▶

NFT AUCTION WITH REACHlang.

Implementing an NFT auction with REACH on Algorand and Ethereum.

1. ▶

Introduction.

What we are going to make.

1. ▶

Walk Through.

Let's summarize what we will be implementing.

1. A `Creator` will initialize the contract and provide three variables:
   - An NFT Token.
   - An initial bid.
   - A time limit.
2. Once these variables are provided, the `Creator` will then publish the contract onto the blockchain.
3. Thereafter, a `Bidder` will be able to connect to the contract and view the `token_id`, `initial_bid`, and `time_limit`.
4. If the `Bidder` accepts the wager, the `Bidder` will place a bid and call the backend.
5. The auction will continue until time-lapse hits.
6. At timeout :
   - The winner will receive the NFT.
   - The `Creator` will receive the highest bid.
   - All `Bidders` who lost the auction will receive their funds back.

> NOTE : The `Creator` is anyone who deploys the contract.

> The `Creator` is a participant class that can take any acceptable variable name.

-

▶

Implementing the Backend.

Let's see how we'll implement the reach backend.

1. ▶

Adding Reach Expressions.

Here we are going to add the various reach initialization options.

1.

▶

Creating a Reach App

Reach.App will contain all the code that we will need to create our contract.

> Let's add this into an `index.rsh` file.

```
'reach 0.1';

export const main = Reach.App(() => {
    //setoptions
})

init();
```

Let's go through the code to see what is happening.

- `reach 0.1;` indicates that this is a Reach program. You'll always have this at the top of every program.
- `export const main` defines the main export from the program. When you compile, this is what the compiler will look at.
- `init()` marks the deployment of the Reach program, which allows the program to start doing things.

-

▶

Adding a Participant

A Participant is a logical actor who takes part in a DApp and is associated with an account on the consensus network.

A **Participant** is a class that represent an account connected to the contract as well as a user connected to the frontend.

```
const Creator = Participant('Creator', {
        //Implement Creator interact interface here.
});
```

In this instance :

- We are creating a `Participant` class called `Creator`.

- The `Creator` will be the deployer of the contract onto the blockchain.

-
▶

Adding it all to `index.rsh`

Let's add what we have so far into `index.rsh`.

```
'reach 0.1';

export const main = Reach.App(() => {

    //++ Add Creator.
    const Creator = Participant('Creator', {
        //Implement Creator interact interface here.
    });

    init();
});
```

> Note that functions added onto the Participant can only be called by the backend.

-
▶

Adding a `Participant` Interface.

In the next step, we'll add the creator interface that will interact with the frontend.

- In order to implement the **Auction** the `Creator` will have to provide the following :

  > - An NFT token to be auctioned.
  > - A starting price for the auction.
  > - A duration for the auction.

- Once the `Creator` provides this information, any `Bidder` can view the deployed contract on the blockchain.

Let's add a function `getSale` in `index.rsh` that does just that.

1. The `Creator` will be responsible for providing NFT data from the frontend. So let's add this function to the Creators interface and call it `getSale()`.

   ```
   //++ Add getSale function.
   getSale: Fun([], Object({
       nftId: Token,
       minBid: UInt,
       lenInBlocks: UInt,
   })),
   ```

Let's decipher the `getSale()` function :

> - `Fun([], UInt)` is a Reach function that takes no arguments and returns a UInt.
> - `Object({nftId: Token,minBid: UInt,lenInBlocks: UInt,})` is a Reach object that has the following properties :
> - `nftId` is `Type` token.
> - `minBid` is `Type` UInt.
> - `lenInBlocks` is `Type` UInt.

- Therefore, the `getSale()` function will be called by the backend, and it will expect the frontend to return an `Object` with the following properties :
  - `nftId`.
  - `minBid`.
  - `lenInBlocks`.

2. Once the contract has been published onto the blockchain, we will need to notify the `Creator`'s frontend that the auction is ready to be deployed.

```
//++ Add auctionReady function.
auctionReady: Fun([], Null)
```

3. We also need to allow the Creator to see each bid in the auction.

   - SeeBid sends a `Bidder`.`Address` and the latest bid `UInt` to the frontend.

```
//++ Add seeBid function.
seeBid: Fun([Address, UInt], Null),
```

4. Finally, we will also allow the creator to see the outcome of the auction.

```
//++ Add showOutcome function.
seeOutcome: Fun([], Object({
    winner: Address,
    bid: UInt,
})),
```

> `SeeOutcome` sends the winner `Address` and the bid `UInt` to the frontend.

Let's add these function into the `index.rsh` file.

`index.rsh`

> Add this to index.rsh.

```
'reach 0.1';

export const main = Reach.App(() => {

    // Deployer of the contract.
    const Creator = Participant('Creator', {
        //++ Add getSale function.
        getSale: Fun([], Object({
            nftId: Token,
            minBid: UInt,
            lenInBlocks: UInt,
        })),
        //++ Add auctionReady function.
        auctionReady: Fun([], Null),

        //++ Add seeBid function.
        seeBid: Fun([Address, UInt], Null),

        //++ Add showOutcome function.
        showOutcome: Fun([Address, UInt], Null),
    });

    init();
});
```

-

▶

Adding a `Bidder` Interface.

The `Bidder` is an API that allows the frontend to interact with the backend.

> This is how the function looks.

```
//++ Add this function to the Bidder interface.

bid: Fun([UInt], Tuple(UInt,Address, UInt)),
```

Let's break down the `bid()` function :

- It takes in a `[UInt]` from the frontend, which is the bid amount.
- It returns a `Tuple(UInt,Address, UInt)` from the backend, which we will implement later.

-

▶

Adding it all into `index.rsh`

Adding the interfaces into the contract.

`index.rsh`

```
'reach 0.1';

export const main = Reach.App(() => {

    // Deployer of the contract.
    const Creator = Participant('Creator', {
        //getSale function.
        getSale: Fun([], Object({
            nftId: Token,
            minBid: UInt,
            lenInBlocks: UInt,
        })),
        //auctionReady function.
        auctionReady: Fun([], Null),

        //seeBid function.
        seeBid: Fun([Address, UInt], Null),

        //showOutcome function.
        showOutcome: Fun([Address, UInt], Null),
    });

    // Any subsequent bidder.
    const Bidder = API('Bidder', {
        //Bidder interface.
        bid: Fun([UInt], Tuple(UInt,Address, UInt)),
    });

    init();
});
```

-

▶

Working with Reach Steps.

1.

▶

Local Step

A local step refers to an action taken by a single `Participant` outside the blockchain.

Each reach program is in a local step after `initialization`.

Since we are building a nft-auction, we need a nft to be auctioned.

As described in the beginning, we will need :

- Nft Id
- Nft price
- Auction duration

All this information will be provided by the `Creator Participant`. To make sure that the `Creator` is the only one who can provide this information, we will use a `Local Step` to do so.

`Reach` provides us with an `only` method that we can use to do so.

```
Creator.only(() => {
    const {nftId, minBid, lenInBlocks} = declassify(interact.getSale());
});
```

Let's break it down:

- `Creator.only(() => {...})` is a `Local Step` that only allows the `Creator` to access the `getSale()` function we created above.
- `{nftId, minBid, lenInBlocks}` is the declassified `Object` that is returned from the `getSale()` function.
- The declassify function makes the return value known.
- The interact function notifies the frontend and awaits for a response.

Now that we have the `nftId`, `minBid`, and `lenInBlocks`, we can publish this information onto the contract.

> Let's add this to `index.rsh`.

```
'reach 0.1';

export const main = Reach.App(() => {

    // Deployer of the contract.
    const Creator = Participant('Creator', {
        //getSale function.
        getSale: Fun([], Object({
            nftId: Token,
            minBid: UInt,
            lenInBlocks: UInt,
        })),
        //auctionReady function.
        auctionReady: Fun([], Null),

        //seeBid function.
        seeBid: Fun([Address, UInt], Null),

        //showOutcome function.
        showOutcome: Fun([Address, UInt], Null),
    });
```

```
    // Any subsequent bidder.
    const Bidder = API('Bidder', {
        //Bidder interface.
        bid: Fun([UInt], Tuple(UInt,Address, UInt)),
    });

    init();

    //++ Add declassify function.
    Creator.only(() => {
        const {nftId, minBid, lenInBlocks} = declassify(interact.getSale());
    });
});
```

-

▶

Consensus Step

A consensus steps occurs on the blockchain network for all participants to see.

After the `init()` reach is always in a `local step`. In order to achieve consensus, we need to call consensus functions :

- Publish can be used to deploy information to the contract and will push the contract into a consensus state.
- Pay, which is paying fees to the contract will also push the contract into a consensus state.

Since we now know the `nftId`, `minBid`, and `lenInBlocks`, we can publish this information onto the contract.

```
Creator.publish(nftId, minBid, lenInBlocks);
```

In order to get back into a local step and allow the Creator to send the nft into the contract, we will use `commit` which pushes the reach into a local step.

We will also specify the number of tokens to send to the contract. We will set the amount to one since it is a unique nft, then pay it to the contract.

```
const amt = 1;

commit();

Creator.pay([[amt, nftId]]);

Creator.interact.auctionReady();
```

Then finally, we will `interact` with the frontend to notify the `Creator` that the auction is ready.

> This is how `index.rsh` looks like.

```
'reach 0.1';

export const main = Reach.App(() => {

    // Deployer of the contract.
    const Creator = Participant('Creator', {
        //getSale function.
        getSale: Fun([], Object({
            nftId: Token,
            minBid: UInt,
            lenInBlocks: UInt,
        })),
        //auctionReady function.
        auctionReady: Fun([], Null),

        //seeBid function.
        seeBid: Fun([Address, UInt], Null),

        //showOutcome function.
        showOutcome: Fun([Address, UInt], Null),
    });

    // Any subsequent bidder.
    const Bidder = API('Bidder', {
        //Bidder interface.
        bid: Fun([UInt], Tuple(UInt,Address, UInt)),
    });

    init();

    //declassify function.
    Creator.only(() => {
        const {nftId, minBid, lenInBlocks} = declassify(interact.getSale());
    });

    //++ Add publish contract.
    Creator.publish(nftId, minBid, lenInBlocks);

    //++ Add nft amount.
    const amt = 1;

    //++ Add step into local-step.
    commit();

    //++ Add send nft to contract.
    Creator.pay([[amt, nftId]]);

    //++ Add notify frontend that contract is ready.
    Creator.interact.auctionReady();
});
```

-

▶

Using Reach Checks

Here we will `assert` that the contract balance and consensus time has changed.

Reach provides various checks that we can use to check the current state of the contract.

We can use reach assert to check wether the `amt` we paid above has been reflected.

```
assert(balance(nftId) == amt, "balance of NFT is wrong");
```

- Here we are using a balance primitive to check the balance of the nft. if we call `balance()` without a passing a parameter, we will get the balance of the contract.

Also, we will check the last consensus time. Last consensus time checks the last time the contract was in consensus : The last time the contract used a `publish` or `pay` step.

```
const lastConsensus = lastConsensusTime();
```

- This is how we use the last consensus time primitive to check the last consensus time.

We can also set the length of the auction by taking the last consensus time and adding lenInBlocks to it.

```
const end = lastConsensus + lenInBlocks;
```

-

▶

Adding it all into `index.rsh`

This is how your `index.rsh` should look like.

```
'reach 0.1';

export const main = Reach.App(() => {

    // Deployer of the contract.
    const Creator = Participant('Creator', {
        //getSale function.
        getSale: Fun([], Object({
            nftId: Token,
            minBid: UInt,
            lenInBlocks: UInt,
```

```
        })),
        //auctionReady function.
        auctionReady: Fun([], Null),

        //seeBid function.
        seeBid: Fun([Address, UInt], Null),

        //showOutcome function.
        showOutcome: Fun([Address, UInt], Null),
    });

    // Any subsequent bidder.
    const Bidder = API('Bidder', {
        //Bidder interface.
        bid: Fun([UInt], Tuple(UInt,Address, UInt)),
    });

    init();

    //declassify function.
    Creator.only(() => {
        const {nftId, minBid, lenInBlocks} = declassify(interact.getSale());
    });

    //publish contract.
    Creator.publish(nftId, minBid, lenInBlocks);

    //nft amount.
    const amt = 1;

    //step into local-step.
    commit();

    //send nft to contract.
    Creator.pay([[amt, nftId]]);

    //notify frontend that contract is ready.
    Creator.interact.auctionReady();

    //++ Add assertion to check nft balance
    assert(balance(nftId) == amt, "balance of NFT is wrong");

    //++ Add checkpoint to set last publish time.
    const lastConsensus = lastConsensusTime();

    //++ Add blocktime to set auction duration.
    const end = lastConsensus + lenInBlocks;
});
```

-

▶

Adding Parallel Reduce.

Here we implement a parallel reduce to run the auction until auction time runs out.

1. All `Bidder`s will be competing against each other to make the highest bid while simultaneously racing against the auction time.
2. We will use a `while` loop that keeps the auction active as long as the auction time is not over.
3. Every time a bidder bids higher than the previous bid price, the previous bidder will be reimbursed.
4. At the end, the parallel reduce will force a single result.

Let's see how this will look.

1. ▶

Adding parallel reduce.

We first create a list that will be used in the parallel reduce.

```
const [highestBidder, lastPrice, isFirstBid] = [0, 0, 0];
```

- Every round of the loop, we will be checking and setting the highest bid, the highest bidder address and whether it is the first bid.

> Since the `Creator` will be the first bidder, we will set the `highestBidder` to the `Creator` address. Set the `lastPrice` to the `minBid` and `isFirstBid` to `true`.

```
const [highestBidder, lastPrice, isFirstBid] = [Creator, minBid, true];
```

> Now let's plug this into the `parallelReduce` function.

```
const [highestBidder, lastPrice, isFirstBid] = parallelReduce([Creator, minBid,
true])
```

-

▶

Adding an Invariant

A while loop can execute a block of code as long as a specified condition is true. Thus, the invariant value should be a `true` value that is set at the start of a loop and changes only when the auction is done.

```
const [highestBidder, lastPrice, isFirstBid] = parallelReduce([Creator, minBid,
true])
```

```
    .invariant(balance(nftId) == amt && balance() == (isFirstBid ? 0 : lastPrice))
```

- Here, the invariant is true as long as the balance of the NFT is equal to one, thus the contract still holds the nft.
- It also checks whether it is the first bid or not. If so then the contract balance is 0, otherwise the contract balance is equal to the last bid price.

-

▶

Using a while loop.

A while loop will run until the last consensus time is less than the end time.

```
const [highestBidder, lastPrice, isFirstBid] = parallelReduce([Creator, minBid,
true])
    .invariant(balance(nftId) == amt && balance() == (isFirstBid ? 0 : lastPrice))
    .while(lastConsensusTime() < end)
```

While the loop is `true`, let's accept bids. Parallel reduce uses `components` to allow `participants` and `api`'s to individually access functions.

-

▶

Using an `API`

Here, we use `.api()` to allow bidders to place bids.

- An `API_EXPR` is used to access the `Bidder` API `bid` function.

```
.api(Bidder.bid ....
```

- An [`ASSUME_EXPR`] evaluates a claim that resolves to true.

```
.api(Bidder.bid,
((bid) => { assume(bid > lastPrice, "bid is too low"); }),
```

> Here we are testing whether the bid is higher than the last price.

- `PAY_EXPR` is used to pay the wager to the contract.

```
.api(Bidder.bid,
((bid) => { assume(bid > lastPrice, "bid is too low"); })
```

```
((bid) => bid),
```

- `CONSENSUS_EXPR` is used to update the consensus state of the contract to notify the bidder of the bid.

```
.api(Bidder.bid,
    ((bid) => { assume(bid > lastPrice, "bid is too low"); }),
    ((bid) => bid),
    ((bid, notify) => {
        require(bid > lastPrice, "bid is too low");
        notify([bid,highestBidder, lastPrice]);
        if ( ! isFirstBid ) {
            transfer(lastPrice).to(highestBidder);
        }
        Creator.interact.seeBid(this, bid);
        return [this, bid, false];
    })
)
```

- Here we are using require to ensure that the bid is higher than the last placed bid.
- We will `notify` the bidder frontend of the `bid` placed, the `highestBidder` and the `lastPrice`.
- We are checking if `isFirstBid` is `false`. If it is, we will reimburse the `lastPrice` back to the last bidder.
- We are also interaction with the `Creator` frontend to notify it of the bid.
- We finally return the `bidder`, the `bid` and setting `isFirstBid` to false.

-

▶

Setting auction timeout.

Reach `timeout` will be called once the auction time reaches. `timeout` takes a parameter `blocktime` and a function once the timeout is reached.

```
.timeout(absoluteTime(end), () => {
    Creator.publish()
    return [highestBidder, lastPrice, isFirstBid];
});
```

- absoluteTime gets the absolute time of the blockchain.
- Once the auction time ends, the `Creator` will `publish` the information onto the blockchain and returns the `highestBidder`, `lastPrice` and `isFirstBid`.

This is how the full parallel reduce looks.

-

▶

Putting the auction together.

```
const [highestBidder, lastPrice, isFirstBid] = parallelReduce([Creator, minBid,
true])
.invariant(balance(nftId) == amt && balance() == (isFirstBid ? 0 : lastPrice))
.while(lastConsensusTime() < end)
.api(Bidder.bid,
((bid) => { assume(bid > lastPrice, "bid is too low"); }),
((bid) => bid),
((bid, notify) => {
    require(bid > lastPrice, "bid is too low");
    notify([bid,highestBidder, lastPrice]);
    if ( ! isFirstBid ) {
        transfer(lastPrice).to(highestBidder);
    }
    Creator.interact.seeBid(this, bid);
    return [this, bid, false];
})
).timeout(absoluteTime(end), () => {
    Creator.publish()
    return [highestBidder, lastPrice, isFirstBid];
});
```

-

▶

Adding it all into `index.rsh`.

This is how your `index.rsh` should be looking like.

```
'reach 0.1';

export const main = Reach.App(() => {

    // Deployer of the contract.
    const Creator = Participant('Creator', {
        //getSale function.
        getSale: Fun([], Object({
            nftId: Token,
            minBid: UInt,
            lenInBlocks: UInt,
        })),
        //auctionReady function.
        auctionReady: Fun([], Null),
```

```
        //seeBid function.
        seeBid: Fun([Address, UInt], Null),

        //showOutcome function.
        showOutcome: Fun([Address, UInt], Null),
    });

    // Any subsequent bidder.
    const Bidder = API('Bidder', {
        //Bidder interface.
        bid: Fun([UInt], Tuple(UInt,Address, UInt)),
    });

    init();

    //declassify function.
    Creator.only(() => {
        const {nftId, minBid, lenInBlocks} = declassify(interact.getSale());
    });

    //publish contract.
    Creator.publish(nftId, minBid, lenInBlocks);

    //nft amount.
    const amt = 1;

    //step into local-step.
    commit();

    //send nft to contract.
    Creator.pay([[amt, nftId]]);

    //notify frontend that contract is ready.
    Creator.interact.auctionReady();

    // assertion to check nft balance
    assert(balance(nftId) == amt, "balance of NFT is wrong");

    // checkpoint to set last publish time.
    const lastConsensus = lastConsensusTime();

    // blocktime to set auction duration.
    const end = lastConsensus + lenInBlocks;

    //++ Add parallel reduce
    const [highestBidder, lastPrice, isFirstBid] = parallelReduce([Creator,
minBid, true])
    .invariant(balance(nftId) == amt && balance() == (isFirstBid ? 0 : lastPrice))
    .while(lastConsensusTime() < end)
    .api(Bidder.bid,
    ((bid) => { assume(bid > lastPrice, "bid is too low"); }),
    ((bid) => bid),
    ((bid, notify) => {
```

```
        require(bid > lastPrice, "bid is too low");
        notify([bid,highestBidder, lastPrice]);
        if ( ! isFirstBid ) {
            transfer(lastPrice).to(highestBidder);
        }
        Creator.interact.seeBid(this, bid);
        return [this, bid, false];
    })
    ).timeout(absoluteTime(end), () => {
        Creator.publish()
        return [highestBidder, lastPrice, isFirstBid];
    });
});
```

-

▶

Setting up onwership Transfer

Transferring the NFT to the winner of the auction.

Transfer is a consensus step that transfers ownership of contract tokens.

After the contract has determined the winner of the auction, we transfer the NFT to the winner.

```
transfer(amt, nftId).to(highestBidder);
```

Then we transfer the highest bid, to the `Creator` of the nft.

```
if ( ! isFirstBid ) { transfer(lastPrice).to(Creator); }
```

Finally, we notify the `Creator` frontend of the auction results.

```
Creator.interact.showOutcome(highestBidder, lastPrice);
```

`commit` back to a local state and `exit` the contract.

```
commit();

exit();
```

-

▶

Here's the complete Backend

This is how your final `index.rsh` should be looking like.

```
'reach 0.1';

export const main = Reach.App(() => {

    // Deployer of the contract.
    const Creator = Participant('Creator', {
        //getSale function.
        getSale: Fun([], Object({
            nftId: Token,
            minBid: UInt,
            lenInBlocks: UInt,
        })),
        //auctionReady function.
        auctionReady: Fun([], Null),

        //seeBid function.
        seeBid: Fun([Address, UInt], Null),

        //showOutcome function.
        showOutcome: Fun([Address, UInt], Null),
    });

    // Any subsequent bidder.
    const Bidder = API('Bidder', {
        //Bidder interface.
        bid: Fun([UInt], Tuple(UInt,Address, UInt)),
    });

    init();

    //declassify function.
    Creator.only(() => {
        const {nftId, minBid, lenInBlocks} = declassify(interact.getSale());
    });

    //publish contract.
    Creator.publish(nftId, minBid, lenInBlocks);

    //nft amount.
    const amt = 1;

    //step into local-step.
    commit();

    //send nft to contract.
    Creator.pay([[amt, nftId]]);

    //notify frontend that contract is ready.
    Creator.interact.auctionReady();
```

```
    // assertion to check nft balance
    assert(balance(nftId) == amt, "balance of NFT is wrong");

    // checkpoint to set last publish time.
    const lastConsensus = lastConsensusTime();

    // blocktime to set auction duration.
    const end = lastConsensus + lenInBlocks;

    // parallel reduce
    const [highestBidder, lastPrice, isFirstBid] = parallelReduce([Creator,
minBid, true])
    .invariant(balance(nftId) == amt && balance() == (isFirstBid ? 0 : lastPrice))
    .while(lastConsensusTime() < end)
    .api(Bidder.bid,
    ((bid) => { assume(bid > lastPrice, "bid is too low"); }),
    ((bid) => bid),
    ((bid, notify) => {
        require(bid > lastPrice, "bid is too low");
        notify([bid,highestBidder, lastPrice]);
        if ( ! isFirstBid ) {
            transfer(lastPrice).to(highestBidder);
        }
        Creator.interact.seeBid(this, bid);
        return [this, bid, false];
    })
    ).timeout(absoluteTime(end), () => {
        Creator.publish()
        return [highestBidder, lastPrice, isFirstBid];
    });

    // Transfer
    if ( ! isFirstBid ) { transfer(lastPrice).to(Creator); }

    // creator show outcome.
    Creator.interact.showOutcome(highestBidder, lastPrice);

    // step to local-step.
    commit();

    // exit contract.
    exit();
});
```

-

▶

Implementing the Frontend.

Let's see how we'll connect the backend the frontend.

1. ▶

Importing the dependencies.

We need to import the Reach Standard Library module for JavaScript.

```
import { loadStdlib } from '@reach-sh/stdlib';
```

> `loadStdlib` is a function that will load the standard library dynamically based on the `REACH_CONNECTOR_MODE` environment variable.

> You can also pass in a `REACH_CONNECTOR_MODE` variable directly to `loadStdlib` if you want to override the default.

```
// connector can be 'ETH', 'ALGO', or 'CFX'
const stdlib = await loadStdlib("ALGO");
```

We also need to import the backend.

- Once we run :

```
./reach compile
```

Reach will transpile the `index.rsh` file to `index.main.mjs` and output it to `build/index.main.mjs`. The `index.main.mjs` file will contain all the code we need to interact with our backend contract. We can now import `index.main.mjs` into our application

```
import * as backend from './build/index.main.mjs';
```

-

▶

Adding code to `index.mjs`.

Let's add what we have done so far into the `index.mjs`.

> This is how it looks.

```
//++ Add Import reach stdlib
import { loadStdlib } from '@reach-sh/stdlib';

//++ Add Import contract backend
import * as backend from './build/index.main.mjs';
```

```
//++ Add Load stdlib
const stdlib = loadStdlib();
```

-

▶

Adding a `Creator Participant` Test Account.

Let's add a test account to our `index.mjs` file.

We will use reach standard library to create a test account with a starting balance of 100 network tokens.

```
//++Add generate starting balance
const startingBalance = stdlib.parseCurrency(100);

//++Add create test account
const accCreator = await stdlib.newTestAccount(startingBalance);
```

-

▶

Creating a nft with launchtoken

Adding an nft to our `index.mjs` file.

If we take a look at `index.rsh` we see that the `Creator.getSale` function expects a `nftId`, a `minBid` and `lenInBlocks` as parameters.

> Reach Standard Library provides a `launchToken` function that can handle creating a network token.

```
const theNFT = await stdlib.launchToken(accCreator, "bumple", "NFT", { supply: 1
});
```

Let's decipher the parameters :

- `Account` = `launchToken` expects the account of the creator of the token. In our instance, `accCreator` is the creator of the token.
- `name` = `launchToken` expects the name of the token. In our instance, `bumple` is the name of the token.
- `sym` = `launchToken` expects the symbol of the token. In our instance, `NFT` is the symbol of the token.
- `opts` = `launchToken` expects an object of options if any. In our instance, `{ supply: 1 }` is the option since we only require unique instance of the NFT.

-

▶

Connecting the `Creator Participant` to the Backend.

Let's see how to connect the `Creator Participant` to the backend and add it into our `index.mjs`.

1. ▶

Connecting the test account to the backend.

Now we will connect the test account to the backend.

```
//++ Add connect account to backend contract.
const ctcCreator = accCreator.contract(backend);
```

> `accCreator.contract(backend);` returns a ***Reach Contract*** that contains the contract address.

-

▶

Connecting to the Interface.

We can now connect to the backend `Creator` interface with :

```
//++ Add setting up the `Creator` interface.
await ctcCreator.participants.Creator({
    // Specify Creator interact interface here
})
```

> `await ctcCreator.participants.Creator` will connect the backend `Creator` interface with the `accCreator`.

> Before we do that, we need to implement the `Creator` interface that we defined in `index.rsh`.

-

▶

Implementing the `getSale` function.

`getSale` function requires three parameters : `nftId`, `minBid` and `lenInBlocks`.

```
//++ Add nft params expected by the `getSale` function.
const nftId = theNFT.id
const minBid = stdlib.parseCurrency(2);
lenInBlocks = 10;
```

- We are getting the `nftId` from the NFT we created earlier.
- The minimum bid is 2 network tokens.
- The number of blocks before the auction ends is 10.

```
```

```
//++ Add putting them in an object.
const params = {
nftId:nftId,
minBid:minBid,
lenInBlocks:lenInBlocks,
};
```

> Since the `getSale` function expects an object, we need to create an object with the parameters.

-

▶

Adding `getSale` to the interface.

Let's add the `params` object to the `Creator` interface.

```
//++ Add setting up the `Creator` interface.
await ctcCreator.participants.Creator({
    // ++ Add get sale function.
    getSale: () => {
        return params;
    },
})
```

-

▶

Adding `seeBid` function to the frontend.

Connecting the `Creator Participant` to the frontend.

Ass you recall, the `seeBid` function from the `backend` sends an `Address` and a `UInt` to the frontend.

```
await ctcCreator.participants.Creator({
    // ++ Add get sale function.
    getSale: () => {
        return params;
    },
    // ++ Add seeBid function.
    seeBid: (who, amt) => {
        let newBidder = stdlib.formatAddress(who)
        let newBid = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newBidder} bid ${newBid}.`);
    },
})
```

-

▶

Adding the `showOutcome` function to the frontend.

Connecting the `Creator Participant` to the frontend.

The `showOutcome` function will notify the frontend, when the contract is ready to begin the auction.

```
await ctcCreator.participants.Creator({
    // ++ Add get sale function.
    getSale: () => {
        return params;
    },
    // ++ Add seeBid function.
    seeBid: (who, amt) => {
        let newBidder = stdlib.formatAddress(who)
        let newBid = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newBidder} bid ${newBid}.`);
    },
    // ++ Add showOutcome function.
    showOutcome: (winner, amt) => {
        let newWinner = stdlib.formatAddress(winner)
        let newAmt = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newWinner} won with ${newAmt}`)
    }
})
```

-

▶

Summing it all up.

Adding it all to `index.mjs`.

Adding it all up, this is how the `index.rhs` interface looks.

```
// Import reach stdlib
import { loadStdlib } from '@reach-sh/stdlib';

// Import contract backend
import * as backend from './build/index.main.mjs';

// Load stdlib
const stdlib = loadStdlib();

// generate starting balance
const startingBalance = stdlib.parseCurrency(100);

// create test account
```

```
const accCreator = await stdlib.newTestAccount(startingBalance);

// nft asset.
const theNFT = await stdlib.launchToken(accCreator, "bumple", "NFT", { supply: 1
});

//++ Add connect account to backend contract.
const ctcCreator = accCreator.contract(backend);

//++ Add nft params expected by the `getSale` function.
const nftId = theNFT.id
const minBid = stdlib.parseCurrency(2);
lenInBlocks = 10;

//++ Add putting them in an object.
const params = {
    nftId:nftId,
    minBid:minBid,
    lenInBlocks:lenInBlocks,
};

//++ Add setting up the `Creator` interface.
await ctcCreator.participants.Creator({
    // ++ Add get sale function.
    getSale: () => {
        return params;
    },
    // ++ Add seeBid function.
    seeBid: (who, amt) => {
        let newBidder = stdlib.formatAddress(who)
        let newBid = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newBidder} bid ${newBid}.`);
    },
    // ++ Add showOutcome function.
    showOutcome: (winner, amt) => {
        let newWinner = stdlib.formatAddress(winner)
        let newAmt = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newWinner} won with ${newAmt}`)
    }
})
```

-

▶

Adding a `Bidder` Test Account.

This how a bidder test account will look like.

Let's create a test account for the `Bidder api` just as we did with the `Creator`.

```
// ++ Add test currrency.
const startingBalance = stdlib.parseCurrency(100);
```

```
// create test account
const accBidder = await stdlib.newTestAccount(startingBalance);
```

-

▶

Connecting the `Bidder API` to the Backend.

This is how the `Bidder` will interact with the contract.

1. ▶

Connecting to the Contract.

Let's connect the `Bidder` to the backend.

In order to connect the `Bidder API` to the backend, we need to get the contract `address` that was created by the `Creator` :

```
// remember this line
const ctcCreator = accCreator.contract(backend);
```

> Reach provides a `ctc.getInfo` function that returns the contract address.

```
const ctc = accBidder.contract(backend, ctcCreator.getInfo());
```

- Here we are calling the `accBidder.contract` function and passing the backend and contract address.

-

▶

Accepting the token.

The `Bidder` will have to accept the token in order transact with the contract.

The `Bidder` must also allow their account to accept the NFT Token. Reach provides a `tokenAccept` function that does just that.

```
await acc.tokenAccept(nftId);
```

- Here we are calling the `tokenAccept` function and passing the `nftId` of the token.

-

▶

Adding A Bidder Interface.

We are now ready to add a `Bidder` interface to the frontend to test the auction.

-
- ▶

Adding an Auction Function.

Creating test bidders.

We are going to put all our `Bidders` into an `async` function and allow each `Bidder` to connect to the backend contract. But before we do that, let's look at how an actor other than the `Creator`/Deployer connects to the backend contract.

```
let done = false;
const bidders = [];
const startBidders = async () => {
    let bid = minBid;
    const runBidder = async (who) => {
        const inc = stdlib.parseCurrency(Math.random() * 10);
        bid = bid.add(inc);

        const accBidder = await stdlib.newTestAccount(startingBalance);
        accBidder.setDebugLabel(who);

        await accBidder.tokenAccept(nftId);
        bidders.push([who, accBidder]);
        const ctc = accBidder.contract(backend, ctcCreator.getInfo());
        const getBal = async () => stdlib.formatCurrency(await
stdlib.balanceOf(accBidder));

        console.log(`${who} decides to bid ${stdlib.formatCurrency(bid)}.`);
        console.log(`${who} balance before is ${await getBal()}`);
        try {
            const [ latestBid,lastBidder, lastBid ] = await
ctc.apis.Bidder.bid(bid);
            console.log(`${who} out bid ${lastBidder} who bid
${stdlib.formatCurrency(lastBid)}. with ${stdlib.formatCurrency(latestBid)}`);
        } catch (e) {
            console.log(`${who} failed to bid, because ${e} is too high`);
        }
        console.log(`${who} balance after is ${await getBal()}`);
    };

    await runBidder('Alice');
    await runBidder('Bob');
    await runBidder('Claire');
    while ( ! done ) {
        await stdlib.wait(1);
    }
};
```

- `let done = false;` will be used to call wait on the contract until the auction is over.
- `const bidders = [];`
- `const startBidders` will be called by the Creator once the auction is ready.
- `let bid = minBid;`
- `const runBidder()`
- `const inc = stdlib.parseCurrency(Math.random() * 10);` uses reach to generate a random number between 0 and 10.
- `bid = bid.add(inc);` adds the random number to the current bid to create a unique bid for each `Bidder`.
- `const accBidder = await stdlib.newTestAccount(startingBalance);` creates a new account for the `Bidder`.
- `accBidder.setDebugLabel(who);` sets the debug label for the `Bidder`, with a unique `Bidder` name.
- `await accBidder.tokenAccept(nftId);` allows the `Bidder` accepts the NFT from the Creator.
- `bidders.push([who, accBidder]);` adds the `Bidder` name and `Bidder` account to the `const bidders = [];` array we created.
- `const ctc = accBidder.contract(backend, ctcCreator.getInfo());` connects the `Bidder` to the contract deployed by the `Creator` by using reach standard library function `getInfo()`.
- `const getBal = async () => stdlib.formatCurrency(await stdlib.balanceOf(accBidder));` gets `Bidder` balance from the `Bidder` account.
- `console.log("${who} decides to bid ${stdlib.formatCurrency(bid)}.");` prints the `Bidder` name and the bid they are going to make.
- `console.log("${who} balance before is ${await getBal()}");` prints the `Bidder` name and the balance before the bid.
- `try {` we will use a try statement because the `backend Bidder.bid` function checks whether the bid is larger than the last bid and returns an error if it's not larger.

> Backend `javascript assume(bid > lastPrice, "bid is too low"); require(bid > lastPrice, "bid is too low");`

- `const [ latestBid,lastBidder, lastBid ] = await ctc.apis.Bidder.bid(bid);` calls the `backend Bidder.bid` function and `await`s the `latestBid`, `lastBidder`, and the `lastBid` from the backend.

> Backend `javascript ((bid, notify) => { require(bid > lastPrice, "bid is too low"); notify([bid,highestBidder, lastPrice]); if ( ! isFirstBid ) { transfer(lastPrice).to(highestBidder); } Creator.interact.seeBid(this, bid); return [this, bid, false]; })`

- `console.log("${who} out bid ${lastBidder} who bid ${stdlib.formatCurrency(lastBid)}.");` prints the `Bidder` name and the `Bidder` name of the last `Bidder` who bid.
- `console.log("${who} failed to bid, because ${e} is too high");`. If the bid is to low, the `try` statement will catch the error from the backend.
- `console.log("${who} balance after is ${await getBal()}");` prints the `Bidder` name and the balance after the bid.

To test the auction, let's add three `Bidder`s, **Alice**, **Bob**, and **Claire**.

```
    await runBidder('Alice');
    await runBidder('Bob');
    await runBidder('Claire');
```

-

▶

Running the Auction

How will we run the auction ?

Remember the creator interface, we are going to add the `startBidders` function onto the `Creator.auctionReady` function so that once the auction is ready, we can start the auction.

```
await ctcCreator.participants.Creator({
    // ++ Add get sale function.
    getSale: () => {
        return params;
    },
    // ++ Add seeBid function.
    seeBid: (who, amt) => {
        let newBidder = stdlib.formatAddress(who)
        let newBid = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newBidder} bid ${newBid}.`);
    },
    // ++ Add showOutcome function.
    showOutcome: (winner, amt) => {
        let newWinner = stdlib.formatAddress(winner)
        let newAmt = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newWinner} won with ${newAmt}`)
    },
    // ++ Add startBidders function.
    auctionReady: () => {
        console.log("Creator sees that the auction is ready.");
        startBidders();
    }
})
```

▶

Adding A Bidder Interface.

We are now ready to add a `Bidder` interface to the frontend to test the auction.

-

▶

Adding an Auction Function.

Creating test bidders.

We are going to put all our `Bidders` into an `async` function and allow each `Bidder` to connect to the backend contract. But before we do that, let's look at how an actor other than the `Creator`/Deployer connects to the backend contract.

```
let done = false;
const bidders = [];
const startBidders = async () => {
    let bid = minBid;
    const runBidder = async (who) => {
        const inc = stdlib.parseCurrency(Math.random() * 10);
        bid = bid.add(inc);

        const accBidder = await stdlib.newTestAccount(startingBalance);
        accBidder.setDebugLabel(who);

        await accBidder.tokenAccept(nftId);
        bidders.push([who, accBidder]);
        const ctc = accBidder.contract(backend, ctcCreator.getInfo());
        const getBal = async () => stdlib.formatCurrency(await
stdlib.balanceOf(accBidder));

        console.log(`${who} decides to bid ${stdlib.formatCurrency(bid)}.`);
        console.log(`${who} balance before is ${await getBal()}`);
        try {
            const [ latestBid,lastBidder, lastBid ] = await
ctc.apis.Bidder.bid(bid);
            console.log(`${who} out bid ${lastBidder} who bid
${stdlib.formatCurrency(lastBid)}. with ${stdlib.formatCurrency(latestBid)}`);
        } catch (e) {
            console.log(`${who} failed to bid, because ${e} is too high`);
        }
        console.log(`${who} balance after is ${await getBal()}`);
    };

    await runBidder('Alice');
    await runBidder('Bob');
    await runBidder('Claire');
    while ( ! done ) {
```

```
            await stdlib.wait(1);
        }
    };
```

- `let done = false;` will be used to call wait on the contract until the auction is over.
- `const bidders = [];`
- `const startBidders` will be called by the Creator once the auction is ready.
- `let bid = minBid;`
- `const runBidder()`
- `const inc = stdlib.parseCurrency(Math.random() * 10);` uses reach to generate a random number between 0 and 10.
- `bid = bid.add(inc);` adds the random number to the current bid to create a unique bid for each `Bidder`.
- `const accBidder = await stdlib.newTestAccount(startingBalance);` creates a new account for the `Bidder`.
- `accBidder.setDebugLabel(who);` sets the debug label for the `Bidder`, with a unique `Bidder` name.
- `await accBidder.tokenAccept(nftId);` allows the `Bidder` accepts the NFT from the Creator.
- `bidders.push([who, accBidder]);` adds the `Bidder` name and `Bidder` account to the `const bidders = [];` array we created.
- `const ctc = accBidder.contract(backend, ctcCreator.getInfo());` connects the `Bidder` to the contract deployed by the `Creator` by using reach standard library function `getInfo()`.
- `const getBal = async () => stdlib.formatCurrency(await stdlib.balanceOf(accBidder));` gets `Bidder` balance from the `Bidder` account.
- `console.log("${who} decides to bid ${stdlib.formatCurrency(bid)}.");` prints the `Bidder` name and the bid they are going to make.
- `console.log("${who} balance before is ${await getBal()}");` prints the `Bidder` name and the balance before the bid.
- `try {` we will use a try statement because the `backend Bidder.bid` function checks whether the bid is larger than the last bid and returns an error if it's not larger.

> Backend `javascript assume(bid > lastPrice, "bid is too low"); require(bid > lastPrice, "bid is too low");`

- `const [ latestBid,lastBidder, lastBid ] = await ctc.apis.Bidder.bid(bid);` calls the `backend Bidder.bid` function and `await`s the `latestBid`, `lastBidder`, and the `lastBid` from the backend.

> Backend `javascript ((bid, notify) => { require(bid > lastPrice, "bid is too low"); notify([bid,highestBidder, lastPrice]); if ( ! isFirstBid ) { transfer(lastPrice).to(highestBidder); } Creator.interact.seeBid(this, bid); return [this, bid, false]; })`

- `console.log("${who} out bid ${lastBidder} who bid ${stdlib.formatCurrency(lastBid)}.");` prints the `Bidder` name and the `Bidder` name of the last `Bidder` who bid.
- `console.log("${who} failed to bid, because ${e} is too high");`. If the bid is to low, the `try` statement will catch the error from the backend.
- `console.log("${who} balance after is ${await getBal()}");` prints the `Bidder` name and the balance after the bid.

To test the auction, let's add three `Bidder`s, **Alice**, **Bob**, and **Claire**.

```
    await runBidder('Alice');
    await runBidder('Bob');
    await runBidder('Claire');
```

-

▶

Running the Auction

How will we run the auction ?

Remember the creator interface, we are going to add the `startBidders` function onto the `Creator.auctionReady` function so that once the auction is ready, we can start the auction.

```
await ctcCreator.participants.Creator({
    // ++ Add get sale function.
    getSale: () => {
        return params;
    },
    // ++ Add seeBid function.
    seeBid: (who, amt) => {
        let newBidder = stdlib.formatAddress(who)
        let newBid = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newBidder} bid ${newBid}.`);
    },
    // ++ Add showOutcome function.
    showOutcome: (winner, amt) => {
        let newWinner = stdlib.formatAddress(winner)
        let newAmt = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newWinner} won with ${newAmt}`)
    },
    // ++ Add startBidders function.
```

```
    auctionReady: () => {
        console.log("Creator sees that the auction is ready.");
        startBidders();
    }
})
```

-

▶

Adding it all up.

Let's add what we have done so far into an `index.mjs`.

We have covered alot, but you don't have to understand everything. Let's try to run the auction and see what happens.

```
// Import reach stdlib
import { loadStdlib } from '@reach-sh/stdlib';

// Import contract backend
import * as backend from './build/index.main.mjs';

// Load stdlib
const stdlib = loadStdlib();

// generate starting balance
const startingBalance = stdlib.parseCurrency(100);

// create test account
const accCreator = await stdlib.newTestAccount(startingBalance);

// nft asset.
const theNFT = await stdlib.launchToken(accCreator, "bumple", "NFT", { supply: 1
});

// connect account to backend contract.
const ctcCreator = accCreator.contract(backend);

// nft params expected by the `getSale` function.
const nftId = theNFT.id
const minBid = stdlib.parseCurrency(2);
lenInBlocks = 10;

// putting them in an object.
const params = {
    nftId:nftId,
    minBid:minBid,
    lenInBlocks:lenInBlocks,
};

//++ Add Bidder Interface.
```

```
let done = false;
const bidders = [];
const startBidders = async () => {
    let bid = minBid;
    const runBidder = async (who) => {
        const inc = stdlib.parseCurrency(Math.random() * 10);
        bid = bid.add(inc);

        const accBidder = await stdlib.newTestAccount(startingBalance);
        accBidder.setDebugLabel(who);

        await accBidder.tokenAccept(nftId);
        bidders.push([who, accBidder]);
        const ctc = accBidder.contract(backend, ctcCreator.getInfo());
        const getBal = async () => stdlib.formatCurrency(await
stdlib.balanceOf(accBidder));

        console.log(`${who} decides to bid ${stdlib.formatCurrency(bid)}.`);
        console.log(`${who} balance before is ${await getBal()}`);
        try {
            const [ latestBid,lastBidder, lastBid ] = await
ctc.apis.Bidder.bid(bid);
            console.log(`${who} out bid ${lastBidder} who bid
${stdlib.formatCurrency(lastBid)}. with ${stdlib.formatCurrency(latestBid)}`);
        } catch (e) {
            console.log(`${who} failed to bid, because ${e} is too high`);
        }
        console.log(`${who} balance after is ${await getBal()}`);
    };

    await runBidder('Alice');
    await runBidder('Bob');
    await runBidder('Claire');
    while ( ! done ) {
        await stdlib.wait(1);
    }
};

// setting up the `Creator` interface.
await ctcCreator.participants.Creator({
    //  get sale function.
    getSale: () => {
        return params;
    },
    //  seeBid function.
    seeBid: (who, amt) => {
        let newBidder = stdlib.formatAddress(who)
        let newBid = stdlib.formatCurrency(amt)
        console.log(`Creator saw that ${newBidder} bid ${newBid}.`);
    },
    //  showOutcome function.
    showOutcome: (winner, amt) => {
        let newWinner = stdlib.formatAddress(winner)
        let newAmt = stdlib.formatCurrency(amt)
```

```
        console.log(`Creator saw that ${newWinner} won with ${newAmt}`)
    },
    // ++ Add startBidders function.
    auctionReady: () => {
        console.log("Creator sees that the auction is ready.");
        startBidders();
    }
})
```