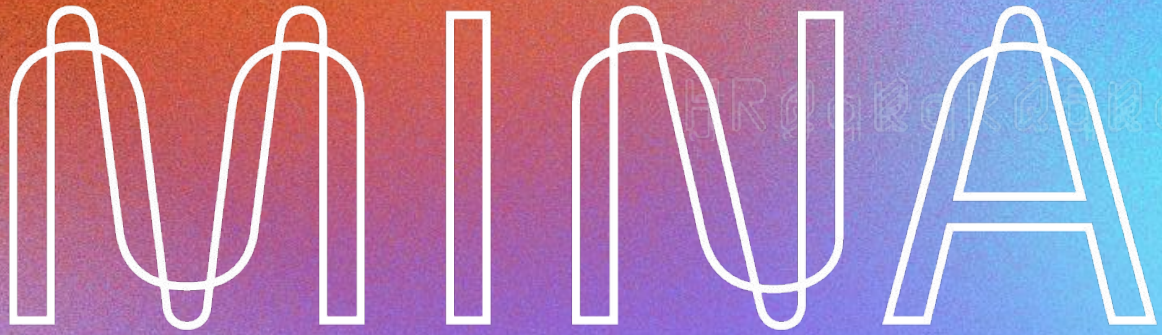


MINA Intro

MINA

(I) Mina Protocol

(II) ZK-SNARK

(III) O1js

(IV) Simple ZKApp

(V) Protokit

Overview

- What is difference?

MINA is Efficient

	Ethereum Smart Contracts	Mina zkApps
Language	Smart contracts are written in Solidity.	zkApp smart contracts are written using o1js (a TypeScript library).
Execution Environment	Smart contracts run on every Ethereum node.	zkApps run client side in a user's web browser, and publish only a small validity proof which is verified by the Mina nodes.
Transaction Cost	Execution costs are variable, and determined using a gas model.	Execution costs are small, and constant because the Mina nodes are verifying the same size proof regardless of the amount of client-side computation.
Application Storage	Ethereum is designed around the idea that storage, and computation are inherently coupled; all state must live on every Ethereum node.	Mina's design allows state, and computation to be decoupled so that application state can live anywhere; developers can choose a solution that fits their cost/security requirements best.
Developer Tooling	New developer tools with unusual patterns like Hardhat, and Truffle are needed in order to manage the deployment of Ethereum smart contracts.	The zkApp CLI manages scaffolding, linting, testing, and deployment using common JavaScript/TypeScript tools you are already familiar with.
Scaling	Ethereum nodes must execute every transaction directly making horizontal scaling hard.	Mina's recursive zero knowledge proofs allow snark-workers to compress the blockchain, and developers to compress transactions using native rollups for exponential scaling.
Consensus	Ethereum nodes must download the entire block history (~700GB) in order to verify the current finalized chain state.	Mina clients can verify the current finalized state using a single 22KB recursive zero knowledge proof.

Provable Program

Example:

Mini Pow(proof of work) game

Full code

```
export class game extends SmartContract{
  @state(Field) counter = State<Field>(Field(0));
  @state(Field) answer = State<Field>();



    @method async setAnswer(answer: Field){

      this.counter.requireEquals(Field(0));

      const answerHash = Poseidon.hash([Field(100), answer]);

      this.answer.set(answerHash);
      //console.log(answerHash);
      this.counter.set(Field(1));
    }

    @method async guess(guess: Field){

      const guessHash = Poseidon.hash([Field(100), guess]);

      const answer = this.answer.get();
      this.answer.requireEquals(answer);
      //when you access state variables,
      //you need to explicitly declare that
      //you're using the current on-chain state.


      guessHash.assertEquals(answer);
      console.log("정답입니다!");

    }


}
```

MINI POW: Explanation

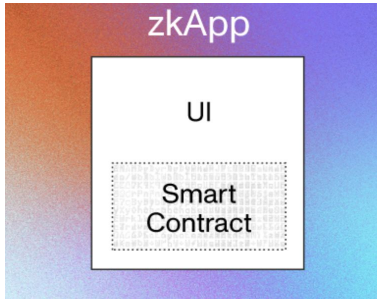
Mini Pow(proof of work) game

There are two role: owner and challenger.

1. Owner set answer and hashed it.
2. AnswerHash is stored at MINA On-chain state
3. Challenger can guess it.

It can be implemented with solidity?

- Well, I think “NO”.
- Thanks to ZKApp architecture, proof of local operation will be submitted.
- As you can see at following page, if you choose incorrect answer, it will be ignored.
- https://nam2ee.github.io/04-zkapp-browser-ui/



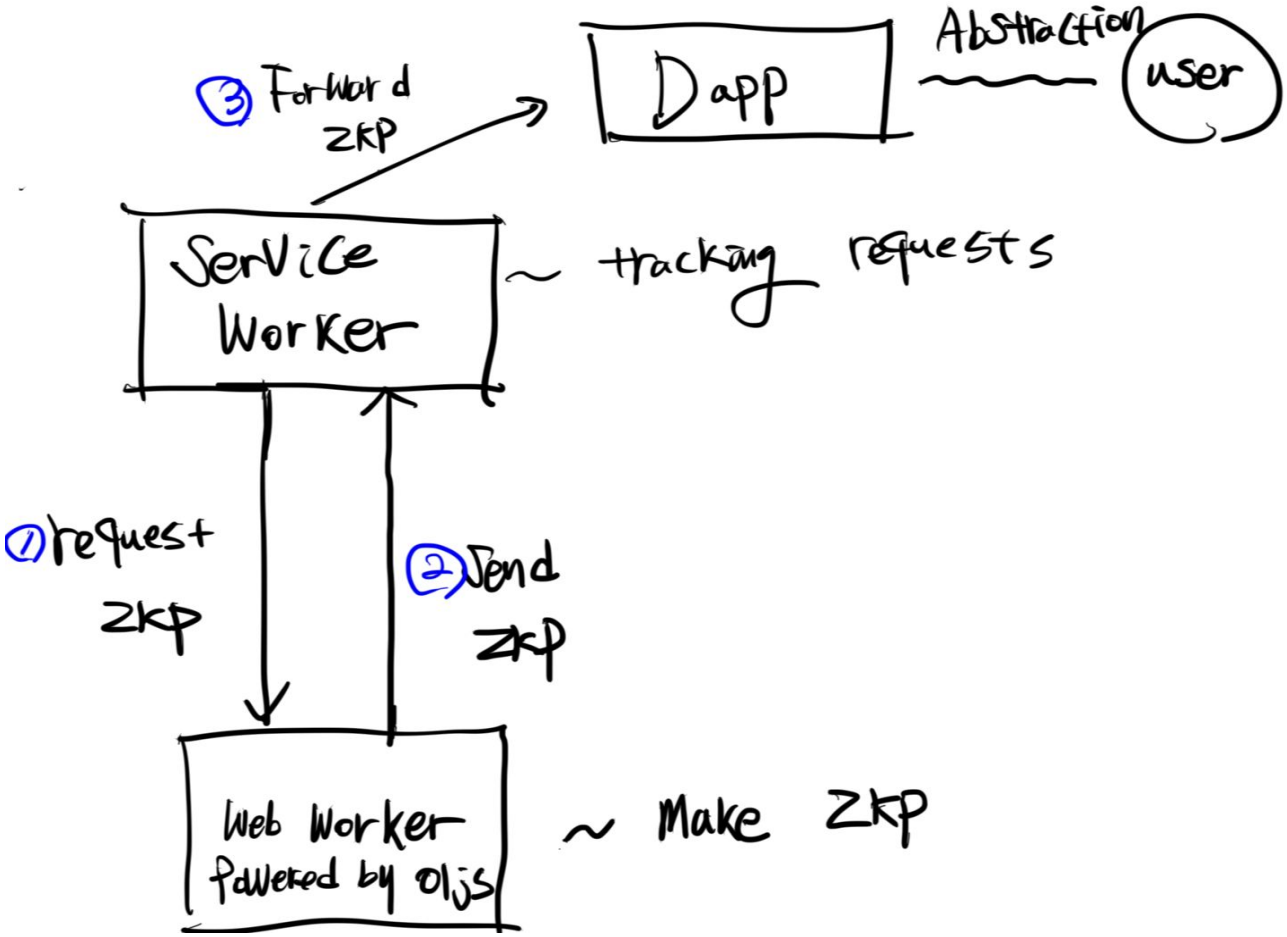
It will be gas-bomb at another blockchain

- You must pay gas for each guessing chance.
- It seems like simple game, but implies potential for MINA.

Provable Program

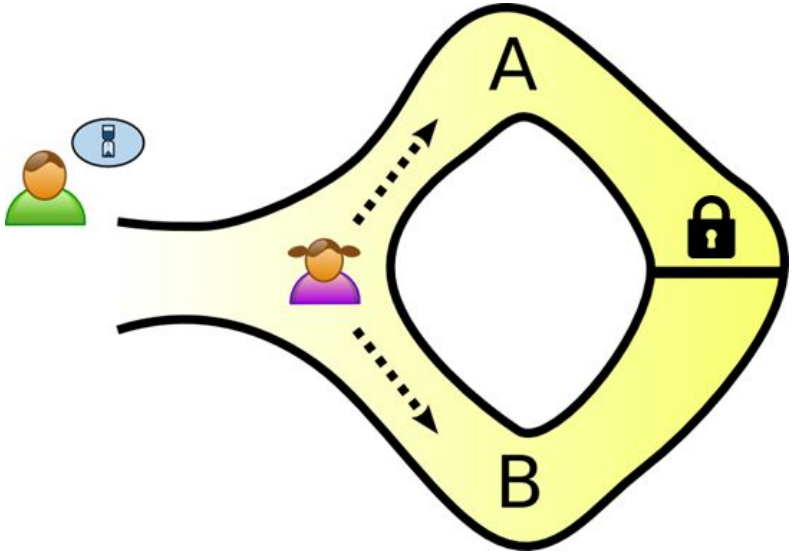
Example:

ZKextension


Dapp
Abstraction
user
3 Forward ZKP
Service Worker
~ tracking requests
1 request ZKP
2 send ZKP
Web Worker powered by o1js
~ Make ZKP

ZK-SNARK

- "Don't think about Ali Baba's cave."



ZK-SNARK: Assume trustless environment

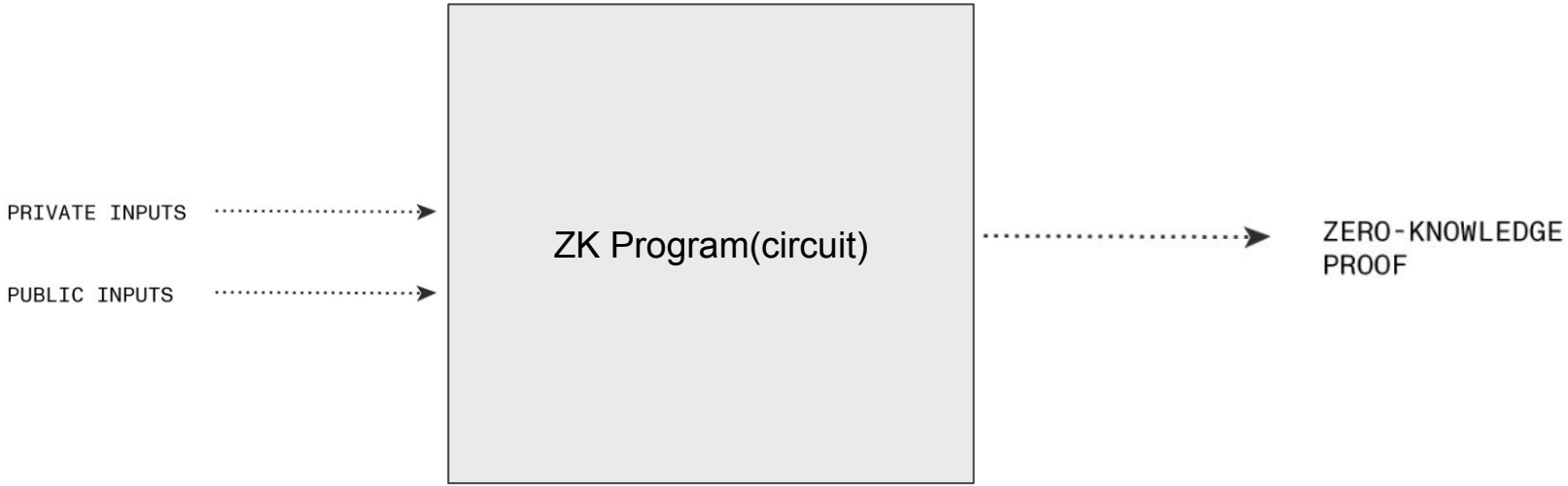
-Before we start talking about high-level zk-snark,we must assume that we are in trustless environment.

-Like previous game, we must prove that we know answer.

$$x = g^a \bmod q.$$

Then, How to prove it?

If you catch the answer, ZK program makes zk-proof!



ZK-SNARK: Assume trustless environment

- The way verifying our proof is processed by some protocol.If you wanna know deeper, read “main-moonmath”

- We need to know just that blackbox protocol which processes my proof.

- So we must focus on ‘circuit’(zk program)

ZK-SNARK: Assume trustless environment

- The way verifying our proof is processed by some protocol.If you wanna know deeper, read “main-moonmath”

- We need to know just that blackbox protocol which processes my proof.

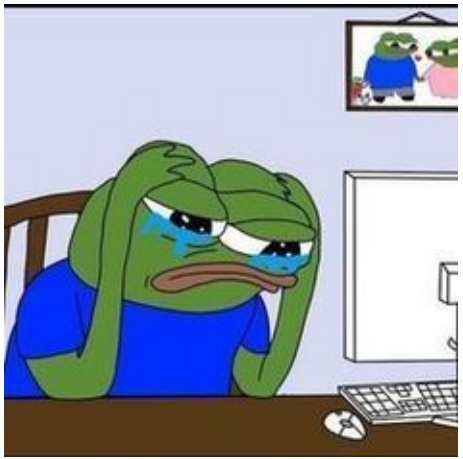
- So we must focus on ‘circuit’(zk program)

So, How to build circuit for zk program?

- Without O1js, we will be cooked for a long time.

```
pragma circom 2.0.0;

template NAND() {
    signal input s1;
    signal input s2;
    signal output s3;

    s3 <== 1 - s1*s2;
}

component main = NAND();
```

```
pragma circom 2.0.0;
include "circomlib/circuits/gates.circom"

template GateSample() {
    signal input s1;
    signal input s2;
    signal input s4;
    signal s3;
    signal output s5;

    component G1 = NAND();
    component G2 = AND();

    G1.a <== s1;
    G1.b <== s2;
    s3 <== G1.out;
    G2.a <== s3;
    G2.b <== s4;
    s5 <== G2.out;
}

component main = GateSample();
```



Using O1js for building ZK Program at mina

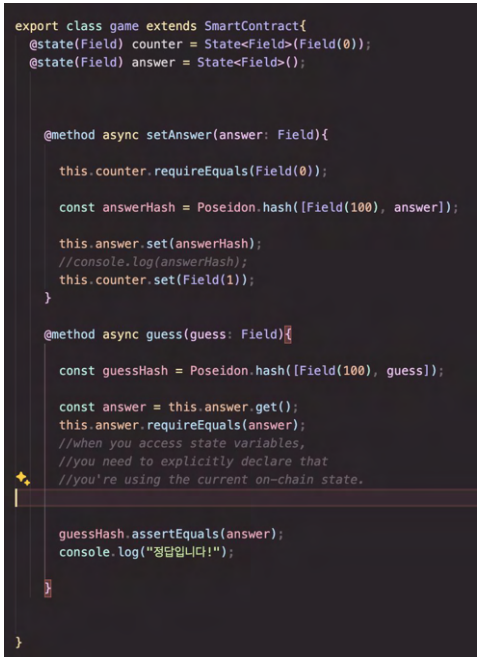
- O1js has many cryptography primitives. Like Fields, Poseidon(ZK-compatible hash), ECDSA …

- And most of them are abstracted so we can use them easily.

O1js Architecture & Work flow

- Smart Contract

- This class is template for building zk circuit



```
export class game extends SmartContract{
  @state(Field) counter = State<Field>(Field(0));
  @state(Field) answer = State<Field>();


    @method async setAnswer(answer: Field){

      this.counter.requireEquals(Field(0));

      const answerHash = Poseidon.hash([Field(100), answer]);

      this.answer.set(answerHash);
      //console.log(answerHash);
      this.counter.set(Field(1));
    }

    @method async guess(guess: Field){

      const guessHash = Poseidon.hash([Field(100), guess]);

      const answer = this.answer.get();
      this.answer.requireEquals(answer);
      //when you access state variables,
      //you need to explicitly declare that
      //you're using the current on-chain state.


      guessHash.assertEquals(answer);
      console.log("정답입니다!");

    }

}
```

O1js Architecture & Work flow

- ZKProgram

- This class is also template for building zk circuit

- My examples have same logic for each other.

```
const zkapp = ZkProgram({
  name: 'game',
  publicInput: Field,

  methods:{
    setAnswer: {
      privateInputs: [Field],
      async method( hash: Field, answer: Field){ //hash: Poseidon.ha
        hash.assertEquals(Poseidon.hash([Field(100), answer]));
      }
    },
    guess: {
      privateInputs: [SelfProof<Field, void>],
      async method(guess: Field, problem: SelfProof<Field, void>){
        problem.verify();
        let challenge = Poseidon.hash([Field(100), guess]);
        challenge.assertEquals(problem.publicInput);
      }
    }
  }
}
)
```

SmartContract

- Basically, @method are returned, that implies all assert statements are true.

-

```
export class game extends SmartContract{
  @state(Field) counter = State<Field>(Field(0));
  @state(Field) answer = State<Field>();


  @method async setAnswer(answer: Field){

    this.counter.requireEquals(Field(0));

    const answerHash = Poseidon.hash([Field(100), answer]);

    this.answer.set(answerHash);
    //console.log(answerHash);
    this.counter.set(Field(1));
  }

  @method async guess(guess: Field){

    const guessHash = Poseidon.hash([Field(100), guess]);

    const answer = this.answer.get();
    this.answer.requireEquals(answer);
    //when you access state variables,
    //you need to explicitly declare that
    //you're using the current on-chain state.


    guessHash.assertEquals(answer);
    console.log("정답입니다!");

  }

}
```

ZKProgram

- Basically, method return explicit ‘proof’
- That proof is verified by verify() method.

- Main difference between smartcontract is dependency for onchain

```
const zkapp = ZkProgram({
  name: 'game',
  publicInput: Field,

  methods:{
    setAnswer: {
      privateInputs: [Field],
      async method( hash: Field, answer: Field){ //hash: Poseidon.ha
        hash.assertEquals(Poseidon.hash([Field(100), answer]));
      }
    },
    guess: {
      privateInputs: [SelfProof<Field, void>],
      async method(guess: Field, problem: SelfProof<Field, void>){
        problem.verify();
        let challenge = Poseidon.hash([Field(100), guess]);
        challenge.assertEquals(problem.publicInput);
      }
    }
  }
}
)
```

Important point for zkapp building

- In general, zk program is compiled by deterministic circuit for proving.

- So, it means side effects must not be allowed. Like runtime inference methods. Feels like functional programming. For example, you may use reduce method, and forEach for handling ‘Action’ in building your own circuit.
- On-chain states up to 256 bytes

Important point for zkapp building

- Because of circuit, you must sure that your data structures are on ‘Field’

- What is ‘Field?’ …

Tutorial

- https://docs.minaprotocol.com/zkapps/zkapp-development-frameworks

- https://nam2ee.github.io/04-zkapp-browser-ui/

Doing some tutorial -O1js

Let’s implementation some simple number guessing problem

First, let’s implement SmartContract

Must include

setAnswer method

: check proper onchain state

: must can be called once.

: privacy-preserving ; must hide real answer

Guessing method

: check the input equals real answer

Deploy frontend

I already deployed my site.

https://nam2ee.github.io/04-zkapp-browser-ui/

Protokit

- Making your own rollup.(App-chain)

- You can make Privacy enable, Faster ZKApp with Protokit.

Protokit Overview

Runtime

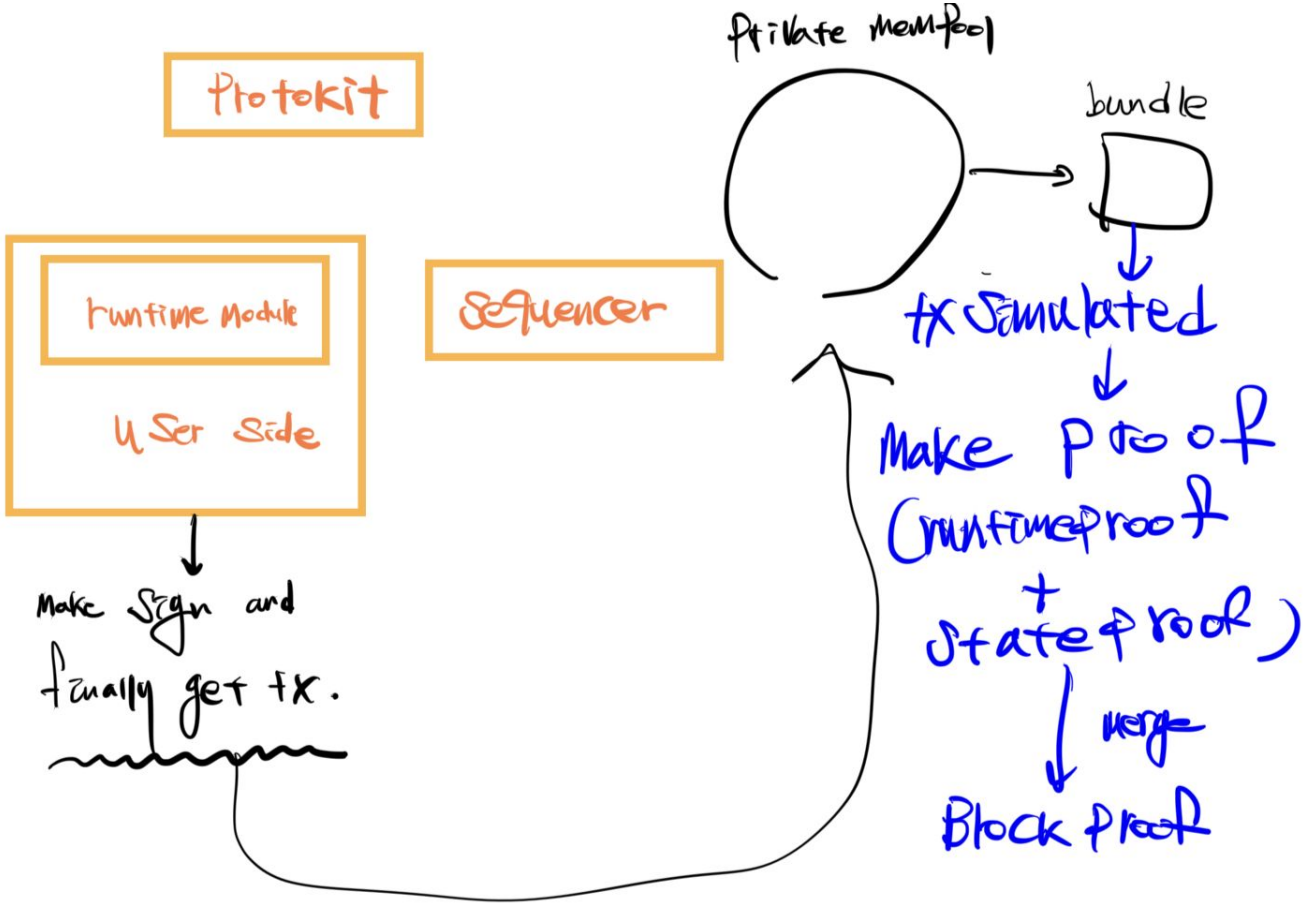
-Interaction with client(business logic)

Sequencer

Sequencing

Mina

-Using settlement L1


Protokit
Runtime Module
User Side
Sequencer
Make Sign and
finally get tx.
Private mempool
bundle
tx Simulated
Make Proof
(runtimeproof
+
stateproof)
merge
Block proof

Difference with MINA ZKapp

- ZKApp - user make ZK-proof

- Protokit - sequencer make ZK-proof with highly efficient proving pipeline. So fast.

- How about Privacy? - I’ll explain next slides

How they can enable privacy?

- As you can see, they using interaction between runtime module and ZKProgram.

- Runtime module can handling Proof as argument. So user's transaction may include ZKP.

```
export class AirdropProof extends Experimental.ZkProgram.Proof(airdrop) {}

type AirdropConfig = Record<string, never>;

@runtimeModule()
export class Airdrop extends RuntimeModule<AirdropConfig> {
  @state() public commitment = State.from<Field>(Field);
  @state() public nullifiers = StateMap.from<Field, Bool>(Field, Bool);

  public constructor(@inject("Balances") public balances: Balances) {
    super();
  }

  @runtimeMethod()
  public setCommitment(commitment: Field) {
    this.commitment.set(commitment);
  }

  @runtimeMethod()
  public claim(airdropProof: AirdropProof) {
    airdropProof.verify();
    const commitment = this.commitment.get();

    assert(
      airdropProof.publicOutput.root.equals(commitment.value),
      "Airdrop proof does not contain the correct commitment"
    );

    const isNullifierUsed = this.nullifiers.get(
      airdropProof.publicOutput.nullifier
    );

    assert(isNullifierUsed.value.not(), "Nullifier has already been used");

    this.nullifiers.set(airdropProof.publicOutput.nullifier, Bool(true));

    this.balances.mint(this.transaction.sender, UInt64.from(1000));
  }
}
```

State

- At Protokit, they use single merkle tree for all runtime modules.

- For sequencer, must need to state change proof and transition proofs. Get them, must test if state is result of all transitions.

Protokit interaction

- Protokit provides convenience developer ui for interacting with client

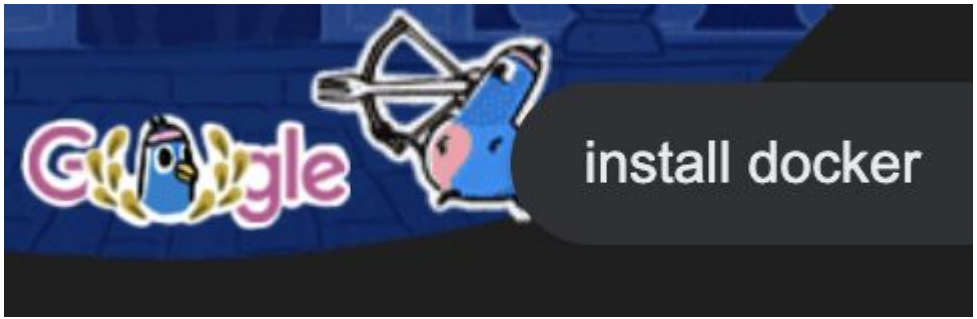
```
import { InMemorySigner } from "@proto-kit/sdk";
import { UInt64 } from "@proto-kit/library";
import { client as appChain } from "./../src/client.config";
import { PrivateKey } from "o1js";
import { GuestBook } from "../src/guest-book";

const signer = PrivateKey.random();
const sender = signer.toPublicKey();

describe("interaction", () => {
  let guestBook: GuestBook;

  beforeAll(async () => {
    await appChain.start();

    const inMemorySigner = new InMemorySigner();

    appChain.registerValue({
      Signer: inMemorySigner,
    });

    const resolvedInMemorySigner = appChain.resolve("Signer") as InMemorySigner;
    resolvedInMemorySigner.config = { signer };

    guestBook = appChain.runtime.resolve("GuestBook");
  });

  it("should interact with the app-chain", async () => {
    const rating = UInt64.from(3);
    const tx = await appChain.transaction(sender, () => {
      guestBook.checkIn(rating);
    });

    await tx.sign();
    await tx.send();
  });
});
```

Using Protokit

https://github.com/nam2ee/MINA_Start

Scan this!!! ---->

Before start, setup Docker



Search at google, and do yourself

Check with cli

- docker
- docker --version

First, Setup -> node, pnpm , docker

Prerequisites:

- Node.js `v18` (we recommend using NVM)
- pnpm `v9.8`
- nvm

For running with persistance / deploying on a server

- docker `>= 24.0`
- docker-compose `>= 2.22.0`

```
git clone https://github.com/proto-kit/starter-kit my-chain
cd my-chain

# ensures you have the right node.js version
nvm use
pnpm install
```

Instructions are on - – - - - - - – >>>>

Setup Docker

```
# start databases
pnpm env:development docker:up -d
# generate prisma client
pnpm env:development prisma:generate
# migrate database schema
pnpm env:development prisma:migrate
```

Build chain / Run Web at Local

```
# build & start sequencer, make sure to prisma:generate & migrate before
pnpm build --filter=chain
pnpm env:development start --filter=chain

# Watch sequencer for local filesystem changes
# Be aware: Flags like --prune won't work with 'dev'
pnpm env:development dev --filter=chain

# Start the UI
pnpm env:development dev --filter web
```

See if it goes well with your eyes
Enter localhost:3000

Start Customizing!

For customizing, you must implement / modify

1. Own runtime module
2. Runtime config

Own runtime module

Thanks to offchain-storage, we can define on-chain state without limitation.

Let’s go

Modifying config

You must modify runtime config for notice of new runtime modules to other components.

```
export const modules = {
  Balances,
  Chating

};

export const config: ModulesConfig<typeof modules> = {
  Balances: {
    totalSupply: Balance.from(10_000),
  },
  Chating: {},
};
```

Implementation of component works like webworker apps/web/lib/stores/chat.tsx

Let’s go

Fetch next.js components

Codes are on - – - - - - - – >>>>

What will you build?