in the constructor too.

Let's implement the first scenario:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";

contract SimpleEscrow is IArbitrable {
    address payable public payer = payable(msg.sender);
    address payable public payee;
    uint256 public value;
    IArbitrator public arbitrator;
    string public agreement;
    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);

    enum Status {
        Initial,
        Reclaimed,
        Disputed,
```

(continues on next page)

(continued from previous page)

```
        Resolved
    }
    Status public status;

    uint256 public reclaimedAt;
```

In `releaseFunds` function, first we do state checks: transaction should be in `Status.Initial` and `reclamationPeriod` should be passed unless the caller is `payer`. If so, we update `status` to `Status.Resolved` and send the funds to `payee`.

Moving forward to second scenario:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";

contract SimpleEscrow is IArbitrable {
    address payable public payer = payable(msg.sender);
    address payable public payee;
    uint256 public value;
    IArbitrator public arbitrator;
    string public agreement;
    uint256 public createdAt;
    uint256 public constant reclamationPeriod = 3 minutes; // Timeframe is short on purpose to be able to test it quickly. Not for production use.
    uint256 public constant arbitrationFeeDepositPeriod = 3 minutes; // Timeframe is short on purpose to be able to test it quickly. Not for production use.

    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);

    enum Status {
        Initial,
        Reclaimed,
        Disputed,
        Resolved
    }
    Status public status;

    uint256 public reclaimedAt;

    enum RulingOptions {
```

(continues on next page)

(continued from previous page)

```
        RefusedToArbitrate,
        PayerWins,
        PayeeWins
    }
    uint256 constant numberOfRulingOptions = 2; // Notice that option 0 is reserved for RefusedToArbitrate.

    constructor(
        address payable _payee,
        IArbitrator _arbitrator,
        string memory _agreement
    ) payable {
        value = msg.value;
        payee = _payee;
        arbitrator = _arbitrator;
        agreement = _agreement;
        createdAt = block.timestamp;
    }

    function releaseFunds() public {
        if (status != Status.Initial) {
            revert InvalidStatus();
        }

        if (msg.sender != payer && block.timestamp - createdAt <= reclamationPeriod) {
            revert ReleasedTooEarly();
        }

        status = Status.Resolved;
        payee.send(value);
    }

    function reclaimFunds() public payable {
        if (status != Status.Initial && status != Status.Reclaimed) {
            revert InvalidStatus();
        }

        if (msg.sender != payer) {
            revert NotPayer();
        }

        if (status == Status.Reclaimed) {
            if (block.timestamp - reclaimedAt <= arbitrationFeeDepositPeriod) {
                revert PayeeDepositStillPending();
            }
            payer.send(address(this).balance);
            status = Status.Resolved;
        } else {
            if (block.timestamp - createdAt > reclamationPeriod) {
                revert ReclaimedTooLate();
            }
            uint256 requiredAmount = arbitrator.arbitrationCost("");
            if (msg.value < requiredAmount) {
                revert InsufficientPayment(msg.value, requiredAmount);
            }
            reclaimedAt = block.timestamp;
            status = Status.Reclaimed;
```

(continues on next page)

(continued from previous page)

```
        }
    }

    function depositArbitrationFeeForPayee() public payable {
        if (status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        arbitrator.createDispute{value: msg.value}(numberOfRulingOptions, "");
        status = Status.Disputed;
    }

    function rule(uint256 _disputeID, uint256 _ruling) public override {
        if (msg.sender != address(arbitrator)) {
            revert NotArbitrator();
        }
        if (status != Status.Disputed) {
            revert InvalidStatus();
        }
        if (_ruling > numberOfRulingOptions) {
            revert InvalidRuling(_ruling, numberOfRulingOptions);
        }

        status = Status.Resolved;
        if (_ruling == uint256(RulingOptions.PayerWins)) payer.send(address(this).
↪balance);
        else if (_ruling == uint256(RulingOptions.PayeeWins)) payee.
↪send(address(this).balance);
        emit Ruling(arbitrator, _disputeID, _ruling);
    }

    function remainingTimeToReclaim() public view returns (uint256) {
        if (status != Status.Initial) {
            revert InvalidStatus();
        }
        return
            (block.timestamp - createdAt) > reclamationPeriod ? 0 : (createdAt +␣
↪reclamationPeriod - block.timestamp);
    }

    function remainingTimeToDepositArbitrationFee() public view returns (uint256) {
        if (status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        return
            (block.timestamp - reclaimedAt) > arbitrationFeeDepositPeriod
                ? 0
                : (reclaimedAt + arbitrationFeeDepositPeriod - block.timestamp);
    }
}
```

`reclaimFunds` function lets `payer` to reclaim their funds. After `payer` calls this function for the first time the window (`arbitrationFeeDepositPeriod`) for `payee` to deposit arbitration fee starts. If they fail to deposit in time, `payer` can call the function for the second time and get the funds back. In case if `payee` deposits the arbitration fee in time a *dispute* gets created and the contract awaits arbitrator's decision.

We define enforcement of rulings in `rule` function. Whoever wins the dispute should get the funds and should get reimbursed for the arbitration fee. Recall that we took the arbitration fee deposit from both sides and used one of them to pay for the arbitrator. Thus the balance of the contract is at least funds plus arbitration fee. Therefore we send `address(this).balance` to the winner. Lastly, we emit `Ruling` as required in the standard.

And also we have two view functions to get remaining times, which will be useful for front-end development.

That's it! We implemented a very simple escrow using ERC-792.

CHAPTER 5

Implementing an Arbitrator

Warning: Smart contracts in this tutorial are not intended for production but educational purposes. Beware of using them on main network.

When developing arbitrator contracts we need to:

- Implement the functions `createDispute` and `appeal`. Don't forget to store the arbitrated contract and the disputeID (which should be unique).
- Implement the functions for cost display (`arbitrationCost` and `appealCost`).
- Allow enforcing rulings. For this a function must execute `arbitrable.rule(disputeID, ruling).`

To demonstrate how to use the standard, we will implement a very simple arbitrator where a single address gives rulings and there aren't any appeals.

Let's start by implementing cost functions:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
```

We set the arbitration fee to `0.1 ether` and the appeal fee to an astronomical amount which can't be afforded. So in practice, we disabled appeal, for simplicity. We made costs constant, again, for the sake of simplicity of this tutorial.

Next, we need a data structure to keep track of disputes:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract SimpleCentralizedArbitrator is IArbitrator {
    address public owner = msg.sender;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
```

Each dispute belongs to an `Arbitrable` contract, so we have `arbitrated` field for it. Each dispute will have a ruling stored in `ruling` field: For example, Party A wins (represented by `ruling = 1`) and Party B wins (represented by `ruling = 2`), recall that `ruling = 0` is reserved for "refused to arbitrate". We also store number of ruling options in `choices` to be able to avoid undefined rulings in the proxy function which executes `arbitrable.rule(disputeID, ruling)`. Finally, each dispute will have a status, and we store it inside `status` field.

Next, we can implement the function for creating disputes:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract SimpleCentralizedArbitrator is IArbitrator {
    address public owner = msg.sender;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
    }
```

(continues on next page)

(continued from previous page)

```
    Dispute[] public disputes;

    function arbitrationCost(bytes memory _extraData) public pure override returns␣
↪(uint256) {
        return 0.1 ether;
    }

    function appealCost(uint256 _disputeID, bytes memory _extraData) public pure␣
↪override returns (uint256) {
        return 2**250; // An unaffordable amount which practically avoids appeals.
    }
```

Note that `createDispute` function should be called by an *arbitrable*.

We require the caller to pay at least `arbitrationCost(_extraData)`. We could send back the excess payment, but we omitted it for the sake of simplicity.

Then, we create the dispute by pushing a new element to the array: `disputes.push( ... )`. The `push` function returns the resulting size of the array, thus we can use the return value of `disputes.push( ... ) -1` as `disputeID` starting from zero. Finally, we emit `DisputeCreation` as required in the standard.

We also need to implement getters for `status` and `ruling`:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract SimpleCentralizedArbitrator is IArbitrator {
    address public owner = msg.sender;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
    }

    Dispute[] public disputes;

    function arbitrationCost(bytes memory _extraData) public pure override returns␣
↪(uint256) {
        return 0.1 ether;
    }

    function appealCost(uint256 _disputeID, bytes memory _extraData) public pure␣
↪override returns (uint256) {
```

(continues on next page)

(continued from previous page)

```
        return 2**250; // An unaffordable amount which practically avoids appeals.
    }

    function createDispute(uint256 _choices, bytes memory _extraData)
        public
        payable
        override
        returns (uint256 disputeID)
    {
        uint256 requiredAmount = arbitrationCost(_extraData);
        if (msg.value < requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }

        disputes.push(
            Dispute({arbitrated: IArbitrable(msg.sender), choices: _choices, ruling:␣
↪0, status: DisputeStatus.Waiting})
```

Finally, we need a proxy function to call `rule` function of the `Arbitrable` contract. In this simple `Arbitrator` we will let one address, the creator of the contract, to give rulings. So let's start by storing contract creator's address:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract SimpleCentralizedArbitrator is IArbitrator {
    address public owner = msg.sender;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
    }

    Dispute[] public disputes;

    function arbitrationCost(bytes memory _extraData) public pure override returns␣
↪(uint256) {
        return 0.1 ether;
    }

    function appealCost(uint256 _disputeID, bytes memory _extraData) public pure␣
↪override returns (uint256) {
```

(continues on next page)

(continued from previous page)

```
        return 2**250; // An unaffordable amount which practically avoids appeals.
    }

    function createDispute(uint256 _choices, bytes memory _extraData)
        public
        payable
        override
        returns (uint256 disputeID)
    {
        uint256 requiredAmount = arbitrationCost(_extraData);
        if (msg.value < requiredAmount) {
```

Then the proxy function:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract SimpleCentralizedArbitrator is IArbitrator {
    address public owner = msg.sender;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
    }

    Dispute[] public disputes;

    function arbitrationCost(bytes memory _extraData) public pure override returns␣
↪(uint256) {
        return 0.1 ether;
    }

    function appealCost(uint256 _disputeID, bytes memory _extraData) public pure␣
↪override returns (uint256) {
        return 2**250; // An unaffordable amount which practically avoids appeals.
    }

    function createDispute(uint256 _choices, bytes memory _extraData)
        public
        payable
        override
```

(continues on next page)

(continued from previous page)

```
        returns (uint256 disputeID)
    {
        uint256 requiredAmount = arbitrationCost(_extraData);
        if (msg.value < requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }

        disputes.push(
            Dispute({arbitrated: IArbitrable(msg.sender), choices: _choices, ruling:␣
↪0, status: DisputeStatus.Waiting})
        );

        disputeID = disputes.length - 1;
        emit DisputeCreation(disputeID, IArbitrable(msg.sender));
    }

    function disputeStatus(uint256 _disputeID) public view override returns␣
↪(DisputeStatus status) {
        status = disputes[_disputeID].status;
    }
```

First we check the caller address, we should only let the `owner` execute this. Then we do sanity checks: the ruling given by the arbitrator should be chosen among the `choices` and it should not be possible to `rule` on an already solved dispute. Afterwards, we update `ruling` and `status` values of the dispute. Then we pay arbitration fee to the arbitrator (`owner`). Finally, we call `rule` function of the `arbitrated` to enforce the ruling.

Lastly, appeal functions:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract SimpleCentralizedArbitrator is IArbitrator {
    address public owner = msg.sender;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
    }

    Dispute[] public disputes;
```

(continues on next page)

(continued from previous page)

```
    function arbitrationCost(bytes memory _extraData) public pure override returns (uint256) {
        return 0.1 ether;
    }

    function appealCost(uint256 _disputeID, bytes memory _extraData) public pure override returns (uint256) {
        return 2**250; // An unaffordable amount which practically avoids appeals.
    }

    function createDispute(uint256 _choices, bytes memory _extraData)
        public
        payable
        override
        returns (uint256 disputeID)
    {
        uint256 requiredAmount = arbitrationCost(_extraData);
        if (msg.value < requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }

        disputes.push(
            Dispute({arbitrated: IArbitrable(msg.sender), choices: _choices, ruling: 0, status: DisputeStatus.Waiting})
        );

        disputeID = disputes.length - 1;
        emit DisputeCreation(disputeID, IArbitrable(msg.sender));
    }

    function disputeStatus(uint256 _disputeID) public view override returns (DisputeStatus status) {
        status = disputes[_disputeID].status;
    }

    function currentRuling(uint256 _disputeID) public view override returns (uint256 ruling) {
        ruling = disputes[_disputeID].ruling;
    }

    function rule(uint256 _disputeID, uint256 _ruling) public {
        if (msg.sender != owner) {
            revert NotOwner();
        }

        Dispute storage dispute = disputes[_disputeID];

        if (_ruling > dispute.choices) {
            revert InvalidRuling(_ruling, dispute.choices);
        }
        if (dispute.status != DisputeStatus.Waiting) {
            revert InvalidStatus(dispute.status, DisputeStatus.Waiting);
        }

        dispute.ruling = _ruling;
        dispute.status = DisputeStatus.Solved;
```

(continues on next page)

(continued from previous page)

```
        payable(msg.sender).send(arbitrationCost(""));
        dispute.arbitrated.rule(_disputeID, _ruling);
    }

    function appeal(uint256 _disputeID, bytes memory _extraData) public payable␣
↪override {
        uint256 requiredAmount = appealCost(_disputeID, _extraData);
        if (msg.value < requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }
    }

    function appealPeriod(uint256 _disputeID) public pure override returns (uint256␣
↪start, uint256 end) {
        return (0, 0);
    }
}
```

Just a dummy implementation to conform the interface, as we don’t actually implement appeal functionality.

That’s it, we have a working, very simple centralized arbitrator!

CHAPTER 6

ERC-1497: Evidence Standard

Note: See the original proposal of the standard here.

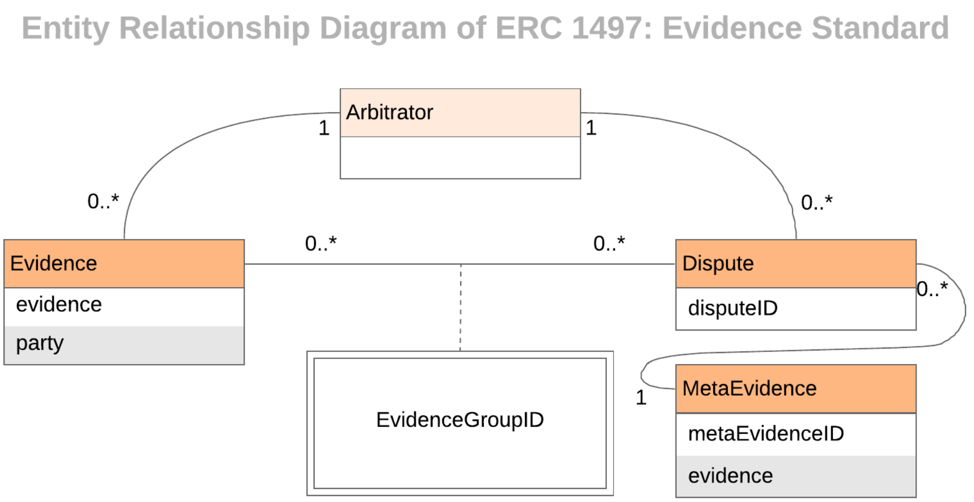
Warning: Smart contracts in this tutorial are not intended for production but educational purposes. Beware of using them on main network.

In `SimpleEscrow` contract, we used a `string` to store the agreement between the parties. The deployer could format that `string` anyway they like, as there are many ways to signal the information about the agreement.

Instead, having a standard would allow interoperability. That's why *ERC-1497: Evidence Standard* describes a standard approach for this. It has two categories of information: evidence and meta-evidence.

Evidence, as the name hints, is a piece of information to support a proposition. Meta-evidence is the information about the dispute itself: the agreement, parties involved, the thing that is to be decided, ruling options etc.

ERC-1497 introduces three new events: MetaEvidence, Evidence and Dispute.



```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.0;

import "../IArbitrator.sol";

/** @title IEvidence
 *  ERC-1497: Evidence Standard
 */
interface IEvidence {
    /**
     * @dev To be emitted when meta-evidence is submitted.
     * @param _metaEvidenceID Unique identifier of meta-evidence.
     * @param _evidence IPFS path to metaevidence, example: '/ipfs/
↪Qmarwkf7C9RuzDEJNnarT3WZ7kem5bk8DZAzx78acJjMFH/metaevidence.json'
     */
    event MetaEvidence(uint256 indexed _metaEvidenceID, string _evidence);

    /**
     * @dev To be raised when evidence is submitted. Should point to the resource␣
↪(evidences are not to be stored on chain due to gas considerations).
     * @param _arbitrator The arbitrator of the contract.
     * @param _evidenceGroupID Unique identifier of the evidence group the evidence␣
↪belongs to.
     * @param _party The address of the party submiting the evidence. Note that 0x0␣
↪refers to evidence not submitted by any party.
     * @param _evidence IPFS path to evidence, example: '/ipfs/
↪Qmarwkf7C9RuzDEJNnarT3WZ7kem5bk8DZAzx78acJjMFH/evidence.json'
     */
    event Evidence(
        IArbitrator indexed _arbitrator,
        uint256 indexed _evidenceGroupID,
```

(continues on next page)

(continued from previous page)

```
        address indexed _party,
        string _evidence
    );

    /**
     * @dev To be emitted when a dispute is created to link the correct meta-evidence to the disputeID.
     * @param _arbitrator The arbitrator of the contract.
     * @param _disputeID ID of the dispute in the Arbitrator contract.
     * @param _metaEvidenceID Unique identifier of meta-evidence.
     * @param _evidenceGroupID Unique identifier of the evidence group that is linked to this dispute.
     */
    event Dispute(
        IArbitrator indexed _arbitrator,
        uint256 indexed _disputeID,
        uint256 _metaEvidenceID,
        uint256 _evidenceGroupID
    );
}
```

- `event MetaEvidence` provides the context of the dispute, the question the arbitrators have to answer, the human readable meanings of rulings and specific modes of display for evidence. We use `_metaEvidenceID` to uniquely identify a piece of meta-evidence. This is necessary when there are multiple meta-evidence use-cases. `_evidence` contains the URI for meta-evidence JSON file.
- `event Evidence` links a piece of evidence with an arbitrator, sending party and dispute. `_evidence` contains the URI for evidence JSON file.
- `event Dispute` is raised when a dispute is created to link the proper meta-evidence and evidence group to the dispute. The event includes a reference to the arbitrator, a unique identifier for the dispute itself, the identifier to look up the meta-evidence event log and the identifier of the evidence group that can be used to look up all evidence submitted in the grouping.

Note: See the original proposal for standard evidence and meta-evidence JSON formats.

Let's return to `SimpleEscrow` and refactor it to implement ERC-1497 interface. Recall `SimpleEscrow`:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";

contract SimpleEscrow is IArbitrable {
    address payable public payer = payable(msg.sender);
    address payable public payee;
    uint256 public value;
```

(continues on next page)

(continued from previous page)

```
    IArbitrator public arbitrator;
    string public agreement;
    uint256 public createdAt;
    uint256 public constant reclamationPeriod = 3 minutes; // Timeframe is short on purpose to be able to test it quickly. Not for production use.
    uint256 public constant arbitrationFeeDepositPeriod = 3 minutes; // Timeframe is short on purpose to be able to test it quickly. Not for production use.

    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);

    enum Status {
        Initial,
        Reclaimed,
        Disputed,
        Resolved
    }
    Status public status;

    uint256 public reclaimedAt;

    enum RulingOptions {
        RefusedToArbitrate,
        PayerWins,
        PayeeWins
    }
    uint256 constant numberOfRulingOptions = 2; // Notice that option 0 is reserved for RefusedToArbitrate.

    constructor(
        address payable _payee,
        IArbitrator _arbitrator,
        string memory _agreement
    ) payable {
        value = msg.value;
        payee = _payee;
        arbitrator = _arbitrator;
        agreement = _agreement;
        createdAt = block.timestamp;
    }

    function releaseFunds() public {
        if (status != Status.Initial) {
            revert InvalidStatus();
        }

        if (msg.sender != payer && block.timestamp - createdAt <= reclamationPeriod) {
            revert ReleasedTooEarly();
        }

        status = Status.Resolved;
```

(continues on next page)

(continued from previous page)

```
        payee.send(value);
    }

    function reclaimFunds() public payable {
        if (status != Status.Initial && status != Status.Reclaimed) {
            revert InvalidStatus();
        }

        if (msg.sender != payer) {
            revert NotPayer();
        }

        if (status == Status.Reclaimed) {
            if (block.timestamp - reclaimedAt <= arbitrationFeeDepositPeriod) {
                revert PayeeDepositStillPending();
            }
            payer.send(address(this).balance);
            status = Status.Resolved;
        } else {
            if (block.timestamp - createdAt > reclamationPeriod) {
                revert ReclaimedTooLate();
            }
            uint256 requiredAmount = arbitrator.arbitrationCost("");
            if (msg.value < requiredAmount) {
                revert InsufficientPayment(msg.value, requiredAmount);
            }
            reclaimedAt = block.timestamp;
            status = Status.Reclaimed;
        }
    }

    function depositArbitrationFeeForPayee() public payable {
        if (status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        arbitrator.createDispute{value: msg.value}(numberOfRulingOptions, "");
        status = Status.Disputed;
    }

    function rule(uint256 _disputeID, uint256 _ruling) public override {
        if (msg.sender != address(arbitrator)) {
            revert NotArbitrator();
        }
        if (status != Status.Disputed) {
            revert InvalidStatus();
        }
        if (_ruling > numberOfRulingOptions) {
            revert InvalidRuling(_ruling, numberOfRulingOptions);
        }

        status = Status.Resolved;
        if (_ruling == uint256(RulingOptions.PayerWins)) payer.send(address(this).
↪balance);
        else if (_ruling == uint256(RulingOptions.PayeeWins)) payee.
↪send(address(this).balance);
        emit Ruling(arbitrator, _disputeID, _ruling);
    }
```

(continues on next page)

(continued from previous page)

```
    function remainingTimeToReclaim() public view returns (uint256) {
        if (status != Status.Initial) {
            revert InvalidStatus();
        }
        return
            (block.timestamp - createdAt) > reclamationPeriod ? 0 : (createdAt +␣
↪reclamationPeriod - block.timestamp);
    }

    function remainingTimeToDepositArbitrationFee() public view returns (uint256) {
        if (status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        return
            (block.timestamp - reclaimedAt) > arbitrationFeeDepositPeriod
                ? 0
                : (reclaimedAt + arbitrationFeeDepositPeriod - block.timestamp);
    }
}
```

Now, first let's implement *IEvidence*:

```
pragma solidity ^0.5;

import "../IArbitrable.sol";
import "../Arbitrator.sol";
import "../erc-1497/IEvidence.sol";

contract SimpleEscrowWithERC1497 is IArbitrable, IEvidence {
    address payable public payer = msg.sender;
    address payable public payee;
    uint public value;
    Arbitrator public arbitrator;
    string public agreement;
    uint public createdAt;
    uint constant public reclamationPeriod = 3 minutes;
    uint constant public arbitrationFeeDepositPeriod = 3 minutes;


    enum Status {Initial, Reclaimed, Disputed, Resolved}
    Status public status;

    uint public reclaimedAt;

    enum RulingOptions {PayerWins, PayeeWins, Count}

    constructor(address payable _payee, Arbitrator _arbitrator, string memory _
↪agreement) payable {
        value = msg.value;
        payee = _payee;
        arbitrator = _arbitrator;
        agreement = _agreement;
        createdAt = block.timestamp;
    }

    function releaseFunds() public {
```

(continues on next page)

(continued from previous page)

```
      require(status == Status.Initial, "Transaction is not in initial status.");

      if(msg.sender != payer)
          require(block.timestamp - createdAt > reclamationPeriod, "Payer still has␣
↪time to reclaim.");

      status = Status.Resolved;
      payee.send(value);
    }

    function reclaimFunds() public payable {
        require(status == Status.Initial || status == Status.Reclaimed, "Status␣
↪should be initial or reclaimed.");
        require(msg.sender == payer, "Only the payer can reclaim the funds.");

        if(status == Status.Reclaimed){
            require(block.timestamp - reclaimedAt > arbitrationFeeDepositPeriod,
↪"Payee still has time to deposit arbitration fee.");
            payer.send(address(this).balance);
            status = Status.Resolved;
        }
        else{
          require(block.timestamp - createdAt < reclamationPeriod, "Reclamation␣
↪period ended.");
          require(msg.value == arbitrator.arbitrationCost(""), "Can't reclaim funds␣
↪without depositing arbitration fee.");
          reclaimedAt = block.timestamp;
          status = Status.Reclaimed;
        }
    }

    function depositArbitrationFeeForPayee() public payable {
        require(status == Status.Reclaimed, "Payer didn't reclaim, nothing to dispute.
↪");
        arbitrator.createDispute{value: msg.value}(uint(RulingOptions.Count), "");
        status = Status.Disputed;
    }

    function rule(uint _disputeID, uint _ruling) public {
        require(msg.sender == address(arbitrator), "Only the arbitrator can execute␣
↪this.");
        require(status == Status.Disputed, "There should be dispute to execute a␣
↪ruling.");
        status = Status.Resolved;
        if(_ruling == uint(RulingOptions.PayerWins)) payer.send(address(this).
↪balance);
        else payee.send(address(this).balance);
        emit Ruling(arbitrator, _disputeID, _ruling);
    }

    function remainingTimeToReclaim() public view returns (uint) {
        if(status != Status.Initial) revert("Transaction is not in initial state.");
        return (createdAt + reclamationPeriod - block.timestamp) > reclamationPeriod ?
↪ 0 : (createdAt + reclamationPeriod - block.timestamp);
    }

    function remainingTimeToDepositArbitrationFee() public view returns (uint) {
```

(continues on next page)

(continued from previous page)

```
        if (status != Status.Reclaimed) revert("Funds are not reclaimed.");
        return (reclaimedAt + arbitrationFeeDepositPeriod - block.timestamp) >␣
↪arbitrationFeeDepositPeriod ? 0 : (reclaimedAt + arbitrationFeeDepositPeriod -␣
↪block.timestamp);
    }
}
```

And then, we will get rid of `string agreement`. Instead we need `uint metaevidenceID`, `string _metaevidence` that contains the URI to metaevidence JSON that is formatted according to the standard and we have to emit `MetaEvidence` event.

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";
import "../erc-1497/IEvidence.sol";

contract SimpleEscrowWithERC1497 is IArbitrable, IEvidence {
    address payable public payer = payable(msg.sender);
    address payable public payee;
    uint256 public value;
    IArbitrator public arbitrator;
    uint256 public constant reclamationPeriod = 3 minutes; // Timeframe is short on␣
↪purpose to be able to test it quickly. Not for production use.
    uint256 public constant arbitrationFeeDepositPeriod = 3 minutes; // Timeframe is␣
↪short on purpose to be able to test it quickly. Not for production use.

    uint256 public createdAt;
    uint256 public reclaimedAt;
    enum Status {
        Initial,
        Reclaimed,
        Disputed,
        Resolved
    }
    Status public status;

    enum RulingOptions {
        RefusedToArbitrate,
        PayerWins,
        PayeeWins
    }
    uint256 constant numberOfRulingOptions = 2;

    uint256 constant metaevidenceID = 0;
    uint256 constant evidenceGroupID = 0;

    error InvalidStatus();
```

(continues on next page)

(continued from previous page)

```
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error ThirdPartyNotAllowed();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error UnexistingDispute();

    constructor(
        address payable _payee,
        IArbitrator _arbitrator,
        string memory _metaevidence
    ) payable {
        value = msg.value;
        payee = _payee;
        arbitrator = _arbitrator;
        createdAt = block.timestamp;

        emit MetaEvidence(metaevidenceID, _metaevidence);

    function releaseFunds() public {
        require(status == Status.Initial, "Transaction is not in Initial state.");

        if (msg.sender != payer && block.timestamp - createdAt <= reclamationPeriod) {
            revert ReleasedTooEarly();
        }

        status = Status.Resolved;
        payee.send(value);
    }

    function reclaimFunds() public payable {
        if (status != Status.Initial && status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        if (msg.sender != payer) {
            revert NotPayer();
        }
```

We set the identifier of meta-evidence to constant zero, as there won't be multiple meta-evidence for this contract. Any constant number would do the job. Then we emit `MetaEvidence` with the provided `_metaevidence`. This string contains the URI from where the content of meta-evidence can be fetched.

Also, we need to emit `Dispute` when we create a new dispute:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

```

(continues on next page)

(continued from previous page)

```
import "../IArbitrable.sol";
import "../IArbitrator.sol";
import "../erc-1497/IEvidence.sol";

contract SimpleEscrowWithERC1497 is IArbitrable, IEvidence {
    address payable public payer = payable(msg.sender);
    address payable public payee;
    uint256 public value;
    IArbitrator public arbitrator;
    uint256 public constant reclamationPeriod = 3 minutes; // Timeframe is short on␣
↪purpose to be able to test it quickly. Not for production use.
    uint256 public constant arbitrationFeeDepositPeriod = 3 minutes; // Timeframe is␣
↪short on purpose to be able to test it quickly. Not for production use.

    uint256 public createdAt;
    uint256 public reclaimedAt;

    enum Status {
        Initial,
        Reclaimed,
        Disputed,
        Resolved
    }
    Status public status;

    enum RulingOptions {
        RefusedToArbitrate,
        PayerWins,
        PayeeWins
    }
    uint256 constant numberOfRulingOptions = 2;

    uint256 constant metaevidenceID = 0;
    uint256 constant evidenceGroupID = 0;

    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error ThirdPartyNotAllowed();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error UnexistingDispute();

    constructor(
        address payable _payee,
        IArbitrator _arbitrator,
        string memory _metaevidence
    ) payable {
        value = msg.value;
        payee = _payee;
        arbitrator = _arbitrator;
        createdAt = block.timestamp;

        emit MetaEvidence(metaevidenceID, _metaevidence);
```

(continues on next page)

(continued from previous page)

```
    }

    function releaseFunds() public {
        require(status == Status.Initial, "Transaction is not in Initial state.");

        if (msg.sender != payer && block.timestamp - createdAt <= reclamationPeriod) {
            revert ReleasedTooEarly();
        }

        status = Status.Resolved;
        payee.send(value);
    }

    function reclaimFunds() public payable {
        if (status != Status.Initial && status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        if (msg.sender != payer) {
            revert NotPayer();
        }
```

There will be only one dispute in this contract so we can use a constant zero for `evidenceGroupID`.

Lastly, we need a function to let parties submit evidence:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";
import "../erc-1497/IEvidence.sol";

contract SimpleEscrowWithERC1497 is IArbitrable, IEvidence {
    address payable public payer = payable(msg.sender);
    address payable public payee;
    uint256 public value;
    IArbitrator public arbitrator;
    uint256 public constant reclamationPeriod = 3 minutes; // Timeframe is short on␣
↪purpose to be able to test it quickly. Not for production use.
    uint256 public constant arbitrationFeeDepositPeriod = 3 minutes; // Timeframe is␣
↪short on purpose to be able to test it quickly. Not for production use.

    uint256 public createdAt;
    uint256 public reclaimedAt;

    enum Status {
        Initial,
        Reclaimed,
        Disputed,
        Resolved
```

(continues on next page)

(continued from previous page)

```
    }
    Status public status;

    enum RulingOptions {
        RefusedToArbitrate,
        PayerWins,
        PayeeWins
    }
    uint256 constant numberOfRulingOptions = 2;

    uint256 constant metaevidenceID = 0;
    uint256 constant evidenceGroupID = 0;

    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error ThirdPartyNotAllowed();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error UnexistingDispute();

    constructor(
        address payable _payee,
        IArbitrator _arbitrator,
        string memory _metaevidence
    ) payable {
        value = msg.value;
        payee = _payee;
        arbitrator = _arbitrator;
        createdAt = block.timestamp;

        emit MetaEvidence(metaevidenceID, _metaevidence);
    }

    function releaseFunds() public {
        require(status == Status.Initial, "Transaction is not in Initial state.");

        if (msg.sender != payer && block.timestamp - createdAt <= reclamationPeriod) {
            revert ReleasedTooEarly();
        }

        status = Status.Resolved;
        payee.send(value);
    }

    function reclaimFunds() public payable {
        if (status != Status.Initial && status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        if (msg.sender != payer) {
            revert NotPayer();
        }

        if (status == Status.Reclaimed) {
```

(continues on next page)

(continued from previous page)

```
            if (block.timestamp - reclaimedAt <= arbitrationFeeDepositPeriod) {
                revert PayeeDepositStillPending();
            }
            payer.send(address(this).balance);
            status = Status.Resolved;
        } else {
            if (block.timestamp - createdAt > reclamationPeriod) {
                revert ReclaimedTooLate();
            }
            uint256 requiredCost = arbitrator.arbitrationCost("");
            if (msg.value < requiredCost) {
                revert InsufficientPayment(msg.value, requiredCost);
            }
            reclaimedAt = block.timestamp;
            status = Status.Reclaimed;
        }
    }

    function depositArbitrationFeeForPayee() public payable {
        if (status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        uint256 disputeID = arbitrator.createDispute{value: msg.value}
↪(numberOfRulingOptions, "");
        status = Status.Disputed;
        emit Dispute(arbitrator, disputeID, metaevidenceID, evidenceGroupID);
    }

    function rule(uint256 _disputeID, uint256 _ruling) public override {
        if (msg.sender != address(arbitrator)) {
            revert NotArbitrator();
        }
        if (status != Status.Disputed) {
            revert UnexistingDispute();
        }
        if (_ruling > numberOfRulingOptions) {
            revert InvalidRuling(_ruling, numberOfRulingOptions);
        }

        status = Status.Resolved;
        if (_ruling == uint256(RulingOptions.PayerWins)) payer.send(address(this).
↪balance);
        else payee.send(address(this).balance);
        emit Ruling(arbitrator, _disputeID, _ruling);
    }

    function remainingTimeToReclaim() public view returns (uint256) {
        if (status != Status.Initial) revert("Transaction is not in Initial state.");
        return
            (createdAt + reclamationPeriod - block.timestamp) > reclamationPeriod
                ? 0
                : (createdAt + reclamationPeriod - block.timestamp);
    }

    function remainingTimeToDepositArbitrationFee() public view returns (uint256) {
        if (status != Status.Reclaimed) revert("Transaction is not in Reclaimed state.
↪");
```

(continues on next page)

(continued from previous page)

```
        return
            (reclaimedAt + arbitrationFeeDepositPeriod - block.timestamp) >␣
↪arbitrationFeeDepositPeriod
                ? 0
                : (reclaimedAt + arbitrationFeeDepositPeriod - block.timestamp);
    }

    function submitEvidence(string memory _evidence) public {
        if (status == Status.Resolved) {
            revert InvalidStatus();
        }
        if (msg.sender != payer && msg.sender != payee) {
            revert ThirdPartyNotAllowed();
        }
        emit Evidence(arbitrator, evidenceGroupID, msg.sender, _evidence);
    }
}
```

Congratulations, now your arbitrable is ERC-1497 compatible!

CHAPTER 7

A Simple DApp

Note: This tutorial requires basic Javascript programming skills and basic understanding of React Framework.

Note: You can find the finished React project source code here. You can test it live here.

Let's implement a simple decentralized application using `SimpleEscrowWithERC1497` contract.

We will create the simplest possible UI, as front-end development is out of the scope of this tutorial.

Tools used in this tutorial:

- Yarn
- React
- Create React App
- Bootstrap
- IPFS
- MetaMask

7.1 Arbitrable Side

7.1.1 Scaffolding The Project And Installing Dependencies

1. Run `yarn create react-app a-simple-dapp` to create a directory "a-simple-dapp" under your working directory and scaffold your application.
2. Run `yarn add web3@1.0.0-beta.37 react-bootstrap` to install required dependencies. Using exact versions for web3 and ipfs-http-client is recommended.

3. Add the following Bootstrap stylesheet in `index.html`

```
<link
rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous"
/>
```

4. Inside the application directory, running `yarn start` should run your application now. By default it runs on port 3000.

7.1.2 Ethereum Interface

Under the `src` directory, let's create a directory called `ethereum` for Ethereum-related files.

Setting Up Web3

Let's create a new file called `web3.js` under `ethereum` directory. We will put a helper inside it which will let us access MetaMask for sending transactions and querying the blockchain. For more details please see the MetaMask documentation .

Listing 1: web3.js

```
import Web3 from 'web3'

let web3

window.addEventListener('load', async () => {
  // Modern dapp browsers...
  if (window.ethereum) {
    window.web3 = new Web3(window.ethereum)
    try {
      // Request account access if needed
      await window.ethereum.enable()
      // Acccounts now exposed
    } catch (_) {
      // User denied account access...
    }
  }
  // Legacy dapp browsers...
  else if (window.web3) window.web3 = new Web3(web3.currentProvider)
  // Acccounts always exposed
  // Non-dapp browsers...
  else
    console.log(
      'Non-Ethereum browser detected. You should consider trying MetaMask!'
    )
})

if (typeof window !== 'undefined' && typeof window.web3 !== 'undefined') {
  console.log('Using the web3 object of the window...')
  web3 = new Web3(window.web3.currentProvider)
}

export default web3
```

Preparing Helper Functions For SimpleEscrowWithERC1497 And Arbitrator Contracts

We need to call functions of `SimpleEscrowWithERC1497` and the arbitrator (for `arbitrationCost`, to be able to send the correct amount when creating a dispute), so we need helpers for them.

We will import build artifacts of `SimpleEscrowWithERC1497` and `Arbitrator` contracts to use their ABIs (application binary interface). So we copy those under `ethereum` directory and create two helper files (`arbitrator.js` and `simple-escrow-with-erc1497.js`) using each of them.

Listing 2: simple-escrow-with-erc1497.js

```
import SimpleEscrowWithERC1497 from './simple-escrow-with-erc1497.json'
import web3 from './web3'

export const contractInstance = address =>
  new web3.eth.Contract(SimpleEscrowWithERC1497.abi, address)

export const deploy = (payer, payee, amount, arbitrator, metaevidence) =>
  new web3.eth.Contract(SimpleEscrowWithERC1497.abi)
    .deploy({
      arguments: [payee, arbitrator, metaevidence],
      data: SimpleEscrowWithERC1497.bytecode
    })
    .send({ from: payer, value: amount })

export const reclaimFunds = (senderAddress, instanceAddress, value) =>
  contractInstance(instanceAddress)
    .methods.reclaimFunds()
    .send({ from: senderAddress, value })

export const releaseFunds = (senderAddress, instanceAddress) =>
  contractInstance(instanceAddress)
    .methods.releaseFunds()
    .send({ from: senderAddress })

export const depositArbitrationFeeForPayee = (
  senderAddress,
  instanceAddress,
  value
) =>
  contractInstance(instanceAddress)
    .methods.depositArbitrationFeeForPayee()
    .send({ from: senderAddress, value })

export const reclamationPeriod = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.reclamationPeriod()
    .call()

export const arbitrationFeeDepositPeriod = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.arbitrationFeeDepositPeriod()
    .call()

export const createdAt = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.createdAt()
    .call()
```

(continues on next page)

(continued from previous page)

```
export const remainingTimeToReclaim = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.remainingTimeToReclaim()
    .call()

export const remainingTimeToDepositArbitrationFee = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.remainingTimeToDepositArbitrationFee()
    .call()

export const arbitrator = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.arbitrator()
    .call()

export const status = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.status()
    .call()

export const value = instanceAddress =>
  contractInstance(instanceAddress)
    .methods.value()
    .call()

export const submitEvidence = (instanceAddress, senderAddress, evidence) =>
  contractInstance(instanceAddress)
    .methods.submitEvidence(evidence)
    .send({ from: senderAddress })
```

Listing 3: arbitrator.js

```
import Arbitrator from './arbitrator.json'
import web3 from './web3'

export const contractInstance = address =>
  new web3.eth.Contract(Arbitrator.abi, address)

export const arbitrationCost = (instanceAddress, extraData) =>
  contractInstance(instanceAddress)
    .methods.arbitrationCost(web3.utils.utf8ToHex(extraData))
    .call()
```

Evidence and Meta-Evidence Helpers

Recall Evidence Standard JSON format. These two javascript object factories will be used to create JSON objects according to the standard.

Listing 4: generate-evidence.js

```
export default (fileURI, name, description) => ({
  fileURI,
  name,
  description
```

(continues on next page)

(continued from previous page)

```
})
```

Listing 5: generate-meta-evidence.js

```
export default (payer, payee, amount, title, description) => ({
  category: 'Escrow',
  title: title,
  description: description,
  question: 'Does payer deserves to be refunded?',
  rulingOptions: {
    type: 'single-select',
    titles: ['Refund the Payer', 'Pay the Payee'],
    descriptions: [
      'Select to return funds to the payer',
      'Select to release funds to the payee'
    ]
  },
  aliases: {
    [payer]: 'payer',
    [payee]: 'payee'
  },
  amount
})
```

7.1.3 Evidence Storage

We want to make sure evidence files are tamper-proof. So we need an immutable file storage. IPFS is perfect fit for this use-case. The following helper will let us publish evidence on IPFS, through the IPFS node at https://ipfs.kleros.io .

Listing 6: ipfs-publish.js

```
/**
 * Send file to IPFS network via the Kleros IPFS node
 * @param {string} fileName - The name that will be used to store the file. This is␣
↪useful to preserve extension type.
 * @param {ArrayBuffer} data - The raw data from the file to upload.
 * @return {object} ipfs response. Should include the hash and path of the stored␣
↪item.
 */
const ipfsPublish = async (fileName, data) => {
  const buffer = await Buffer.from(data)

  return new Promise((resolve, reject) => {
    fetch('https://ipfs.kleros.io/add', {
      method: 'POST',
      body: JSON.stringify({
        fileName,
        buffer
      }),
      headers: {
        'content-type': 'application/json'
      }
    })
```

(continues on next page)

(continued from previous page)

```
      .then(response => response.json())
      .then(success => resolve(success.data))
      .catch(err => reject(err))
  })
}

export default ipfsPublish
```

7.1.4 React Components

We will create a single-page react application to keep it simple. The main component, `App` will contain two sub-components:

- `Deploy`
- `Interact`

`Deploy` component will contain a form for arguments of `SimpleEscrowWithERC1497` deployment and a deploy button.

`Interact` component will have an input field for entering a contract address that is deployed already, to interact with. It will also have badges to show some state variable values of the contract. In addition, it will have three buttons for three main functions: `releaseFunds`, `reclaimFunds` and `depositArbitrationFeeForPayee`. Lastly, it will have a file picker and submit button for submitting evidence.

`App` will be responsible for accessing Ethereum. So it will give callbacks to `Deploy` and `Interact` to let them access Ethereum through `App`.

App

Listing 7: app.js

```
import React from "react";
import web3 from "./ethereum/web3";
import generateEvidence from "./ethereum/generate-evidence";
import generateMetaevidence from "./ethereum/generate-meta-evidence";
import * as SimpleEscrowWithERC1497 from "./ethereum/simple-escrow-with-erc1497";
import * as Arbitrator from "./ethereum/arbitrator";
import Ipfs from "ipfs-http-client";
import ipfsPublish from "./ipfs-publish";

import Container from "react-bootstrap/Container";
import Jumbotron from "react-bootstrap/Jumbotron";
import Button from "react-bootstrap/Button";
import Form from "react-bootstrap/Form";
import Row from "react-bootstrap/Row";
import Col from "react-bootstrap/Col";
import Deploy from "./deploy.js";
import Interact from "./interact.js";

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activeAddress: "0x0000000000000000000000000000000000000000",
```

(continues on next page)

(continued from previous page)

```
      lastDeployedAddress: "0x0000000000000000000000000000000000000000",
    };
    this.ipfs = new Ipfs({
      host: "ipfs.kleros.io",
      port: 5001,
      protocol: "https",
    });
  }

  deploy = async (amount, payee, arbitrator, title, description) => {
    const { activeAddress } = this.state;

    let metaevidence = generateMetaevidence(
      web3.utils.toChecksumAddress(activeAddress),
      web3.utils.toChecksumAddress(payee),
      amount,
      title,
      description
    );
    const enc = new TextEncoder();
    const ipfsHashMetaEvidenceObj = await ipfsPublish("metaEvidence.json", enc.
↪encode(JSON.stringify(metaevidence)));

    let result = await SimpleEscrowWithERC1497.deploy(
      activeAddress,
      payee,
      amount,
      arbitrator,

      "/ipfs/" + ipfsHashMetaEvidenceObj[1]["hash"] + ipfsHashMetaEvidenceObj[0]["path
↪"]
    );

    this.setState({ lastDeployedAddress: result._address });
  };

  load = (contractAddress) => SimpleEscrowWithERC1497.
↪contractInstance(contractAddress);

  reclaimFunds = async (contractAddress, value) => {
    const { activeAddress } = this.state;
    await SimpleEscrowWithERC1497.reclaimFunds(activeAddress, contractAddress, value);
  };

  releaseFunds = async (contractAddress) => {
    const { activeAddress } = this.state;

    await SimpleEscrowWithERC1497.releaseFunds(activeAddress, contractAddress);
  };

  depositArbitrationFeeForPayee = (contractAddress, value) => {
    const { activeAddress } = this.state;

    SimpleEscrowWithERC1497.depositArbitrationFeeForPayee(activeAddress,␣
↪contractAddress, value);
  };
```

(continues on next page)

(continued from previous page)

```
  reclamationPeriod = (contractAddress) => SimpleEscrowWithERC1497.
↪reclamationPeriod(contractAddress);

  arbitrationFeeDepositPeriod = (contractAddress) =>
    SimpleEscrowWithERC1497.arbitrationFeeDepositPeriod(contractAddress);

  remainingTimeToReclaim = (contractAddress) => SimpleEscrowWithERC1497.
↪remainingTimeToReclaim(contractAddress);

  remainingTimeToDepositArbitrationFee = (contractAddress) =>
    SimpleEscrowWithERC1497.remainingTimeToDepositArbitrationFee(contractAddress);

  arbitrationCost = (arbitratorAddress, extraData) => Arbitrator.
↪arbitrationCost(arbitratorAddress, extraData);

  arbitrator = (contractAddress) => SimpleEscrowWithERC1497.
↪arbitrator(contractAddress);

  status = (contractAddress) => SimpleEscrowWithERC1497.status(contractAddress);

  value = (contractAddress) => SimpleEscrowWithERC1497.value(contractAddress);

  submitEvidence = async (contractAddress, evidenceBuffer) => {
    const { activeAddress } = this.state;

    const result = await ipfsPublish("name", evidenceBuffer);

    let evidence = generateEvidence("/ipfs/" + result[0]["hash"], "name", "description
↪");
    const enc = new TextEncoder();
    const ipfsHashEvidenceObj = await ipfsPublish("evidence.json", enc.encode(JSON.
↪stringify(evidence)));

    SimpleEscrowWithERC1497.submitEvidence(contractAddress, activeAddress, "/ipfs/" +␣
↪ipfsHashEvidenceObj[0]["hash"]);
  };

  async componentDidMount() {
    if (window.ethereum && window.ethereum.isMetaMask)
      window.ethereum.request({ method: "eth_requestAccounts" }).then((accounts) => {
        this.setState({ activeAddress: accounts[0] });
      });
    else console.error("MetaMask account not detected :(");

    window.ethereum.on("accountsChanged", (accounts) => {
      this.setState({ activeAddress: accounts[0] });
    });
  }

  render() {
    const { lastDeployedAddress } = this.state;
    return (
      <Container>
        <Row>
          <Col>
            <h1 className="text-center my-5">A Simple DAPP Using␣
↪SimpleEscrowWithERC1497</h1>
```

(continues on next page)

(continued from previous page)

```
          </Col>
        </Row>

        <Row>
          <Col>
            <Deploy deployCallback={this.deploy} />
          </Col>
          <Col>
            <Interact
              arbitratorCallback={this.arbitrator}
              arbitrationCostCallback={this.arbitrationCost}
              escrowAddress={lastDeployedAddress}
              loadCallback={this.load}
              reclaimFundsCallback={this.reclaimFunds}
              releaseFundsCallback={this.releaseFunds}
              depositArbitrationFeeForPayeeCallback={this.
↪depositArbitrationFeeForPayee}
              remainingTimeToReclaimCallback={this.remainingTimeToReclaim}
              remainingTimeToDepositArbitrationFeeCallback={this.
↪remainingTimeToDepositArbitrationFee}
              statusCallback={this.status}
              valueCallback={this.value}
              submitEvidenceCallback={this.submitEvidence}
            />
          </Col>
        </Row>
        <Row>
          <Col>
            <Form action="https://centralizedarbitrator.netlify.com">
              <Jumbotron className="m-5 text-center">
                <h1>Need to interact with your arbitrator contract?</h1>
                <p>
                  We have a general purpose user interface for centralized␣
↪arbitrators (like we have developed in the
                  tutorial) already.
                </p>
                <p>
                  <Button type="submit" variant="primary">
                    Visit Centralized Arbitrator Dashboard
                  </Button>
                </p>
              </Jumbotron>
            </Form>
          </Col>
        </Row>
      </Container>
    );
  }
}

export default App;
```

Deploy

Listing 8: deploy.js

```
import React from 'react'
import Container from 'react-bootstrap/Container'
import Form from 'react-bootstrap/Form'
import Button from 'react-bootstrap/Button'
import Card from 'react-bootstrap/Card'

class Deploy extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      amount: '',
      payee: '',
      arbitrator: '',
      title: '',
      description: ''
    }
  }

  onAmountChange = e => {
    this.setState({ amount: e.target.value })
  }

  onPayeeChange = e => {
    this.setState({ payee: e.target.value })
  }

  onArbitratorChange = e => {
    this.setState({ arbitrator: e.target.value })
  }

  onTitleChange = e => {
    this.setState({ title: e.target.value })
  }

  onDescriptionChange = e => {
    this.setState({ description: e.target.value })
  }

  onDeployButtonClick = async e => {
    e.preventDefault()
    const { amount, payee, arbitrator, title, description } = this.state
    console.log(arbitrator)
    await this.props.deployCallback(
      amount,
      payee,
      arbitrator,
      title,
      description
    )
  }
  render() {
    const { amount, payee, arbitrator, title, description } = this.state

    return (
      <Container>
```

(continues on next page)

(continued from previous page)

```
        <Card className="my-4 text-center " style={{ width: 'auto' }}>
          <Card.Body>
            <Card.Title>Deploy</Card.Title>
            <Form>
              <Form.Group controlId="amount">
                <Form.Control
                  as="input"
                  rows="1"
                  value={amount}
                  onChange={this.onAmountChange}
                  placeholder={'Escrow Amount in Weis'}
                />
              </Form.Group>
              <Form.Group controlId="payee">
                <Form.Control
                  as="input"
                  rows="1"
                  value={payee}
                  onChange={this.onPayeeChange}
                  placeholder={'Payee Address'}
                />
              </Form.Group>
              <Form.Group controlId="arbitrator">
                <Form.Control
                  as="input"
                  rows="1"
                  value={arbitrator}
                  onChange={this.onArbitratorChange}
                  placeholder={'Arbitrator Address'}
                />
              </Form.Group>
              <Form.Group controlId="title">
                <Form.Control
                  as="input"
                  rows="1"
                  value={title}
                  onChange={this.onTitleChange}
                  placeholder={'Title'}
                />
              </Form.Group>
              <Form.Group controlId="description">
                <Form.Control
                  as="input"
                  rows="1"
                  value={description}
                  onChange={this.onDescriptionChange}
                  placeholder={'Describe The Agreement'}
                />
              </Form.Group>
              <Button
                variant="primary"
                type="button"
                onClick={this.onDeployButtonClick}
                block
              >
                Deploy
              </Button>
```

(continues on next page)

(continued from previous page)

```
            </Form>
          </Card.Body>
        </Card>
      </Container>
    )
  }
}

export default Deploy
```

Interact

Listing 9: interact.js

```
import React from 'react'
import Container from 'react-bootstrap/Container'
import Form from 'react-bootstrap/Form'
import Button from 'react-bootstrap/Button'
import Badge from 'react-bootstrap/Badge'
import ButtonGroup from 'react-bootstrap/ButtonGroup'
import Card from 'react-bootstrap/Card'
import InputGroup from 'react-bootstrap/InputGroup'

class Interact extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      escrowAddress: this.props.escrowAddress,
      remainingTimeToReclaim: 'Unassigned',
      remainingTimeToDepositArbitrationFee: 'Unassigned',
      status: 'Unassigned',
      arbitrator: 'Unassigned',
      value: 'Unassigned'
    }
  }

  async componentDidUpdate(prevProps) {
    if (this.props.escrowAddress !== prevProps.escrowAddress) {
      await this.setState({ escrowAddress: this.props.escrowAddress })
      this.updateBadges()
    }
  }

  onEscrowAddressChange = async e => {
    await this.setState({ escrowAddress: e.target.value })
    this.updateBadges()
  }

  updateBadges = async () => {
    const { escrowAddress, status } = this.state

    try {
      await this.setState({
        status: await this.props.statusCallback(escrowAddress)
      })
```

(continues on next page)

(continued from previous page)

```
    } catch (e) {
      console.error(e)
      this.setState({ status: 'ERROR' })
    }

    try {
      this.setState({
        arbitrator: await this.props.arbitratorCallback(escrowAddress)
      })
    } catch (e) {
      console.error(e)
      this.setState({ arbitrator: 'ERROR' })
    }

    try {
      this.setState({ value: await this.props.valueCallback(escrowAddress) })
    } catch (e) {
      console.error(e)
      this.setState({ value: 'ERROR' })
    }

    if (Number(status) === 0)
      try {
        this.setState({
          remainingTimeToReclaim: await this.props.remainingTimeToReclaimCallback(
            escrowAddress
          )
        })
      } catch (e) {
        console.error(e)
        this.setState({ status: 'ERROR' })
      }

    if (Number(status) === 1)
      try {
        this.setState({
          remainingTimeToDepositArbitrationFee: await this.props.
↪remainingTimeToDepositArbitrationFeeCallback(
            escrowAddress
          )
        })
      } catch (e) {
        console.error(e)
        this.setState({ status: 'ERROR' })
      }
  }

  onReclaimFundsButtonClick = async e => {
    e.preventDefault()
    const { escrowAddress } = this.state

    let arbitrator = await this.props.arbitratorCallback(escrowAddress)
    console.log(arbitrator)

    let arbitrationCost = await this.props.arbitrationCostCallback(
      arbitrator,
      ''
```

(continues on next page)

(continued from previous page)

```
    )

    await this.props.reclaimFundsCallback(escrowAddress, arbitrationCost)

    this.updateBadges()
  }

  onReleaseFundsButtonClick = async e => {
    e.preventDefault()
    const { escrowAddress } = this.state

    await this.props.releaseFundsCallback(escrowAddress)
    this.updateBadges()
  }

  onDepositArbitrationFeeFromPayeeButtonClicked = async e => {
    e.preventDefault()
    const { escrowAddress } = this.state

    let arbitrator = await this.props.arbitratorCallback(escrowAddress)
    let arbitrationCost = await this.props.arbitrationCostCallback(
      arbitrator,
      ''
    )

    await this.props.depositArbitrationFeeForPayeeCallback(
      escrowAddress,
      arbitrationCost
    )

    this.updateBadges()
  }

  onInput = e => {
    console.log(e.target.files)
    this.setState({ fileInput: e.target.files[0] })
    console.log('file input')
  }

  onSubmitButtonClick = async e => {
    e.preventDefault()
    const { escrowAddress, fileInput } = this.state
    console.log('submit clicked')
    console.log(fileInput)

    var reader = new FileReader()
    reader.readAsArrayBuffer(fileInput)
    reader.addEventListener('loadend', async () => {
      const buffer = Buffer.from(reader.result)
      this.props.submitEvidenceCallback(escrowAddress, buffer)
    })
  }

  render() {
    const { escrowAddress, fileInput } = this.state
    return (
      <Container className="container-fluid d-flex h-100 flex-column">
```

(continues on next page)

(continued from previous page)

```
        <Card className="h-100 my-4 text-center" style={{ width: 'auto' }}>
          <Card.Body>
            <Card.Title>Interact</Card.Title>
            <Form.Group controlId="escrow-address">
              <Form.Control
                className="text-center"
                as="input"
                rows="1"
                value={escrowAddress}
                onChange={this.onEscrowAddressChange}
              />
            </Form.Group>
            <Card.Subtitle className="mt-3 mb-1 text-muted">
              Smart Contract State
            </Card.Subtitle>

            <Badge className="m-1" pill variant="info">
              Status Code: {this.state.status}
            </Badge>
            <Badge className="m-1" pill variant="info">
              Escrow Amount in Weis: {this.state.value}
            </Badge>
            <Badge className="m-1" pill variant="info">
              Remaining Time To Reclaim Funds:{' '}
              {this.state.remainingTimeToReclaim}
            </Badge>
            <Badge className="m-1" pill variant="info">
              Remaining Time To Deposit Arbitration Fee:{' '}
              {this.state.remainingTimeToDepositArbitrationFee}
            </Badge>
            <Badge className="m-1" pill variant="info">
              Arbitrator: {this.state.arbitrator}
            </Badge>
            <ButtonGroup className="mt-3">
              <Button
                className="mr-2"
                variant="primary"
                type="button"
                onClick={this.onReleaseFundsButtonClick}
              >
                Release
              </Button>
              <Button
                className="mr-2"
                variant="secondary"
                type="button"
                onClick={this.onReclaimFundsButtonClick}
              >
                Reclaim
              </Button>
              <Button
                variant="secondary"
                type="button"
                onClick={this.onDepositArbitrationFeeFromPayeeButtonClicked}
                block
              >
                Deposit Arbitration Fee For Payee
```

(continues on next page)

(continued from previous page)

```
              </Button>
            </ButtonGroup>
            <InputGroup className="mt-3">
              <div className="input-group">
                <div className="custom-file">
                  <input
                    type="file"
                    className="custom-file-input"
                    id="inputGroupFile04"
                    onInput={this.onInput}
                  />
                  <label
                    className="text-left custom-file-label"
                    htmlFor="inputGroupFile04"
                  >
                    {(fileInput && fileInput.name) || 'Choose evidence file'}
                  </label>
                </div>
                <div className="input-group-append">
                  <button
                    className="btn btn-primary"
                    type="button"
                    onClick={this.onSubmitButtonClick}
                  >
                    Submit
                  </button>
                </div>
              </div>
            </InputGroup>
          </Card.Body>
        </Card>
      </Container>
    )
  }
}

export default Interact
```

7.2 Arbitrator Side

To interact with an arbitrator, we can use Centralized Arbitrator Dashboard. It let's setting up an arbitrator easily and provides UI to interact with, very useful for debugging and testing arbitrable implementations. As arbitrator, it deploys AutoAppealableArbitrator which is very similar to the one we developed in the tutorials.

To Use Centralized Arbitrator Dashboard (CAD):

1. Deploy a new arbitrator by specifying arbitration fee, choose a tiny amount for convenience, like *0.001* Ether.
2. Copy the arbitrator address and use this address as the arbitrator, in your arbitrable contract.
3. Create a dispute on your arbitrable contract.
4. Go back to CAD, select the arbitrator you created in the first step, by entering the contract address.
5. Now you should be able to see the dispute you created. You can give rulings to it using CAD.

Alternatively, you can use Kleros Arbitrator on Kovan network for testing. In that case, use this arbitrator address in your arbitrable contract, then simply go to https://court.kleros.io and switch your web3 provider to Kovan network. To be able to stake in a court, you will need Kovan PNK token, which you can buy from https://court.kleros.io/tokens.

Finally, when your arbitrable contract is ready, use Kleros Arbitrator on main network to integrate with Kleros.

CHAPTER 8

Implementing a Complex Arbitrable

Warning: Smart contracts in this tutorial are not intended for production but educational purposes. Beware of using them on main network.

Let's implement a full-fledged escrow this time, extending `SimpleEscrowWithERC1497` contract we implemented earlier. We will call it just `Escrow` this time.

Recall `SimpleEscrowWithERC1497`:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";
import "../erc-1497/IEvidence.sol";

contract SimpleEscrowWithERC1497 is IArbitrable, IEvidence {
    address payable public payer = payable(msg.sender);
    address payable public payee;
    uint256 public value;
    IArbitrator public arbitrator;
    uint256 public constant reclamationPeriod = 3 minutes; // Timeframe is short on␣
↪purpose to be able to test it quickly. Not for production use.
    uint256 public constant arbitrationFeeDepositPeriod = 3 minutes; // Timeframe is␣
↪short on purpose to be able to test it quickly. Not for production use.
```

(continues on next page)

(continued from previous page)

```
    uint256 public createdAt;
    uint256 public reclaimedAt;

    enum Status {
        Initial,
        Reclaimed,
        Disputed,
        Resolved
    }
    Status public status;

    enum RulingOptions {
        RefusedToArbitrate,
        PayerWins,
        PayeeWins
    }
    uint256 constant numberOfRulingOptions = 2;

    uint256 constant metaevidenceID = 0;
    uint256 constant evidenceGroupID = 0;

    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error ThirdPartyNotAllowed();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error UnexistingDispute();

    constructor(
        address payable _payee,
        IArbitrator _arbitrator,
        string memory _metaevidence
    ) payable {
        value = msg.value;
        payee = _payee;
        arbitrator = _arbitrator;
        createdAt = block.timestamp;

        emit MetaEvidence(metaevidenceID, _metaevidence);
    }

    function releaseFunds() public {
        require(status == Status.Initial, "Transaction is not in Initial state.");

        if (msg.sender != payer && block.timestamp - createdAt <= reclamationPeriod) {
            revert ReleasedTooEarly();
        }

        status = Status.Resolved;
        payee.send(value);
    }

    function reclaimFunds() public payable {
```

(continues on next page)

(continued from previous page)

```
        if (status != Status.Initial && status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        if (msg.sender != payer) {
            revert NotPayer();
        }

        if (status == Status.Reclaimed) {
            if (block.timestamp - reclaimedAt <= arbitrationFeeDepositPeriod) {
                revert PayeeDepositStillPending();
            }
            payer.send(address(this).balance);
            status = Status.Resolved;
        } else {
            if (block.timestamp - createdAt > reclamationPeriod) {
                revert ReclaimedTooLate();
            }
            uint256 requiredCost = arbitrator.arbitrationCost("");
            if (msg.value < requiredCost) {
                revert InsufficientPayment(msg.value, requiredCost);
            }
            reclaimedAt = block.timestamp;
            status = Status.Reclaimed;
        }
    }

    function depositArbitrationFeeForPayee() public payable {
        if (status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        uint256 disputeID = arbitrator.createDispute{value: msg.value}(numberOfRulingOptions, "");
        status = Status.Disputed;
        emit Dispute(arbitrator, disputeID, metaevidenceID, evidenceGroupID);
    }

    function rule(uint256 _disputeID, uint256 _ruling) public override {
        if (msg.sender != address(arbitrator)) {
            revert NotArbitrator();
        }
        if (status != Status.Disputed) {
            revert UnexistingDispute();
        }
        if (_ruling > numberOfRulingOptions) {
            revert InvalidRuling(_ruling, numberOfRulingOptions);
        }

        status = Status.Resolved;
        if (_ruling == uint256(RulingOptions.PayerWins)) payer.send(address(this).balance);
        else payee.send(address(this).balance);
        emit Ruling(arbitrator, _disputeID, _ruling);
    }

    function remainingTimeToReclaim() public view returns (uint256) {
        if (status != Status.Initial) revert("Transaction is not in Initial state.");
        return
```

(continues on next page)

(continued from previous page)

```
            (createdAt + reclamationPeriod - block.timestamp) > reclamationPeriod
                ? 0
                : (createdAt + reclamationPeriod - block.timestamp);
    }

    function remainingTimeToDepositArbitrationFee() public view returns (uint256) {
        if (status != Status.Reclaimed) revert("Transaction is not in Reclaimed state.
↪");
        return
            (reclaimedAt + arbitrationFeeDepositPeriod - block.timestamp) >␣
↪arbitrationFeeDepositPeriod
                ? 0
                : (reclaimedAt + arbitrationFeeDepositPeriod - block.timestamp);
    }

    function submitEvidence(string memory _evidence) public {
        if (status == Status.Resolved) {
            revert InvalidStatus();
        }
        if (msg.sender != payer && msg.sender != payee) {
            revert ThirdPartyNotAllowed();
        }
        emit Evidence(arbitrator, evidenceGroupID, msg.sender, _evidence);
    }
}
```

The payer needs to deploy a contract for each transaction, but contract deployment is expensive. Instead, we could use the same contract for multiple transactions between arbitrary parties with arbitrary arbitrators.

Let's separate contract deployment and transaction creation:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrable.sol";
import "../IArbitrator.sol";
import "../erc-1497/IEvidence.sol";

contract Escrow is IArbitrable, IEvidence {
    enum Status {
        Initial,
        Reclaimed,
        Disputed,
        Resolved
    }
    enum RulingOptions {
        RefusedToArbitrate,
        PayerWins,
        PayeeWins
    }
```

(continues on next page)

(continued from previous page)

```
    uint256 constant numberOfRulingOptions = 2;

    error InvalidStatus();
    error ReleasedTooEarly();
    error NotPayer();
    error NotArbitrator();
    error ThirdPartyNotAllowed();
    error PayeeDepositStillPending();
    error ReclaimedTooLate();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);

    struct TX {
        address payable payer;
        address payable payee;
        IArbitrator arbitrator;
        Status status;
        uint256 value;
        uint256 disputeID;
        uint256 createdAt;
        uint256 reclaimedAt;
        uint256 payerFeeDeposit;
        uint256 payeeFeeDeposit;
        uint256 reclamationPeriod;
        uint256 arbitrationFeeDepositPeriod;
    }

    TX[] public txs;
    mapping(uint256 => uint256) disputeIDtoTXID;

    function newTransaction(
        address payable _payee,
        IArbitrator _arbitrator,
        string memory _metaevidence,
        uint256 _reclamationPeriod,
        uint256 _arbitrationFeeDepositPeriod
    ) public payable returns (uint256 txID) {
        emit MetaEvidence(txs.length, _metaevidence);

        txs.push(
            TX({
                payer: payable(msg.sender),
                payee: _payee,
                arbitrator: _arbitrator,
                status: Status.Initial,
                value: msg.value,
                disputeID: 0,
                createdAt: block.timestamp,
                reclaimedAt: 0,
                payerFeeDeposit: 0,
                payeeFeeDeposit: 0,
                reclamationPeriod: _reclamationPeriod,
                arbitrationFeeDepositPeriod: _arbitrationFeeDepositPeriod
            })
        );

        txID = txs.length;
```

(continues on next page)

(continued from previous page)

```
    }

    function releaseFunds(uint256 _txID) public {
        TX storage transaction = txs[_txID];

        if (transaction.status != Status.Initial) {
            revert InvalidStatus();
        }
        if (
            msg.sender != transaction.payer && block.timestamp - transaction.
↪createdAt <= transaction.reclamationPeriod
        ) {
            revert ReleasedTooEarly();
        }

        transaction.status = Status.Resolved;
        transaction.payee.send(transaction.value);
    }

    function reclaimFunds(uint256 _txID) public payable {
        TX storage transaction = txs[_txID];

        if (transaction.status != Status.Initial && transaction.status != Status.
↪Reclaimed) {
            revert InvalidStatus();
        }
        if (msg.sender != transaction.payer) {
            revert NotPayer();
        }

        if (transaction.status == Status.Reclaimed) {
            if (block.timestamp - transaction.reclaimedAt <= transaction.
↪arbitrationFeeDepositPeriod) {
                revert PayeeDepositStillPending();
            }
            transaction.payer.send(transaction.value + transaction.payerFeeDeposit);
            transaction.status = Status.Resolved;
        } else {
            if (block.timestamp - transaction.createdAt > transaction.
↪reclamationPeriod) {
                revert ReclaimedTooLate();
            }

            uint256 requiredAmount = transaction.arbitrator.arbitrationCost("");
            if (msg.value < requiredAmount) {
                revert InsufficientPayment(msg.value, requiredAmount);
            }

            transaction.payerFeeDeposit = msg.value;
            transaction.reclaimedAt = block.timestamp;
            transaction.status = Status.Reclaimed;
        }
    }

    function depositArbitrationFeeForPayee(uint256 _txID) public payable {
        TX storage transaction = txs[_txID];
```

(continues on next page)

(continued from previous page)

```
        if (transaction.status != Status.Reclaimed) {
            revert InvalidStatus();
        }

        transaction.payeeFeeDeposit = msg.value;
        transaction.disputeID = transaction.arbitrator.createDispute{value: msg.value}
↪(numberOfRulingOptions, "");
        transaction.status = Status.Disputed;
        disputeIDtoTXID[transaction.disputeID] = _txID;
        emit Dispute(transaction.arbitrator, transaction.disputeID, _txID, _txID);
    }

    function rule(uint256 _disputeID, uint256 _ruling) public override {
        uint256 txID = disputeIDtoTXID[_disputeID];
        TX storage transaction = txs[txID];

        if (msg.sender != address(transaction.arbitrator)) {
            revert NotArbitrator();
        }
        if (transaction.status != Status.Disputed) {
            revert InvalidStatus();
        }
        if (_ruling > numberOfRulingOptions) {
            revert InvalidRuling(_ruling, numberOfRulingOptions);
        }
        transaction.status = Status.Resolved;

        if (_ruling == uint256(RulingOptions.PayerWins))
            transaction.payer.send(transaction.value + transaction.payerFeeDeposit);
        else transaction.payee.send(transaction.value + transaction.payeeFeeDeposit);
        emit Ruling(transaction.arbitrator, _disputeID, _ruling);
    }

    function submitEvidence(uint256 _txID, string memory _evidence) public {
        TX storage transaction = txs[_txID];

        if (transaction.status == Status.Resolved) {
            revert InvalidStatus();
        }

        if (msg.sender != transaction.payer && msg.sender != transaction.payee) {
            revert ThirdPartyNotAllowed();
        }
        emit Evidence(transaction.arbitrator, _txID, msg.sender, _evidence);
    }

    function remainingTimeToReclaim(uint256 _txID) public view returns (uint256) {
        TX storage transaction = txs[_txID];

        if (transaction.status != Status.Initial) {
            revert InvalidStatus();
        }
        return
            (block.timestamp - transaction.createdAt) > transaction.reclamationPeriod
                ? 0
                : (transaction.createdAt + transaction.reclamationPeriod - block.
↪timestamp);
```

(continues on next page)

(continued from previous page)

```
    }

    function remainingTimeToDepositArbitrationFee(uint256 _txID) public view returns␣
↪(uint256) {
        TX storage transaction = txs[_txID];

        if (transaction.status != Status.Reclaimed) {
            revert InvalidStatus();
        }
        return
            (block.timestamp - transaction.reclaimedAt) > transaction.
↪arbitrationFeeDepositPeriod
                ? 0
                : (transaction.reclaimedAt + transaction.arbitrationFeeDepositPeriod -
↪ block.timestamp);
    }
}
```

We first start by removing the global state variables and defining `TX` struct. Each instance of this struct will represent a transaction, thus will have transaction-specific variables instead of globals. We stored transactions inside `txs` array. We also created new transactions via `newTransaction` function.

`newTransaction` function simply takes transaction-specific information and pushes a `TX` into `txs`. This `txs` array is append-only, we will never remove any item. By implementing this, we can uniquely identify each transaction by their index in the array.

Next, we updated all the functions with transaction-specific variables instead of globals. Changes are merely adding `tx.` prefixes in front of expressions.

We also stored fee deposits for each party, as the smart contract now has balances for multiple transactions that we can't `send(address(this).balance)`. Instead, we used `tx.payer.send(tx.value + tx.payerFeeDeposit);` if `payer` wins and `tx.payee.send(tx.value + tx.payeeFeeDeposit);` if `payee` wins.

Notice that `rule` function has no transaction ID parameter, but we need to obtain transaction details of given dispute. We achieved this by storing the transaction ID for respective dispute ID as `disputeIDtoTXID`. Just after dispute creation (inside `depositArbitrationFeeForPayee`), we store this relation with `disputeIDtoTXID[tx.disputeID] = _txID;` statement.

Good job! Now we have an escrow contract which can handle multiple transactions between different parties and arbitrators.

CHAPTER 9

Implementing a Complex Arbitrator

Warning: Smart contracts in this tutorial are not intended for production but educational purposes. Beware of using them on main network.

We will refactor `SimpleCentralizedArbitrator` to add appeal functionality and dynamic costs.

Recall `SimpleCentralizedArbitrator`:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract SimpleCentralizedArbitrator is IArbitrator {
    address public owner = msg.sender;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
```

(continues on next page)

(continued from previous page)

```
    }

    Dispute[] public disputes;

    function arbitrationCost(bytes memory _extraData) public pure override returns␣
↪(uint256) {
        return 0.1 ether;
    }

    function appealCost(uint256 _disputeID, bytes memory _extraData) public pure␣
↪override returns (uint256) {
        return 2**250; // An unaffordable amount which practically avoids appeals.
    }

    function createDispute(uint256 _choices, bytes memory _extraData)
        public
        payable
        override
        returns (uint256 disputeID)
    {
        uint256 requiredAmount = arbitrationCost(_extraData);
        if (msg.value < requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }

        disputes.push(
            Dispute({arbitrated: IArbitrable(msg.sender), choices: _choices, ruling:␣
↪0, status: DisputeStatus.Waiting})
        );

        disputeID = disputes.length - 1;
        emit DisputeCreation(disputeID, IArbitrable(msg.sender));
    }

    function disputeStatus(uint256 _disputeID) public view override returns␣
↪(DisputeStatus status) {
        status = disputes[_disputeID].status;
    }

    function currentRuling(uint256 _disputeID) public view override returns (uint256␣
↪ruling) {
        ruling = disputes[_disputeID].ruling;
    }

    function rule(uint256 _disputeID, uint256 _ruling) public {
        if (msg.sender != owner) {
            revert NotOwner();
        }

        Dispute storage dispute = disputes[_disputeID];

        if (_ruling > dispute.choices) {
            revert InvalidRuling(_ruling, dispute.choices);
        }
        if (dispute.status != DisputeStatus.Waiting) {
            revert InvalidStatus(dispute.status, DisputeStatus.Waiting);
        }
```

(continues on next page)

(continued from previous page)

```
        dispute.ruling = _ruling;
        dispute.status = DisputeStatus.Solved;

        payable(msg.sender).send(arbitrationCost(""));
        dispute.arbitrated.rule(_disputeID, _ruling);
    }

    function appeal(uint256 _disputeID, bytes memory _extraData) public payable␣
↪override {
        uint256 requiredAmount = appealCost(_disputeID, _extraData);
        if (msg.value < requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }
    }

    function appealPeriod(uint256 _disputeID) public pure override returns (uint256␣
↪start, uint256 end) {
        return (0, 0);
    }
}
```

First, let's implement the appeal:

```
pragma solidity ^0.5;

import "../Arbitrator.sol";

contract CentralizedArbitratorWithAppeal is Arbitrator {

    address public owner = msg.sender;
    uint constant appealWindow = 3 minutes;

    struct Dispute {
        IArbitrable arbitrated;
        uint choices;
        uint ruling;
        DisputeStatus status;
        uint appealPeriodStart;
        uint appealPeriodEnd;
    }

    Dispute[] public disputes;

    function arbitrationCost(bytes memory _extraData) public view returns(uint fee) {
        fee = 0.1 ether;
    }

    function appealCost(uint _disputeID, bytes memory _extraData) public view␣
↪returns(uint fee) {
        fee = 2**250; // An unaffordable amount which practically avoids appeals.
    }

    function createDispute(uint _choices, bytes memory _extraData) public payable␣
↪returns(uint disputeID) {
        super.createDispute(_choices, _extraData);
        disputeID = disputes.push(Dispute({
```

(continues on next page)

(continued from previous page)

```
          arbitrated: IArbitrable(msg.sender),
          choices: _choices,
          ruling: 0,
          status: DisputeStatus.Waiting,
          appealPeriodStart: 0,
          appealPeriodEnd: 0
        }));

        emit DisputeCreation(disputeID, IArbitrable(msg.sender));
    }

    function disputeStatus(uint _disputeID) public view returns(DisputeStatus status)
↪{
        Dispute storage dispute = disputes[_disputeID];
        if (disputes[_disputeID].status == DisputeStatus.Appealable && block.
↪timestamp >= dispute.appealPeriodEnd)
            return DisputeStatus.Solved;
        else
            return disputes[_disputeID].status;
    }

    function currentRuling(uint _disputeID) public view returns(uint ruling) {
        ruling = disputes[_disputeID].ruling;
    }

    function giveRuling(uint _disputeID, uint _ruling) public {
        require(msg.sender == owner, "Only the owner of this contract can execute␣
↪rule function.");

        Dispute storage dispute = disputes[_disputeID];

        require(_ruling <= dispute.choices, "Ruling out of bounds!");
        require(dispute.status != DisputeStatus.Solved, "Can't rule an already solved␣
↪dispute!");

        dispute.ruling = _ruling;
        dispute.status = DisputeStatus.Appealable;
        dispute.appealPeriodStart = block.timestamp;
        dispute.appealPeriodEnd = dispute.appealPeriodStart + appealWindow;
    }

    function executeRuling(uint _disputeID) public {
        Dispute storage dispute = disputes[_disputeID];
        require(dispute.status == DisputeStatus.Appealable, "The dispute must be␣
↪appealable.");
        require(block.timestamp >= dispute.appealPeriodEnd, "The dispute must be␣
↪executed after its appeal period has ended.");

        dispute.status = DisputeStatus.Solved;
        dispute.arbitrated.rule(_disputeID, dispute.ruling);
    }

    function appeal(uint _disputeID, bytes memory _extraData) public payable {
        Dispute storage dispute = disputes[_disputeID];

        super.appeal(_disputeID, _extraData);
```

(continues on next page)

(continued from previous page)

```
        require(dispute.status == DisputeStatus.Appealable, "The dispute must be␣
↪appealable.");
        require(block.timestamp < dispute.appealPeriodEnd, "The appeal must occur␣
↪before the end of the appeal period.");

        dispute.status = DisputeStatus.Waiting;
    }

    function appealPeriod(uint _disputeID) public view returns(uint start, uint end) {
        Dispute storage dispute = disputes[_disputeID];

        return (dispute.appealPeriodStart, dispute.appealPeriodEnd);
  }
}
```

We first define `appealWindow` constant, which is the amount of time a dispute stays appealable.

To implement `appealPeriod` function of the ERC-792 interface, we define two additional variables in `Dispute` struct: `appealPeriodStart` and `appealPeriodEnd`.

`DisputeStatus` function is also updated to handle the case where a dispute has `DisputeStatus.Appealable` status, but the appeal window is closed, so actually it is `DisputeStatus.Solved`.

The important change is we divided proxy `rule` function into two parts.

- `giveRuling`: Gives ruling, but does not enforce it.
- `executeRuling` Enforces ruling, only after the appeal window is closed.

Before, there was no appeal functionality, so we didn't have to wait for appeal and ruling was enforced immediately after giving the ruling. Now we need to do them separately.

`appeal` function checks whether the dispute is eligible for appeal and performs the appeal by setting `status` back to the default value, `DisputeStatus.Waiting`.

Now let's revisit cost functions:

```
/**
 * @authors: [@ferittuncer, @hbarcelos]
 * @reviewers: []
 * @auditors: []
 * @bounties: []
 * @deployments: []
 * SPDX-License-Identifier: MIT
 */
pragma solidity ^0.8.9;

import "../IArbitrator.sol";

contract CentralizedArbitratorWithAppeal is IArbitrator {
    address public owner = msg.sender;
    uint256 constant appealWindow = 3 minutes;
    uint256 internal arbitrationFee = 1e15;

    error NotOwner();
    error InsufficientPayment(uint256 _available, uint256 _required);
    error InvalidRuling(uint256 _ruling, uint256 _numberOfChoices);
    error InvalidStatus(DisputeStatus _current, DisputeStatus _expected);
    error BeforeAppealPeriodEnd(uint256 _currentTime, uint256 _appealPeriodEnd);
```

(continues on next page)

(continued from previous page)

```
    error AfterAppealPeriodEnd(uint256 _currentTime, uint256 _appealPeriodEnd);

    struct Dispute {
        IArbitrable arbitrated;
        uint256 choices;
        uint256 ruling;
        DisputeStatus status;
        uint256 appealPeriodStart;
        uint256 appealPeriodEnd;
        uint256 appealCount;
    }

    Dispute[] public disputes;

    function arbitrationCost(bytes memory _extraData) public view override returns␣
↪(uint256) {
        return arbitrationFee;
    }

    function appealCost(uint256 _disputeID, bytes memory _extraData) public view␣
↪override returns (uint256) {
        return arbitrationFee * (2**(disputes[_disputeID].appealCount));
    }

    function setArbitrationCost(uint256 _newCost) public {
        arbitrationFee = _newCost;
    }

    function createDispute(uint256 _choices, bytes memory _extraData)
        public
        payable
        override
        returns (uint256 disputeID)
    {
        uint256 requiredAmount = arbitrationCost(_extraData);
        if (msg.value > requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }

        disputes.push(
            Dispute({
                arbitrated: IArbitrable(msg.sender),
                choices: _choices,
                ruling: 0,
                status: DisputeStatus.Waiting,
                appealPeriodStart: 0,
                appealPeriodEnd: 0,
                appealCount: 0
            })
        );

        disputeID = disputes.length - 1;
        emit DisputeCreation(disputeID, IArbitrable(msg.sender));
    }

    function disputeStatus(uint256 _disputeID) public view override returns␣
↪(DisputeStatus status) {
```

(continues on next page)

(continued from previous page)

```
        Dispute storage dispute = disputes[_disputeID];
        if (disputes[_disputeID].status == DisputeStatus.Appealable && block.
↪timestamp >= dispute.appealPeriodEnd)
            return DisputeStatus.Solved;
        else return disputes[_disputeID].status;
    }

    function currentRuling(uint256 _disputeID) public view override returns (uint256␣
↪ruling) {
        ruling = disputes[_disputeID].ruling;
    }

    function giveRuling(uint256 _disputeID, uint256 _ruling) public {
        if (msg.sender != owner) {
            revert NotOwner();
        }

        Dispute storage dispute = disputes[_disputeID];

        if (_ruling > dispute.choices) {
            revert InvalidRuling(_ruling, dispute.choices);
        }
        if (dispute.status != DisputeStatus.Waiting) {
            revert InvalidStatus(dispute.status, DisputeStatus.Waiting);
        }

        dispute.ruling = _ruling;
        dispute.status = DisputeStatus.Appealable;
        dispute.appealPeriodStart = block.timestamp;
        dispute.appealPeriodEnd = dispute.appealPeriodStart + appealWindow;

        emit AppealPossible(_disputeID, dispute.arbitrated);
    }

    function executeRuling(uint256 _disputeID) public {
        Dispute storage dispute = disputes[_disputeID];
        if (dispute.status != DisputeStatus.Appealable) {
            revert InvalidStatus(dispute.status, DisputeStatus.Appealable);
        }

        if (block.timestamp <= dispute.appealPeriodEnd) {
            revert BeforeAppealPeriodEnd(block.timestamp, dispute.appealPeriodEnd);
        }

        dispute.status = DisputeStatus.Solved;
        dispute.arbitrated.rule(_disputeID, dispute.ruling);
    }

    function appeal(uint256 _disputeID, bytes memory _extraData) public payable␣
↪override {
        Dispute storage dispute = disputes[_disputeID];
        dispute.appealCount++;

        uint256 requiredAmount = appealCost(_disputeID, _extraData);
        if (msg.value < requiredAmount) {
            revert InsufficientPayment(msg.value, requiredAmount);
        }
```

(continues on next page)

(continued from previous page)

```
        if (dispute.status != DisputeStatus.Appealable) {
            revert InvalidStatus(dispute.status, DisputeStatus.Appealable);
        }

        if (block.timestamp > dispute.appealPeriodEnd) {
            revert AfterAppealPeriodEnd(block.timestamp, dispute.appealPeriodEnd);
        }

        dispute.status = DisputeStatus.Waiting;

        emit AppealDecision(_disputeID, dispute.arbitrated);
    }

    function appealPeriod(uint256 _disputeID) public view override returns (uint256 start, uint256 end) {
        Dispute storage dispute = disputes[_disputeID];

        return (dispute.appealPeriodStart, dispute.appealPeriodEnd);
    }
}
```

We implemented a setter for arbitration cost and we made the appeal cost as exponentially increasing. We achieved that by counting the number of appeals with `appealCount` variable, which gets increased each time `appeal` is executed.

This concludes our implementation of a centralized arbitrator with appeal functionality.

CHAPTER 10

Summary and Wrap-Up

In this tutorial, we explained ERC 792 and ERC 1497 standards, implemented arbitrable and arbitrator contracts following the standards and built a simple React application with `SimpleEscrowWithERC1497` we developed in the tutorials.

All the contracts we developed in this documentation can be found under the contracts directory and the source code of the React application resides under the src directory.

Important links:

- https://github.com/kleros/kleros
- https://github.com/kleros/kleros-interaction
- https://centralizedarbitrator.kleros.io/ You can use this to test your arbitrable contract against.

Any questions? Feel free to reach out to us on Slack or Discord .