Building A Custom Ethereum Testnet (Union Square)

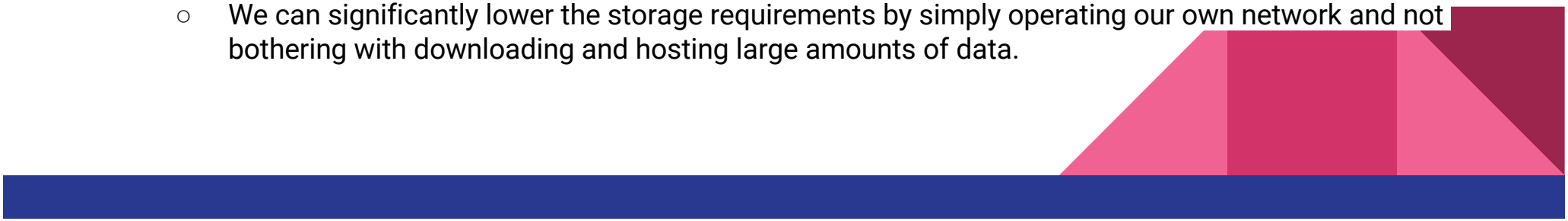
James Menacho, Julian Oquendo,
Kevin Machine, Fred Fu, and Steven Cherry II

Some Fun Facts About Ethereum Test-Nets

- Ethereum has 5 different official testnets, all of which are named after subway stations all across the world: **Ropsten, Kovan, Rinkeby, Sokol & Görli** (Which was the inspiration for naming our's "Union Square")
- There are 2 different types of consensus mechanisms for Ethereum testnets: **Proof-Of-Authority & Proof-Of-Work**
    - **POA:** A certain number of actors are pre-determined to be able to confirm transactions and mine blocks but anyone in the world can connect and make transactions on these networks.
    - **POW:** All transaction are verified by miners through a complex system of solving difficult mathematical puzzles but, because the network and the tokens exchanged on it hold no real value, a lot of miners do not necessarily care enough to to uphold the integrity of the testnet long-term which makes it prone to coordinated attacks.
- While Kovan, Rinkeby, Sokol & Gorli all use POA, Ropsten actually uses POW (Proof-Of-Work) which makes it the testnet that most closely resembles Ethereum's Mainnet.
- For simplicity's sake, we chose to use Proof-Of-Authority for Union Square.

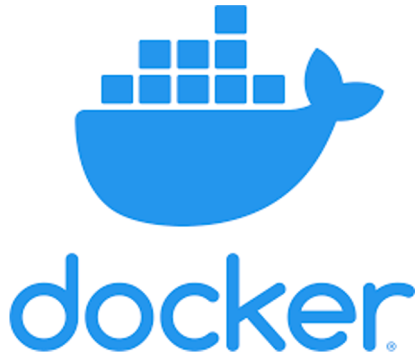
The Benefits Of Hosting Our Own Testnet

- **Persistence** - There is no guarantee that any of the official public testnets will not be reset by the organizers.
    - With no unauthorized participants on the network, we will never run into a situation where transactions do not get approved or the network is clogged with spam transactions.
- **More Control** - By hosting our own testnet, we have total control over where the coins are placed and when new blocks are being mined.
    - Rather than relying on a faucet to retrieve fake Ether, we were actually able to pre-fund a wallet on the network which makes our possible transactions less restrictive in terms of amount or size of transactions.
- **Storage Requirements** - Participating in any of these networks requires you to sync up with them.
    - We can significantly lower the storage requirements by simply operating our own network and not bothering with downloading and hosting large amounts of data.



Tools Needed to Set Up The Testnet

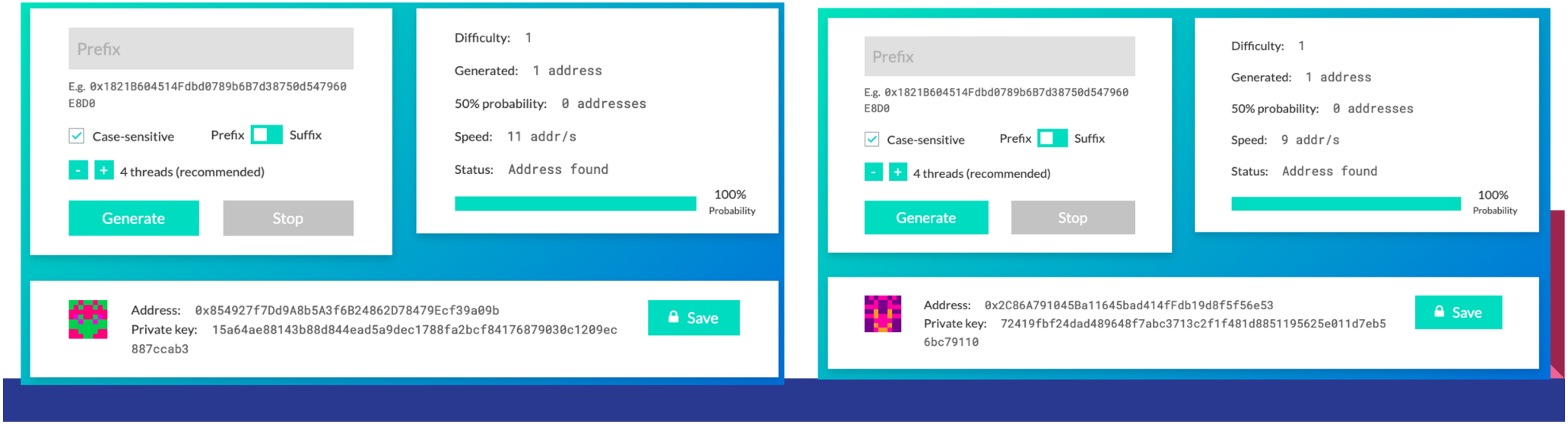
- **A single GETH node**
  - We ran the official implementation downloaded directly from Go Ethereum
- **Blockscout**
  - We used Blockscout to track and verify our testnet locally on our computers
- **PostgreSQL**
  - Incorporated behind the scenes by Blockscout
- **Docker**
  - We used Docker to tie together all the pieces and “containerize” our custom network



Step 1: Generating Two Ethereum Addresses

- We used **Vanity ETH** to establish two seperate Ethereum addresses for our custom testnett (https://vanity-eth.tk/)
  - **Buffer:** We preloaded all of our fake Ether onto this address
  - **Signer:** We used this address to interact with our Geth node as the main miner for the network

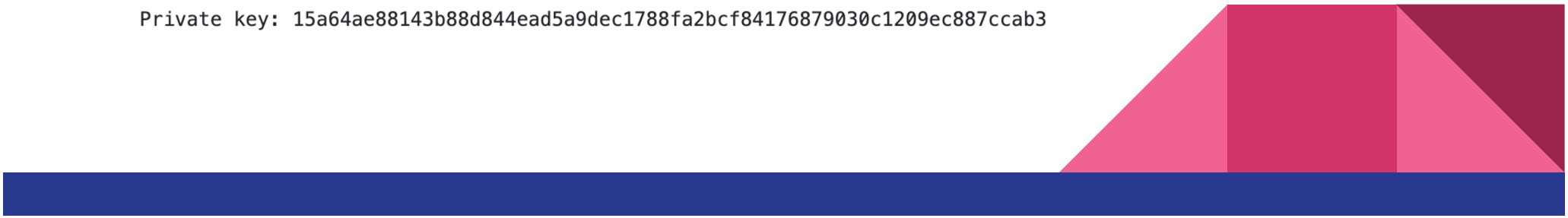
Signer Account:



Step 1: (Continued)

- We generated our two addresses and corresponding key pairs, saving them somewhere handy because we would need to use these at multiple points later on in the process

```
Testnet

Buffer (This is the wallet we use to log into our network)

Address: 0x2C86A791045Ba11645bad414fFdb19d8f5f56e53
Private key: 72419fbf24dad489648f7abc3713c2f1f481d8851195625e011d7eb56bc79110

Signer

Address: 0x854927f7Dd9A8b5A3f6B24862D78479Ecf39a09b
Private key: 15a64ae88143b88d844ead5a9dec1788fa2bcf84176879030c1209ec887ccab3
```



Step 2: Creating & Defining Our Network Using A Custom JSON File

- We prepared our genesis.json file.
  - This file describes the initial state of our testnet, defining every part of it at its genesis!

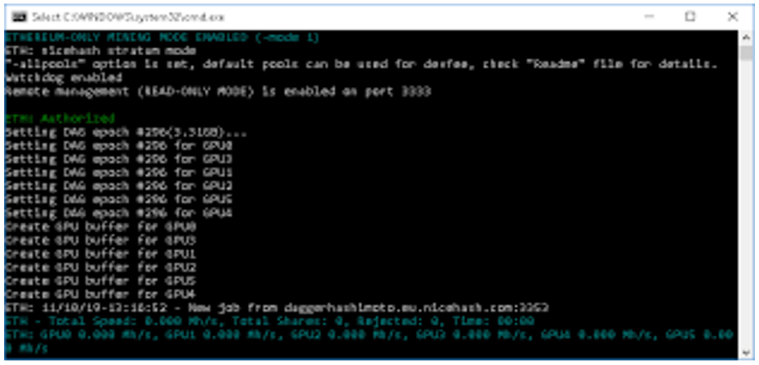
genesis.json File:


```
{
    "config": {
        "chainId": 5555,
        "clique": {
            "period": 60,
            "epoch": 30000
        },
        "homesteadBlock": 0,
        "eip150Block": 0,
        "eip150Hash": "0x0000000000000000000000000000000000000000000000000000000000000000",
        "eip155Block": 0,
        "eip158Block": 0,
        "byzantiumBlock": 0,
        "constantinopleBlock": 0,
        "petersburgBlock": 0,
        "istanbulBlock": 0,
        "muirGlacierBlock": 0
    },
    "extraData": "0x0000000000000000000000000000000000000000000000000000000000000000854927f7Dd9A8b5A3f6B24862D78479Ecf39a09b0000000000000000000000000000000000000000
    "alloc": {
        "2C86A791045Ba11645bad414fFdb19d8f5f56e53": {
            "balance": "0x2000000000000000000000000000000000000000000000000000000000000000"
        }
    },
    "nonce": "0x0",
    "timestamp": "0x5eaa7b09",
    "gasLimit": "0x47b760",
    "difficulty": "0x1",
    "mixHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "coinbase": "0x0000000000000000000000000000000000000000",
    "number": "0x0",
    "gasUsed": "0x0",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000"
}
```

Step 2: (Continued)

Below is a brief explanation for some of the key sections which we defined in our genesis.json file:

- **Config Section**- Config section describes core blockchain settings.
  - **ChainID** - Identifies the blockchain: **We chose 2323 as our ChainID**
  - **Clique -** describes configuration for Clique consensus protocol which contains two important fields:
    - **Period -** How often blocks will be mined (in seconds): **We chose 60 seconds, or every minute**
    - **Epoch-** The length of an epoch: **Epoch is a period in which signers can vote on either adding or kicking out validators from the group.**
  - **Alloc** - This section describes initial balances of accounts on the network: **This is where we loaded in our Buffer Account address to pre-load all of our network's fake Ether**



Step 3: Bootstrapping The Network

In order to bootstrap the network, we had to create a docker-compose.yml file.

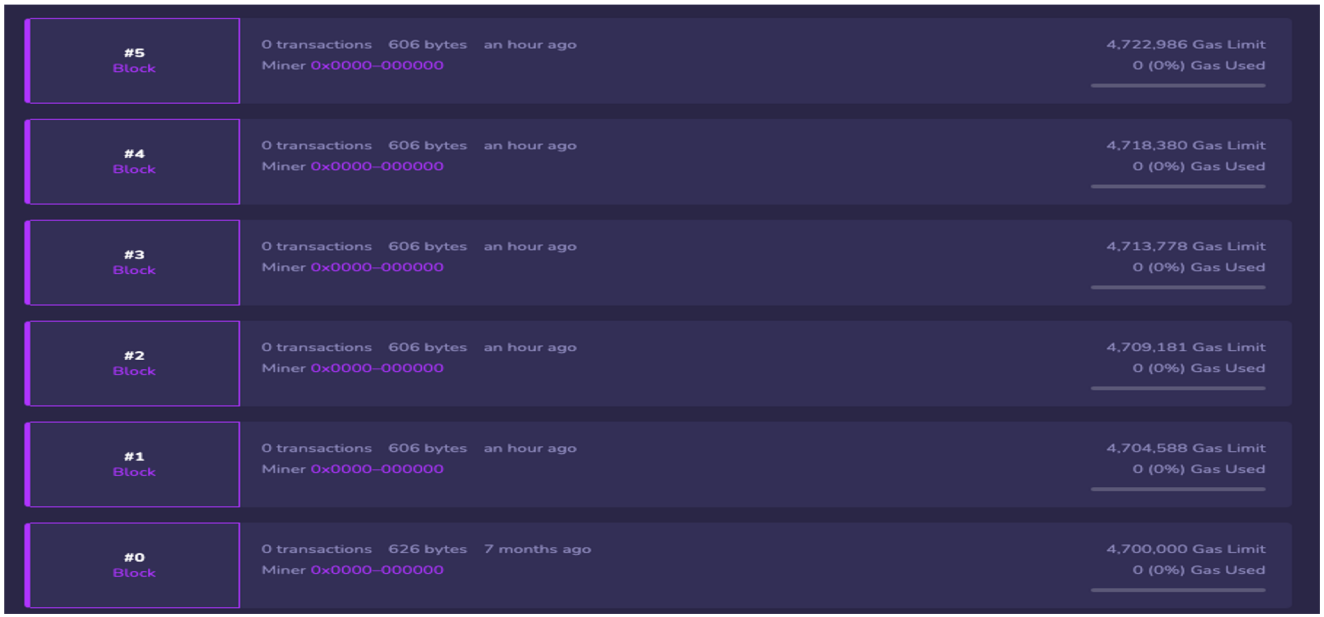
docker-compose.yml:

```yaml
version: '3'

services:
  geth:
    image: ulamlabs/geth-poa-testnet:latest
    environment:
      - ETH_PASSWORD=abcdefghijkl
      - ETH_PRIVATE_KEY=15a64ae88143b88d844ead5a9dec1788fa2bcf84176879030c1209ec887ccab3
      - ETH_ADDRESS=854927f7Dd9A8b5A3f6B24862D78479Ecf39a09b
    ports:
      - 8178:8178
      - 8546:8546
    volumes:
      - ./genesis.json:/app/genesis.json

  blockscout:
    image: ulamlabs/blockscout:latest
    links:
      - geth
      - postgres
    ports:
      - 4000:4000
    environment:
      - DATABASE_URL=postgresql://postgres:@postgres:5432/postgres?ssl=false
      - ETHEREUM_JSONRPC_VARIANT=geth
      - ETHEREUM_JSONRPC_HTTP_URL=http://geth:8178
      - ETHEREUM_JSONRPC_WS_URL=ws://geth:8546
      - MIX_ENV=prod
      - BLOCKSCOUT_HOST=localhost
      - COIN=eth
      - NETWORK=POA
      - SUBNETWORK=Local Testnet

  postgres:
    image: postgres:12
    command: postgres -c 'max_connections=500'
    environment:
      - POSTGRES_HOST_AUTH_METHOD=trust
```

Step 4: Confirming The Testnet Is Up & Running

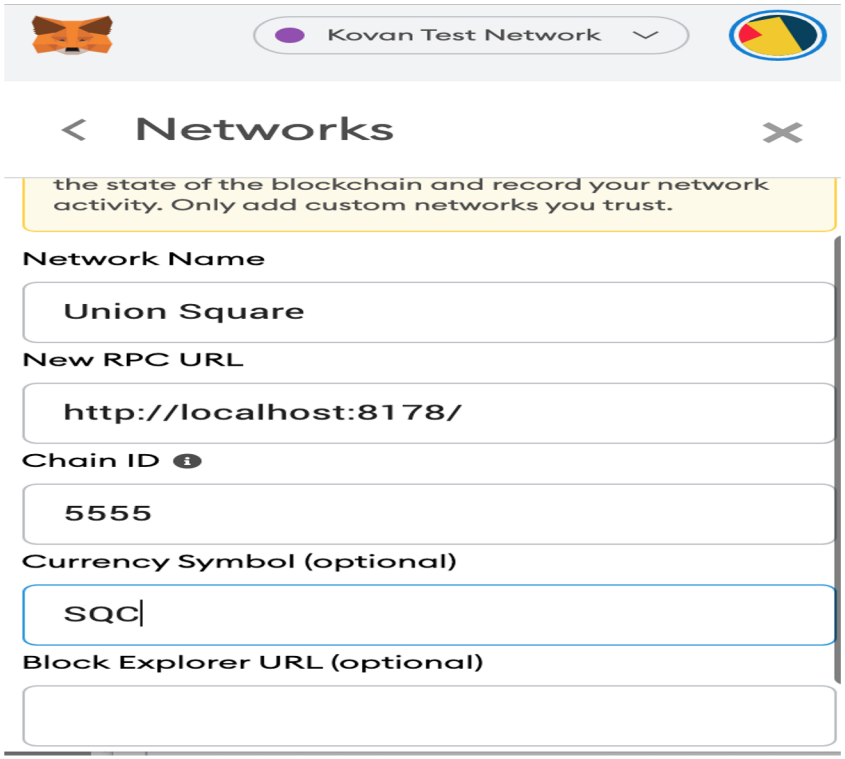
After we have compiled both the genesis.json and docker-compose.yml file in the same folder, we used Terminal to navigate to the folder and ran the following command:`$ docker-compose up -d`

….And voila, new blocks are being mined and Union Square is alive!



Step 5: Adding Our Testnet To MetaMask

- Finally, we added our custom testnet to MetaMask:



Crowdsale

Example of a Crowdsale open to the public using our created network, **Union Square Testnet**

- ERC20 token created and minted through a crowdsale contract (UnionSqaureCrowdsale) that will leverage the OpenZeppelin Solidity library.
- This contract will mint the tokens (SquareCoin) automatically and distribute them to buyers in one transaction
- We conducted the sale through our created testnet "**Union Square Testnet**"

In order for crowdsale contracts to function accurately, smart contracts should be executed in the following order.

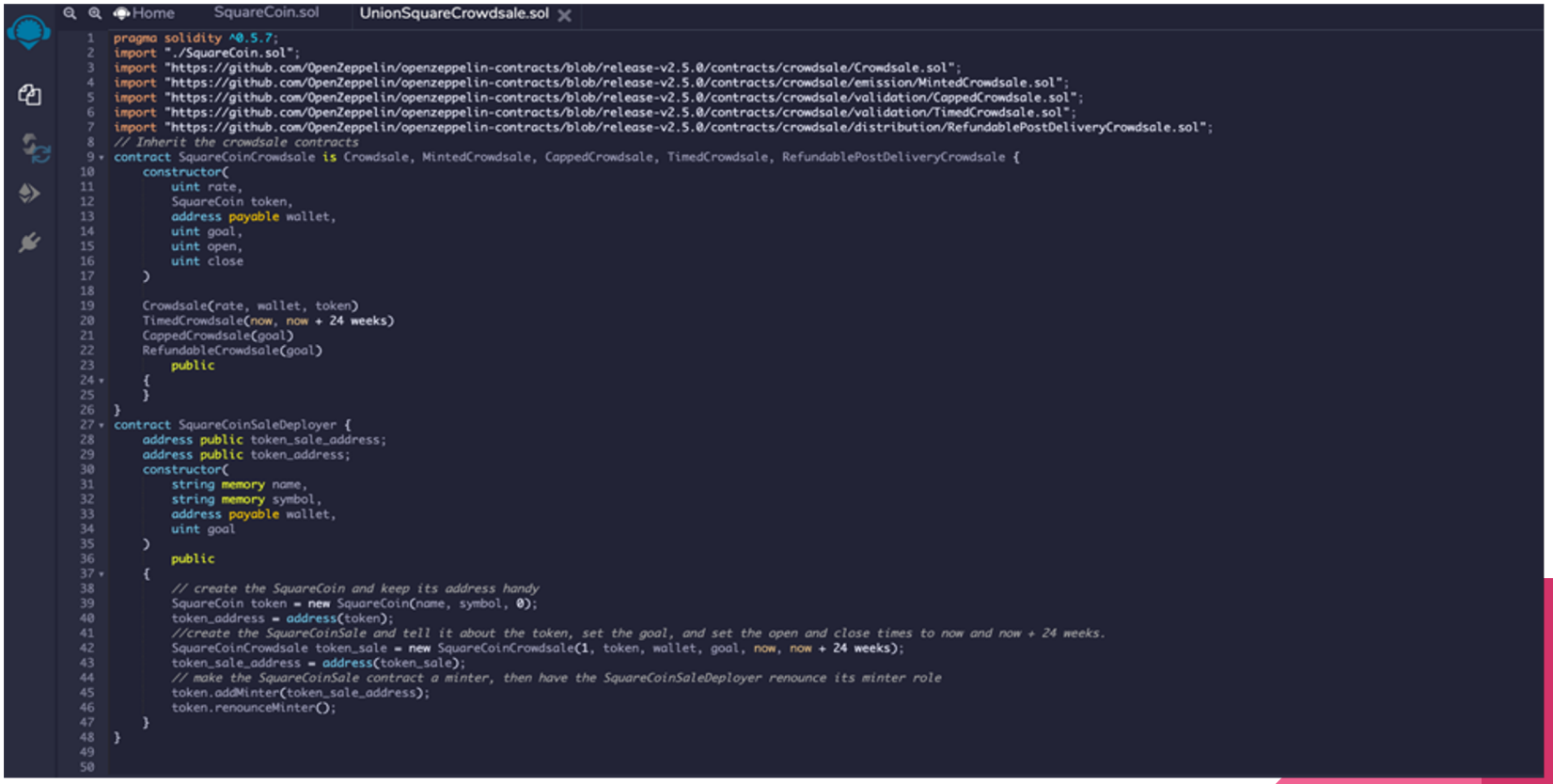
1. Open Ganache and Metamask, change the network to Union Square Testnet. Pre-fund the address to ensure successful deployment of the contract as it would require some Gas.
2. Deployment of the first contract Squarecoin (Solidity codes written in this contract should be imported to UnionSquareCrowdsale.sol). Paramaters required for deployment: name, symbol, initial_supply.
3. Deploy SquareCoinSaleDeployer Contract. Parameters required: name, symbol, wallet (Same as your Wallet Address) and goal.
4. Deploy SquareCoinSale Contract with Token_sale_address in the At_Address section
5. Deploy SquareCoin Contract with Token_Address in the At_Address section
6. Contract Deployed - Check the getter functions to see whether contract has been deployed properly.

Crowdsale Cont’d

SquareCoin.sol:

Crowdsale Cont’d

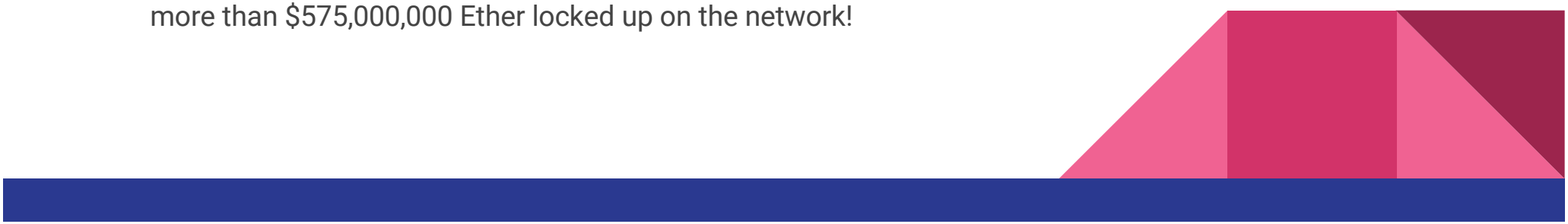
UnionSquareCrowdsale.sol:




```
pragma solidity ^0.5.7;
import "./SquareCoin.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v2.5.0/contracts/crowdsale/Crowdsale.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v2.5.0/contracts/crowdsale/emission/MintedCrowdsale.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v2.5.0/contracts/crowdsale/validation/CappedCrowdsale.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v2.5.0/contracts/crowdsale/validation/TimedCrowdsale.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v2.5.0/contracts/crowdsale/distribution/RefundablePostDeliveryCrowdsale.sol";
// Inherit the crowdsale contracts
contract SquareCoinCrowdsale is Crowdsale, MintedCrowdsale, CappedCrowdsale, TimedCrowdsale, RefundablePostDeliveryCrowdsale {
    constructor(
        uint rate,
        SquareCoin token,
        address payable wallet,
        uint goal,
        uint open,
        uint close
    )

    Crowdsale(rate, wallet, token)
    TimedCrowdsale(now, now + 24 weeks)
    CappedCrowdsale(goal)
    RefundableCrowdsale(goal)
        public
    {
    }
}
contract SquareCoinSaleDeployer {
    address public token_sale_address;
    address public token_address;
    constructor(
        string memory name,
        string memory symbol,
        address payable wallet,
        uint goal
    )
        public
    {
        // create the SquareCoin and keep its address handy
        SquareCoin token = new SquareCoin(name, symbol, 0);
        token_address = address(token);
        //create the SquareCoinSale and tell it about the token, set the goal, and set the open and close times to now and now + 24 weeks.
        SquareCoinCrowdsale token_sale = new SquareCoinCrowdsale(1, token, wallet, goal, now, now + 24 weeks);
        token_sale_address = address(token_sale);
        // make the SquareCoinSale contract a minter, then have the SquareCoinSaleDeployer renounce its minter role
        token.addMinter(token_sale_address);
        token.renounceMinter();
    }
}
```

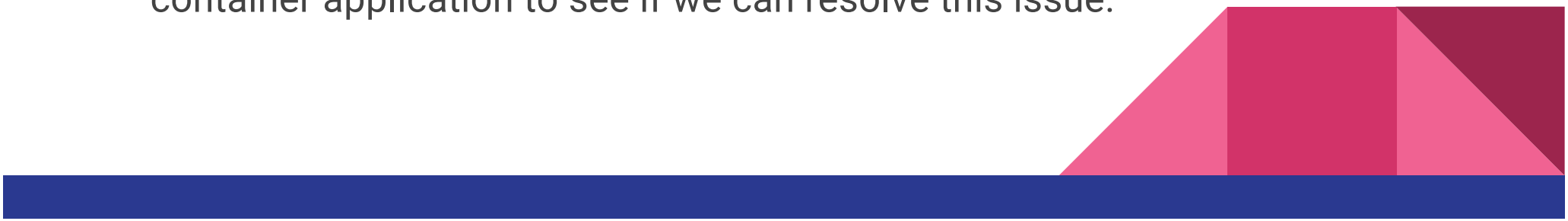
Drawing Comparisons To Medalla & ETH 2.0

- **Medalla** (Aptly named after a subway station in Buenos Aires, Argentina) was ETH 2.0's final testnet before moving forward to Phase 0 of ETH 2.0
  - After years and years of buildup for the upgrade of Ethereum's protocol, and despite a few successful testnets up until this point, Medalla failed pretty significantly and was widely ridiculed throughout the cryptocurrency space.
  - Some of the client's running Medalla started recognizing the wrong time, and this error ended up trickling down throughout the entire network and caused a majority of the nodes to fall out of sync and begin validating blocks that had not actually been created yet.
    - This goes to show just how fragile these networks and just how important decentralization and code really are!
- Despite this mishap, ETH 2.0 launched earlier this month and currently has more than 30,000 validators with more than $575,000,000 Ether locked up on the network!



Post-Mortem

- Unfortunately, we did run into a few issues with our network!
  - We had a decent amount of trouble connecting our computers to one another to successfully sync the network.
  - Even 3 hours of troubleshooting from Garence and Kyle could not solve the issue, which we believe has something to do with my laptop not allowing access from external IP addresses to connect to Union Square.
  - Docker, while seemingly easy to use, was surprisingly difficult to get around and not very composable.
  - Web-browser wallets gave us a lot of headaches as well
- Moving forward, we are going to try to recreate the network using a different container application to see if we can resolve this issue.



Works Cited

- Rotkiewicz, Konrad. "How To Setup A Custom Ethereum Testnet? | Ulam Labs". *Ulam Labs*, 2020, https://www.ulam.io/blog/how-to-setup-custom-ethereum-testnet/. Accessed 7 Dec 2020.
- https://www.bissresearch.com/proof-of-stake-vs-proof-of-work-vs-proof-of-authority/#:~:text=In%20the%20case%20of%20Proof,very%20common%20in%20private%20blockchains.
- https://ethereum.org/en/developers/docs/networks/
- https://cointelegraph.com/news/medalla-testnet-problems-will-not-delay-eth-20-says-prysmatic-labs
- https://consensys.net/blog/blockchain-explained/a-short-history-of-ethereum/
- https://cryptobriefing.com/ethereum-2-0s-medalla-testnet-back-online-after-multi-day-outage/