▶

Introduction.

Welcome to my NFT auction workshop! This workshop will teach you the basics of programming a NFT auction using the reach programming language. Reach is a powerful and easy to use programming language that is perfect for creating NFT auctions. This workshop will cover the following topics:

1. Common Terminologies.
2. Define the use case and requirements for the DApp.
3. Choose a suitable blockchain platform on which to build the DApp.
4. Develop the DApp using the chosen blockchain platform's programming language and tools.
5. Test the DApp to ensure it works as intended.
6. Deploy the DApp on the chosen blockchain platform.
7. Promote the DApp to users and encourage them to use it.

I hope you enjoy this workshop!

In this workshop, we will be introducing the NFT auction workshop and the reach programming language.

> The purpose of the workshop is to walk you through the thought process that went into developing the DApp. If you are looking for a step-by-step approach, please refer to the NFT Auction Tutorial.

Here are some of the key concepts that we will be repeating throughout the workshop:

What is Reach Programming Language

The reach programming language is a language specifically designed for writing smart contracts on the blockchain. It is a statically typed, functional programming language that can be used on multiple blockchains with a focus on safety and security.

What is an auction?

An auction is a process of buying and selling goods or services by offering them up for bid, taking bids, and then selling the item to the highest bidder.

What is a NFT?

a NFT is a non-fungible token. This means that each NFT is unique and not interchangeable with any other NFT.

What is a DApp?

A DApp is a type of decentralized application that is governed by a set of rules encoded on the blockchain.

What is a smart contract?

A smart contract is a computer program that runs on the blockchain and automatically executes transactions when certain conditions are met.

What is a token?

A token is a digital asset that is used to represent a stake in a decentralized application. Tokens can be used to purchase goods and services, or to participate in governance.

What is a blockchain?

A blockchain is a distributed database that is used to store data in a secure and tamper-proof way. Blockchains are used to power decentralized applications.

▶

Defining the Use Case and Requirements.

-

Use Cases

A NFT auction is a mechanism for exchanging NFTs between participants. The use case is to allow two or more parties to trade NFTs without the need for a third-party intermediary. The requirements are that the auction be secure, transparent, and efficient.

-

Requirements.

- You will need Reach Programming language installed on your computer. We are going to use Reach for our backend contract.
- One of the advantages of Reach is that it can support multiple blockchains and can be compiled to different programming languages. For simplicity we will use Reach in JavaScript. This means we will need NodeJs installed.

▶

Choosing a Blockchain Platform.

Selecting a Blockchain Platform

Reach is a programming language that supports multiple blockchains, including Algorand, Ethereum, and Conflux. This allows developers to create applications that can interact with multiple blockchain networks. This makes Reach a powerful tool for building cross-chain applications.

What this means is that we can write just one contract and let Reach deal with the complexities of deploying it to multiple blockchains.

You can learn more about that here:

👉 Using Reach with multiple blockchains

▶

Developing the DApp.

Developing the DApp using Problem Analysis and Problem Design.

▶

Problem Analysis

Now that we have a basic understanding of the concepts we need to know, we can start to ask some critical questions.

As a programmer we need to understand the problem that we are trying to solve. Here's a run through of the questions that pop up in my head.

1. What is the purpose of the auction?

> Our purpose is to create a DApp that autonomously runs a safe and secure auction that allows users to buy and sell items.

2. What is the value of the NFT being auctioned?

> The value of the NFT is determined by the owner of the NFT.

3. What is the minimum bid?

> For simplicity, we can use the asset price as the minimum bid.

4. What is the duration of the auction?

> The duration of the auction can be fixed or can be determined by the deployer of the DApp.

5. How will the auction be conducted?

> The auction will be conducted by the DApp itself.

6. What is the NFT being auctioned?

> Again, for simplicity, we will limit the auctioned NFT's to images.

7. How many bidders are there?

> Ideally, we would love to have as many bidders as possible.

10. When is the auction over?

> The auction will be over when the auction duration has elapsed.

11. How will the auction be secured?

> The auction will be secured using Reach programming language.

These questions are pertinent to the development of an internal conversation. However, because we are developing a DApp, we can reframe the problem by limiting the questions to defining the data that we expect the DApp to handle:

▶

Performing Data Analysis.

Turning the information, we know to data.

1.

What information does the DApp need to track?

- The NFT being auctioned.
- The NFT price.
- The NFT amount.
- The auction duration.
- The NFT owner.
- The last bid.
- The latest bid.
- Bidder Address.

•

What information does the DApp need to display?

Each participant in the auction will require the following information:

- The NFT being auctioned.
- The NFT price.
- The NFT amount.
- The auction duration.

However, depending on the role of the participant, the auction may limit how much information each participant has access to.

If the participant is an auctioneer, for example, they can access functions that only they have access to. Such as:

- Adding an NFT to the contract.

- Deciding when the auction will start.

The bidder, on the other hand, does not need to know much. In fact, once a bidder joins the DApp, all they need to see is the highest bid price.

•

How should the app handle user input?

The DApp needs to differentiate private data and public data. Private data should only be accessible to a local computer while public data can be displayed on the blockchain.

▶

Functional Requirements.

In this section, we look at the functions provided by the Reach language that we can use to run the auction.

1.

How can we create a new DApp in Reach.

We'll have to take a look at the Reach syntax to conform to the methods available to us?

•

How will we send the NFT to the contract?

To ensure that the DApp is truly decentralized, we'll need the deployer to forfeit ownership of the NFT until the auction is over.

•

How can we publish the NFT being auctioned to the blockchain.

We'll have to make the NFT information public to all participants.

•

How can we allow a bidder to OPT-IN to the DApp.

There has to be a frontend mechanism that allows the bidder to opt-in to the DApp and place a bid.

•

How will we perform transfers?

Once the auction is done, we'll need to transfer the highest bid to the Auctioneer and the NFT to the winner.

▶

Consensus Mechanisms.

We'll also need to look at what Reach offers when it comes to consensus.

Reaching consensus means that all parties involved in a decision-making process agree on a course of action. This can be difficult to achieve, especially when there are multiple stakeholders with different interests and goals. However, consensus can be reached through careful deliberation and compromise.

1.

How can we ensure that the auction is conducted in a safe and secure manner?

-

How can we secure data that is private?

-

How can we run an open auction on the blockchain?

▶

Problem Design

Let us attempt to respond to the questions raised at Problem Analysis.

The goal of this workshop is to establish a NFT auction and have bidders race to see who can make the biggest bid in the shortest amount of time.

Let's go through some of the questions we need to address before we can start designing the DAPP.

-

▶

In which programming language will we build our DAPP?

Reach is a domain-specific language for developing distributed applications.

1. ▶

Reach Module

The Reach Module must begin with a `version type` on the first line and stored in a `index.rsh` file.

> index.rsh

```
'reach 0.1';
```

> Reach Syntax is written in **JavaScript** syntax.

-
▶

Reach App.

The Reach App specifies the DAPP in it's entirety. It is the body of the DAPP.

Reach uses Module-level Identifiers such as export to identify the module to be compiled.

> index.rsh

```
export const main = Reach.App(() => {
  //DAPP body.
})
```

> The'main' function will contain all the functions we want to perform.

-
▶

Reach Participant.

A Participant is a logical actor that participates in a DAPP and is assigned an address on the consensus network. A Reach participant is capable of storing persistent data on the local state.

> index.rsh

```
export const main = Reach.App(() => {
  //DAPP body.
  const Auctioneer = Participant('Auctioneer', {
      //Auctioneer body
  });
})
```

> All the functions that the 'auctioneer' will need to perform will be housed within the 'Auctioneer body.'

-
▶

Reach API.

A Reach API is a group of Reach Participants competing in a DAPP to achieve the same goal.

> index.rsh

```
export const main = Reach.App(() => {
  //DAPP body.
  const Bidder = API('Bidder', {
      //Bidder interface.
  });
})
```

> The primary distinction between a 'Reach Participant' and a 'Reach API' is that the latter can be called from the actors' frontend.

> The 'Bidder Interface' will contain all the functions that the 'bidder' will need to perform.

-

▶

Thinking Data Analysis.

To decide which types to use to represent our data, we can use reach Types.

We can examine our expected input and output and attempt to convert all of that information to Reach Types.

-

▶

Processing Output Data

Let's look at the `Reach Types` that we'll be using to represent our output data.

Announcing a winner at the end of the auction.

- We will need the participant to learn new information in order to announce a winner:
    1. The winning bid.
    2. The Winner.
- How do we represent these two pieces of data in a DAPP?
    1. The winning bid can be represented by a UInt type.
    2. The winner can be represented by a Address type.

-

▶

Processing Input Data

Let's look at the 'Reach Types' we'll be using to represent our input data.

Adding the NFT for the auction.

- We will need the following data to add a NFT to the contract:
    1. The NFT ID.
    2. The NFT price / starting bid.
    3. The auction duration.
- How can we represent this information in a DAPP ?
    1. To represent the NFT ID, we can use a Token type.
    2. Because the price is a number, we can represent it with a UInt type.
    3. We can represent the auction duration with a UInt type, which will represent block height rather than actual time.

-
▶

Testing Functional Requirements.

To decide which types to use to represent our data, we can use the Reach Functions type.

Reach Functions type will be useful for more efficiently arranging input and output data.

-
▶

Output Functions.

Output functions that will notify our frontend.

-

At the end of the auction, a winner is announced.

- We will need the participant to learn new information in order to announce a winner:
    1. The winning bid.
    2. The Winner.
- We've already established how to represent data; now let's look at how to send this information to the frontend.

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

`showOutcome` is a function that does not expect a return value and sends the `[Address, UInt]` which are the '[winner, winning bid]' to the frontend.

-

Transferring the NFT to the winner.

- We will need to transfer the NFT from the contract to the winner once the auction is completed.
- Reach provides a Transfer function, which is a consensus step that instructs the contract to send a token to the specified address.

```
transfer(`UInt`,`Token`).to(`Address`);
```

When a condition is met, `transfer` takes a `amount` (`UInt`), a `Token`, and transfers the amount to an `Address`.

-

Transferring the highest bid to the auctioneer.

- Once the auction is over, we must transfer the highest bid to the auctioneer.
- Reach provides a Transfer function, which is a consensus step that instructs the contract to send a token to the specified address.

```
transfer(`UInt`).to(`Address`);
```

When a condition is met, `transfer` takes a `amount` (`UInt`) and transfers it to a `Address`.

-

▶

Input Functions.

Input functions will be used to inform our frontend about what the backend expects, as well as to call backend functions from the frontend.

-

Receiving the NFT to be auctioned from the frontend.

Because it is the auctioneers' responsibility to include the NFT in the contract, we will ensure that only the Auctioneer is capable of setting the NFT.

We can use an 'interact' function to obtain information from the frontend whenever a participant backend requires it.

Here is the information we will require from the auctioneer:

1. The NFT ID.
2. The NFT price / starting bid.
3. The auction duration.

- We've already determined how to represent the data; now let's look at how to get this information from the frontend.

```
//getSale function
  getSale: Fun([],[Token, UInt, UInt]),
```

`getSale` function expects the [Token, UInt, UInt]/([nftId, price, auctionTime]) from the frontend.

Reach also includes an Object type for nesting other types.

```
Object({
  nftId: Token,
  minBid: UInt,
  lenInBlocks: UInt,
})
```

Let's add this to the function:

```
getSale: Fun([], Object({
  nftId: Token,
  minBid: UInt,
  lenInBlocks: UInt,
}))
```

-

Allowing a bidder to place a bid.

- Bidders must also place a bid, i.e., call a bid function from the frontend.

```
bid: Fun([UInt], Null),
```

`bid` expects a number from the frontend which a Bidder address will be attached to during the auction.

Alerting when the auction is ready.

- When the auction is ready to begin, we can also notify the Auctioneer.

```
auctionReady: Fun([], Null),
```

`auctionReady` notifies the Auctioneer frontend when the auction is ready.

▶

Looking at Consensus Mechanisms.

Introduction to Reach Steps

In this section, we will introduce new concepts that will help you understand how Reach works.

Reach can be in two states:

- Local step
- Consensus step

The majority of DAPPs include a creator, an actor, a wager, and a condition. Before a contract becomes autonomous, the creator publishes the wager and condition criteria. Once the creator has done this, they have no control over the outcome and cannot pause the contract once it has begun. The bidder can view the contract on the blockchain and decide whether to participate.

Local steps are performed locally by a single actor, whereas consensus steps are performed on the blockchain in consensus. Local steps exist to ensure that each actor is unaware of what any other actor is up to in order to improve anonymity and security.

If they choose to make the information public, they must go through a consensus step and publish it on the blockchain. Consensus steps also ensure that the contract's core logic and conditions are run on the blockchain, where all active actors can see what is happening.

Let's go over the tasks that we'll need to complete in order to have a successful auction:

▶

Adding Actors

We've already decided how we'll represent our data, and we've established functions that can be used to get the necessary data; the last step is to incorporate the functions into classes that can perform logic and store

states. They are referred to as Participants in Reach.

-

Adding an Auctioneer Participant

- We saw how to collect data using input and output functions in the functions section; now let's add the necessary data to our auctioneer participant.

```
const Auctioneer = Participant('Auctioneer', {
  //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),
});
```

- Here, we create an Auctioneer participant with the name 'Auctioneer' and the auction data.
- We used the `getSale` function to get the NFT data from the frontend.
- We used the `auctionReady` function to notify the Auctioneer when the auction is ready.
- We used the `seeBid` function to notify the Auctioneer when a bidder has placed a bid.
- We used the `showOutcome` function to notify the Auctioneer when the auction is over and who the winner is.

-

Adding a Bidder Participant.

- A participant class will also be used for the bidder. However, unlike the Auctioneer, who is a single actor, we anticipate that multiple bidders will be added to the contract.
- Reach provides a way of representing multiple participants with the Reach API class.
- Consider the API to be a representation of multiple participants racing toward a common goal.
- In our case, we anticipate that Bidders will be able to place a bid.

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

- A Bidder interface is available for representing multiple bidders.
- Each bidder will have a bid function through which they can place a bid.

> One benefit of the Reach API is that functions can be called from the frontend.

-
▶

Initializing the contract.

What happens after the actors/participants are created.

So far we've only discussed the API and the Participant. However, there are other Reach interfaces that we have not covered include Views and Events.

These interfaces represent which functions and classes the frontend should replicate and should be placed before the 'init()' statement.

```
const newParticipant = Participant(participantName, participantInteractInterface)

const newAPI = API(APIName, APIInteractInterface)

const newView = View(ViewName, ViewInteractInterface)

const newEvent = Events(EventName, EventInteractInterface)

init()

// Consensus step or local step.
```

`init()` symbolizes the beginning of the DApp to be compiled. In other words, anything that follows the `init` statement is either a local step or a consensus step.

-
▶

Using Local Steps.

What follows the 'init()' statement.

-

Local Private Step.

When the 'init()' statement is executed, the DApp enters a local private step. This means that any information accessed is only available on the participant's local machine.

Local Public Step.

Local private is not very useful if we have information that we need other actors to access, such as NFT data. So, how do we make the transition from local private to local public?

- To accomplish this, we use Reach declassify.

> Reach declassify allows you to send data from the frontend to the backend. To get the NFT information from the frontend, let's test this with the 'Auctioneer' participants' 'getSale' function.

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

- `Auctioneer.[only]()` function makes sure that only the `Auctioneer` i.e., the creator of the contract, can access this function.
- `interact` is a function used to get information from the frontend.
- `declassify` makes the information public.

> Here, we are interacting with the `Auctioneer` frontend and `await`ing the result (const {nftId, minBid, lenInBlocks}).

The information is public but it's still local. Let's move to a consensus step to publicize the information on the contract.

Using Consensus Steps.

Publishing information onto the contract.

Reach provides a few mechanisms that can assist us in moving from a local step to a consensus step.

1.

Publish

We can use Reach Publish to share NFT information with the contract during the consensus step.

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

- `Auctioneer.publish()` function makes sure that only the `Auctioneer` i.e., the creator of the contract, can publish this information onto the contract.

> Using Commit()

Once in a consensus step, we can use [Commit()] to return to a local step (https://docs.reach.sh/rsh/consensus/#rsh_commit).

How does this help the integrity of the DApp?

- Security reasons

We use commit to ensure that we are back in a 'local private' state before performing sensitive functions like contract payments.

```
commit();
```

-

Pay

We can now transfer the NFT from the Auctioneer to the contract because we are back in a 'local private' step.

```
Auctioneer.pay([[1, nftId]])
```

- The `Auctioneer.pay()` function ensures that only the `Auctioneer`, i.e., the contract's creator, can pay.
- We are submitting one NFT Token for auction to the contract.

> Because an NFT should be unique, we send [1] NFT. Rather than sending the 'UInt 1' directly, we can store the information in a variable.

```
const amt = 1;
```

Then, Pay becomes :

```
Auctioneer.pay([[amt, nftId]])
```

The DApp now has the information it needs to conduct an auction. The auction logic is all that remains. But first, let us inform the Auctioneer that the auction is ready.

```
Auctioneer.interact.auctionReady();
```

-
▶

What consensus transfer can we use for the auction ?

Now let's take a look at the consensus transfer that we can use for the auction.

When it comes to consensus transfer, or when multiple actors come together to agree on a single state, we can determine which consensus approach to use by asking ourselves a few questions:

1. How many participants can act at a particular time?
2. How many things can be done?
3. How many times can it be done?

We could use Pay to transfer tokens to the contract if there was only one participant. However, because multiple bidders are expected to compete, we can use a Reach Race. A reach race allows multiple actors to compete for the publication of a consensus step.

However, there is a problem with this solution; the race function only runs once, and we need to allow bidders to place as many bids as they want as long as two conditions are met:

- The bid is placed before timeout.
- The bid placed is larger than the last bid placed.

We need to put the race in a while loop that allows us to do this. A while loop that runs until timeout is reached.

Alternatively, Reach Parallel Reduce can be used. In a parallel, actors are racing against the clock to publish data onto the contract. Parallel reduce uses a while loop that resolves the auction to a single outcome or winner.

Parallel reduce is a recursive algorithm that generates a single winner from a tree of bidders.

```
    const [winner] = parallelReduce([Auctioneer])
```

> The Auctioneer is the default winner before any bids are placed.

However, this is not a complete solution; for a closer look at the format, see Reach Parallel Reduce. For now we're going to look at how we can use parallel reduce for the auction.

-

The Invariant

```
.invatiant(balance() == 0)
```

After each iteration, the invariant is checked to ensure that the parallel reduce is still valid. We're checking to see if the balance is zero.

-

The While loop

```
.while(lastConsensusTime() < timeOut)
```

The while loop is active as long as the lastConsensusTime is less than the time out value.

> The time of the last consensus step is represented by the lastConsensusTime (The last time a pay,publish or transfer was used).