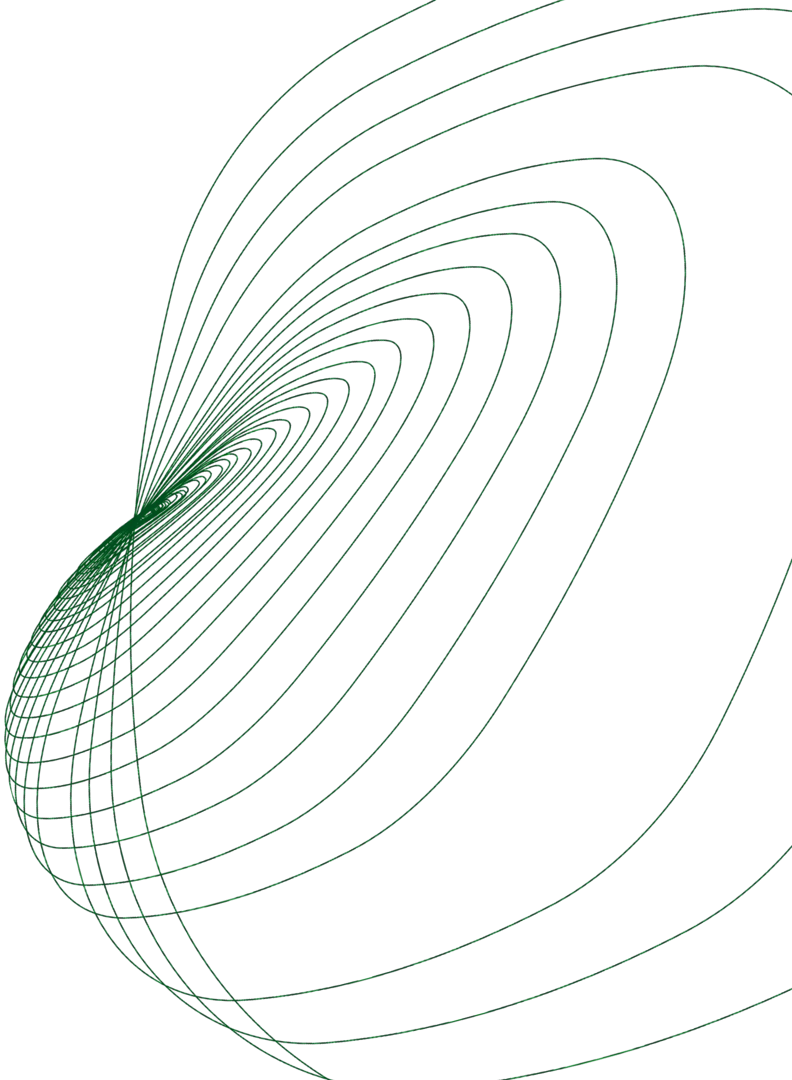
PAXOS

Deploying Dollars on the Blockchain



September 8th, 2021

Today’s Topics

Overview of what we’ll learn

Repo Setup

Application Overview

Deploying the Smart Contract

Mint/Withdrawals

Signing a Transaction

Deploying Dollars on the Blockchain

A whirlwind tour of the Ethereum blockchain, crypto signing, tokenized assets, and web3

What We’ll Learn

By the end of this workshop you should be able to explain to your friends about…

Ethereum Basics

What is a blockchain?

How does Ethereum differ from Bitcoin?

Smart Contracts

What does it mean to deploy code to Ethereum?

What functionality does this unlock?

Transactions

What does it mean to sign a transaction?

How do you call smart contract functions?

Web3 Integrations

As a bonus, we will preview how you can interact with live blockchain based applications.

Repo Setup

To get started with this workshop

1. Fork the workshop repo
https://github.com/paxosglobal/fintech-devcon-2021-stablecoin-workshop

2. Start installing the dependencies (go, docker, and node)

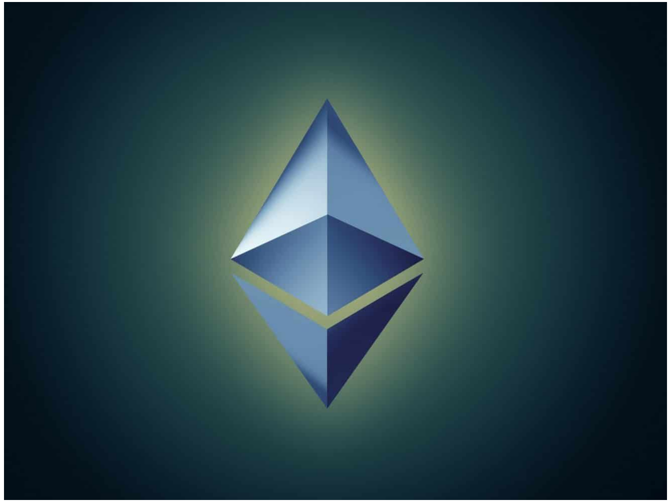
3. Checkout the `exercise 1` branch

4. Open the code in your IDE

Application Features

We'll build an app that can do the following:

- **Deposit** USD into the application
- **Mint** USDK as part of the withdrawal flow
- **Withdraw** USDK to a provided Ethereum address
- **Reconcile** expected USDK balances with on-chain amounts



PAXOS

Application Overview

Main areas to focus on in the repo

Webapp

The webapp exists to make interacting with the application easier.

We will not focus on the code here.

Smart Contract

The smart contract is in `contracts`. USDK uses standard ERC20 implementations by openzeppelin.

API Server

The server code is in `pkg/server`.

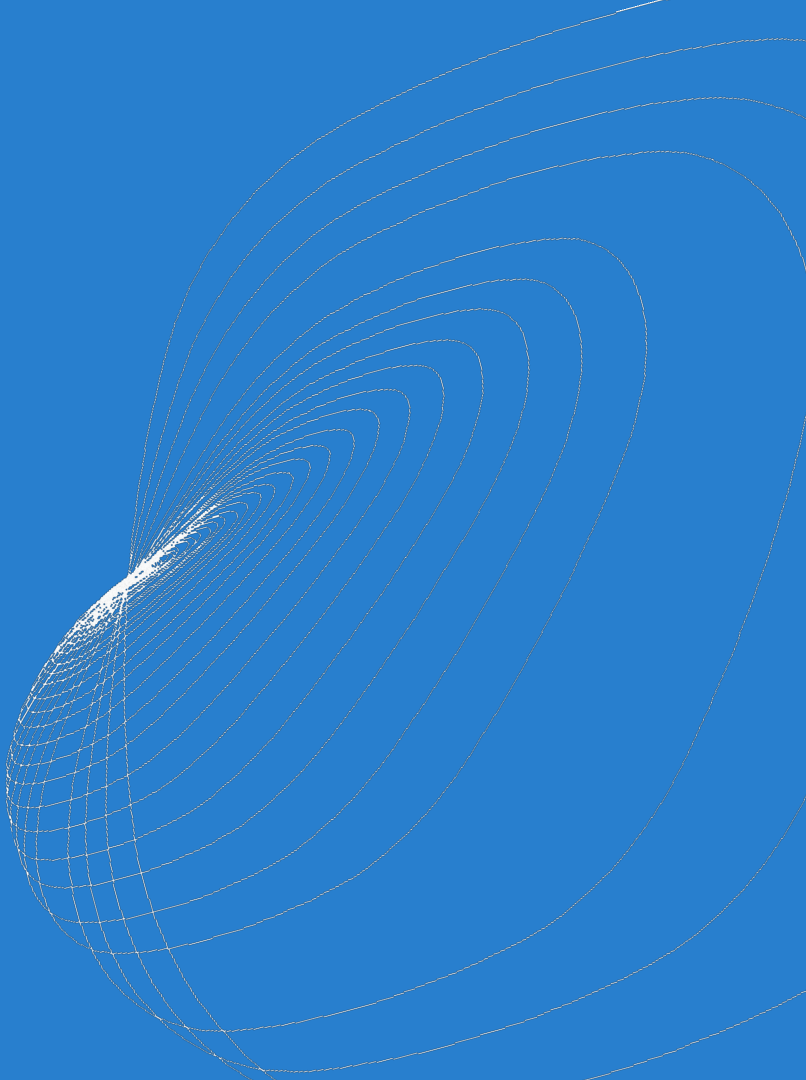
Most of the workshop will focus on this package.

Local Blockchain

Ganache is deployed via Docker for a local Ethereum blockchain.

Our application will interact with Ganache.

Bitcoin vs. Ethereum

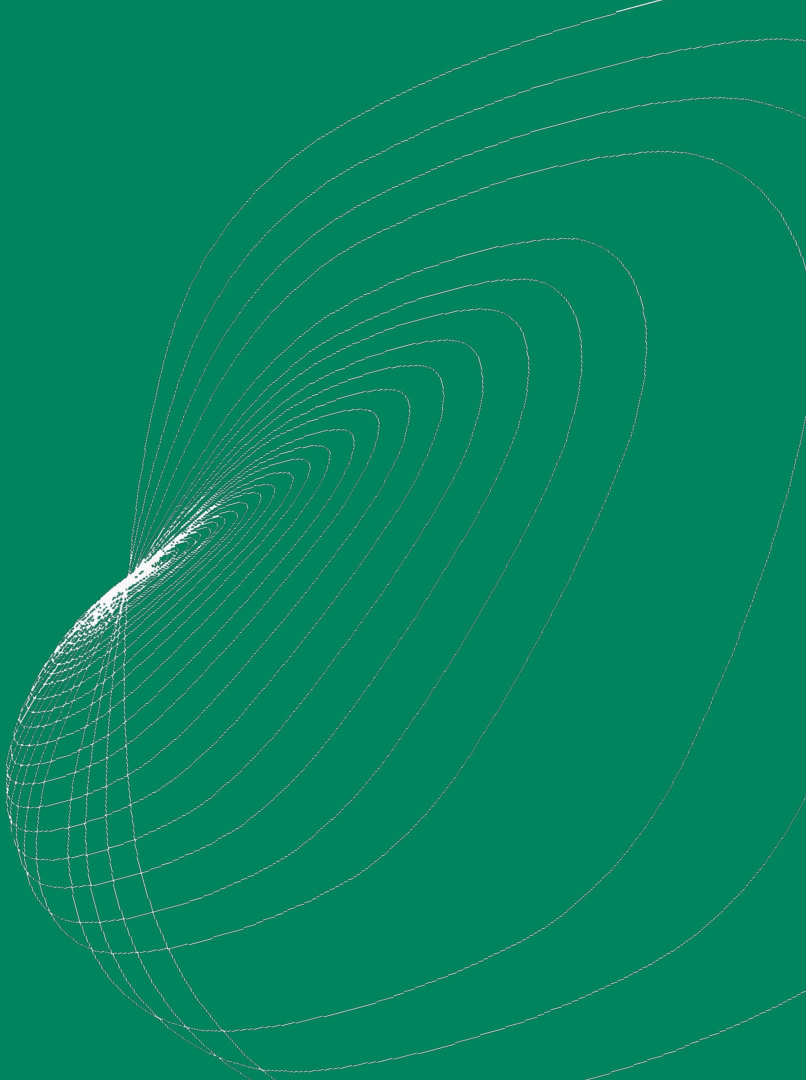


Bitcoin Vs. Ethereum

Ethereum built on top of Bitcoin's innovations

	Bitcoin	Ethereum
Decentralized Blockchain	✔	✔
Non-native Tokens Enables Stablecoins Etc.	✔	✔
Turing Complete Fully Programmable	x	✔

Smart Contracts



What is a smart contract?

Smart Contracts are collections of functions with a distributed DB

- Can store **state** such as address balances and roles
- Has **public** and **private** methods
- Can **call** other smart contracts
- Contract calls are executed by all Ethereum nodes deterministically updating contract state

Smart contracts enable

lending protocols, token exchanges, NFTs etc

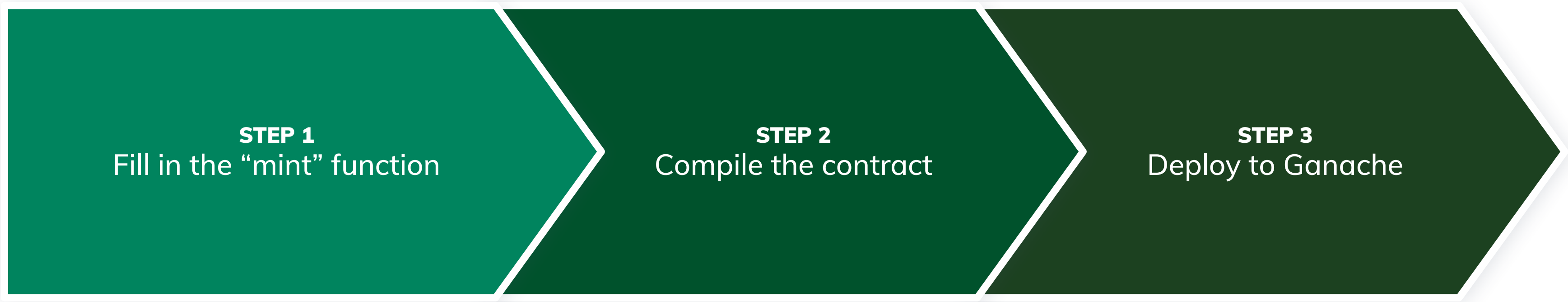
Smart Contract Features

USDK is built on top of tested and audited standards created by OpenZeppelin

	1	2	3	4
Description	ERC20.sol	Ownable.sol	SafeMath.sol	USDK.sol
Feature 1	Adheres to ERC20 standard	Adds administrative features	Wraps solidity arithmetic operations	is ERC20 & Ownable
Feature 2	Enables Transfers	Can set owner of the contract	Protects against underflows	Uses SafeMath
Feature 3	Stores Balances	Can change the owner	Protects against division by zero	Exposes mint/burn to owner

Exercise 1

Finish the smart contract and deploy it



STEP 1
Fill in the “mint” function

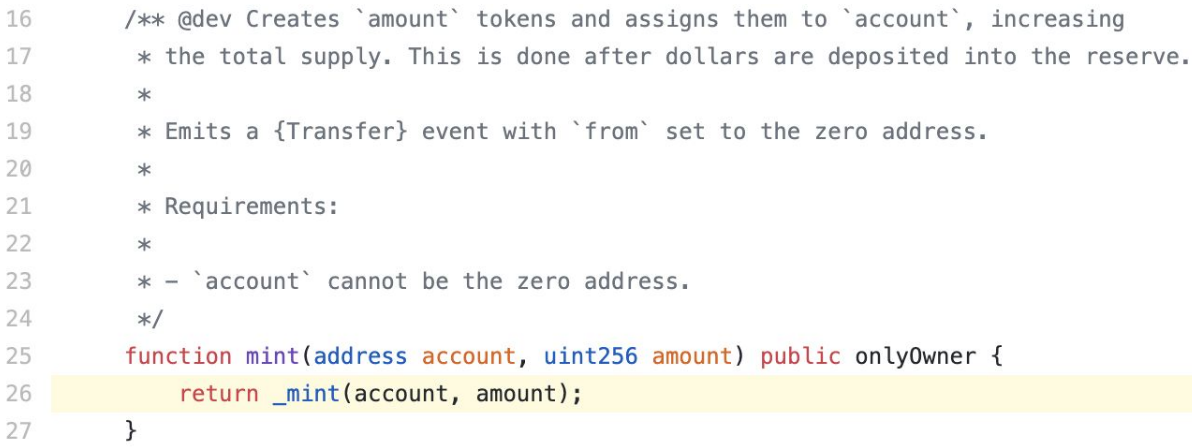
STEP 2
Compile the contract

STEP 3
Deploy to Ganache

* See “bonus point” slide towards the end to learn how to deploy to testnet / mainnet

Mint Function

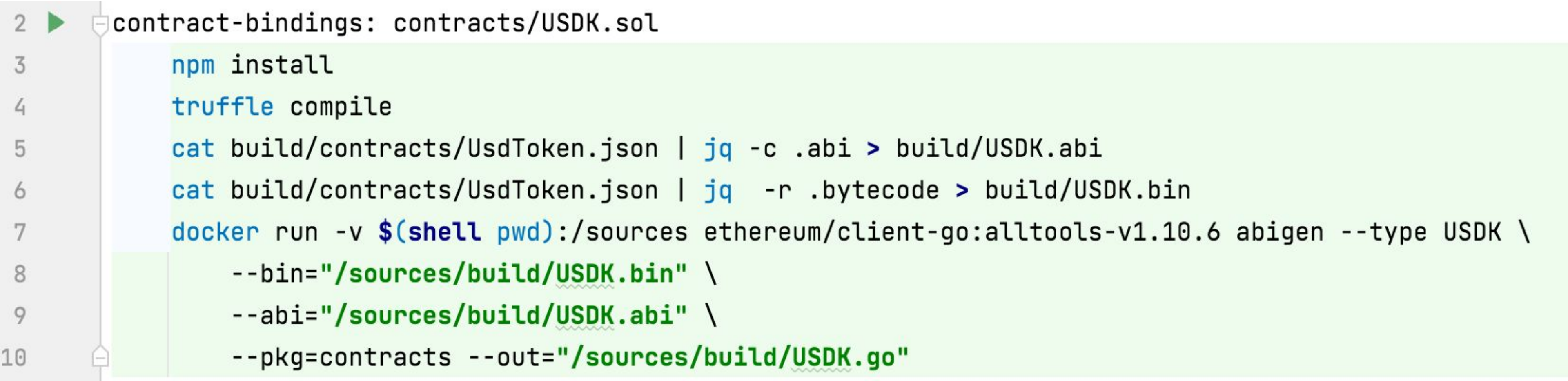
Smart Contracts can call functions on contracts they inherit



```
/** @dev Creates `amount` tokens and assigns them to `account`, increasing
 * the total supply. This is done after dollars are deposited into the reserve.
 *
 * Emits a {Transfer} event with `from` set to the zero address.
 *
 * Requirements:
 *
 * - `account` cannot be the zero address.
 */
function mint(address account, uint256 amount) public onlyOwner {
    return _mint(account, amount);
}
```

Compile USDK

Run `make contract-bindings` to compile and write go bindings

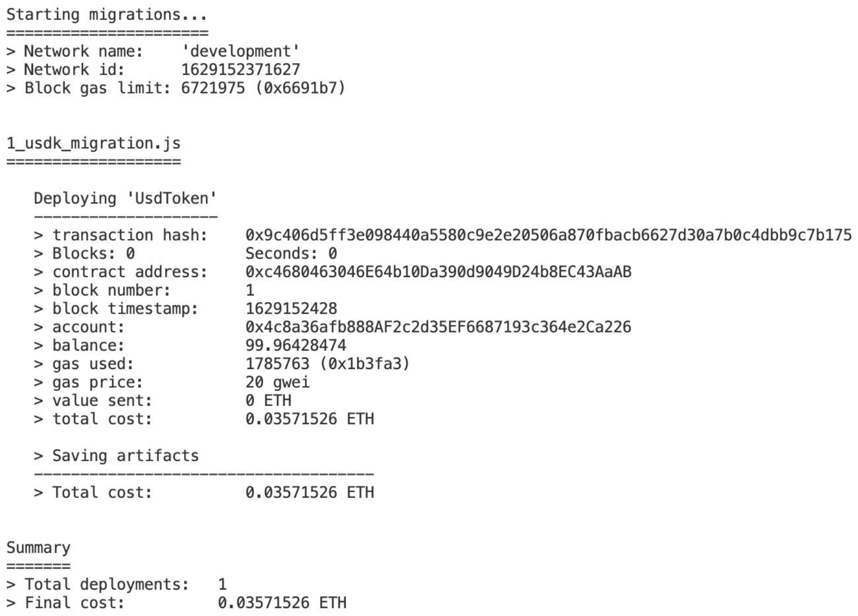


```
contract-bindings: contracts/USDK.sol
    npm install
    truffle compile
    cat build/contracts/UsdToken.json | jq -c .abi > build/USDK.abi
    cat build/contracts/UsdToken.json | jq  -r .bytecode > build/USDK.bin
    docker run -v $(shell pwd):/sources ethereum/client-go:alltools-v1.10.6 abigen --type USDK \
        --bin="/sources/build/USDK.bin" \
        --abi="/sources/build/USDK.abi" \
        --pkg=contracts --out="/sources/build/USDK.go"
```

Deploying to the Local Chain

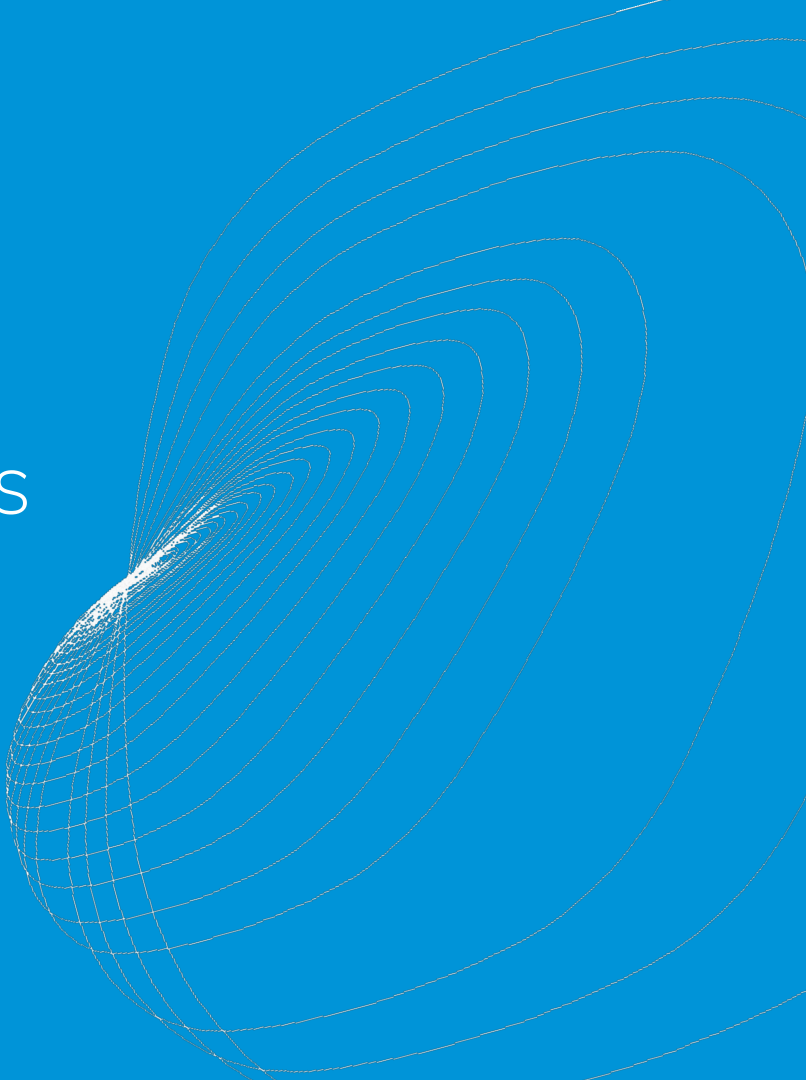
Make sure `make start-local` is running in another terminal

```
func main() {
        ethClient, err :=
ethclient.Dial(server.GanacheNetworkAddr)
        if err != nil {
                panic(err)
        }
        addr, _, _, err :=
contracts.DeployUSDK(server.OwnerTransactor, ethClient)
        if err != nil {
                panic(err)
        }
        log.Print("contract address: ", addr) //
0xc4680463046E64b10Da390d9049D24b8EC43AaAB
}
```



```
Starting migrations...
======================
> Network name:    'development'
> Network id:      1629152371627
> Block gas limit: 6721975 (0x6691b7)


1_usdk_migration.js
===================

   Deploying 'UsdToken'
   --------------------
   > transaction hash:    0x9c406d5ff3e098440a5580c9e2e20506a870fbacb6627d30a7b0c4dbb9c7b175
   > Blocks: 0            Seconds: 0
   > contract address:    0xc4680463046E64b10Da390d9049D24b8EC43AaAB
   > block number:        1
   > block timestamp:     1629152428
   > account:             0x4c8a36afb888AF2c2d35EF6687193c364e2Ca226
   > balance:             99.96428474
   > gas used:            1785763 (0x1b3fa3)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.03571526 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:          0.03571526 ETH


Summary
=======
> Total deployments:   1
> Final cost:          0.03571526 ETH
```

Smart Contract Interactions



What’s in an Ethereum Transaction?

Nonce: Incrementing counter - number of txs previously sent
To: Destination address (note: often a contract address)
Gas: A number of fields that define gas limitations for the tx
Amount: Amount of ETH transferred in this transaction
Sig Data: V, R, & S fields that make up the signature for the transaction
Data: Additional data including data to send to the smart contract

Decoding Transaction Data

Example Data:
0xa9059cbb000000000000000000000000b85d233efaa52d928e4aae460610365ab462b8da0
0000000000000000000000000000000000000000000000006f05b59d3b2000000

0xa9059cbb -> This hex value is derived from taking the method name and its argument types, removing any whitespace, taking the keccak hash of the result, and then taking the first 4 bytes of it and displaying it in hex

000000000000000000000000b85d233efaa52d928e4aae460610365ab462b8da ->
32 byte hex of the “to” address

00000000000000000000000000000000000000000000000006f05b59d3b2000000 ->
32 byte hex of the amount to send

Special Report:

Good News

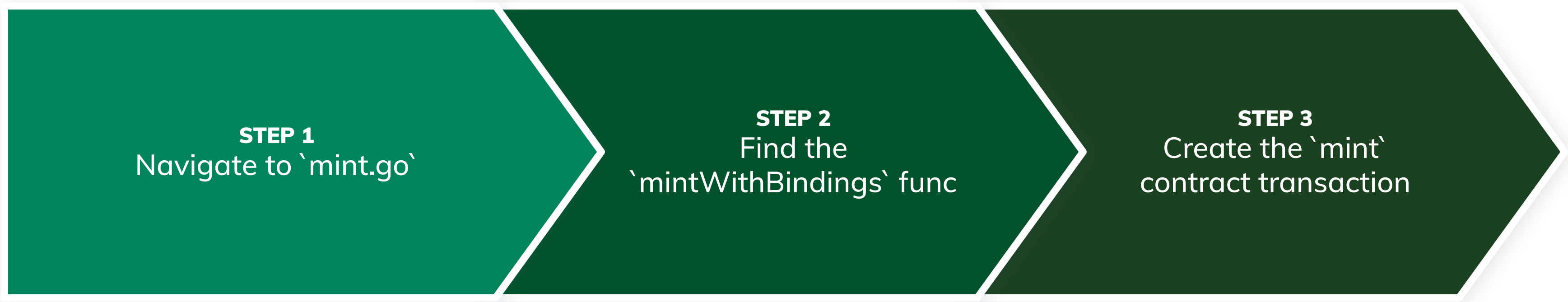
Using contract bindings in Go/JS abstracts this away

Contract Methods	Bindings expose methods for Contract functionality
Arguments	Go’s typing system is integrated
Helper Functions	Bindings include helpers like “WatchTransfer”

```
tx, err := USDK.Transfer(to, amount)
```

Exercise 2

Create a mint transaction

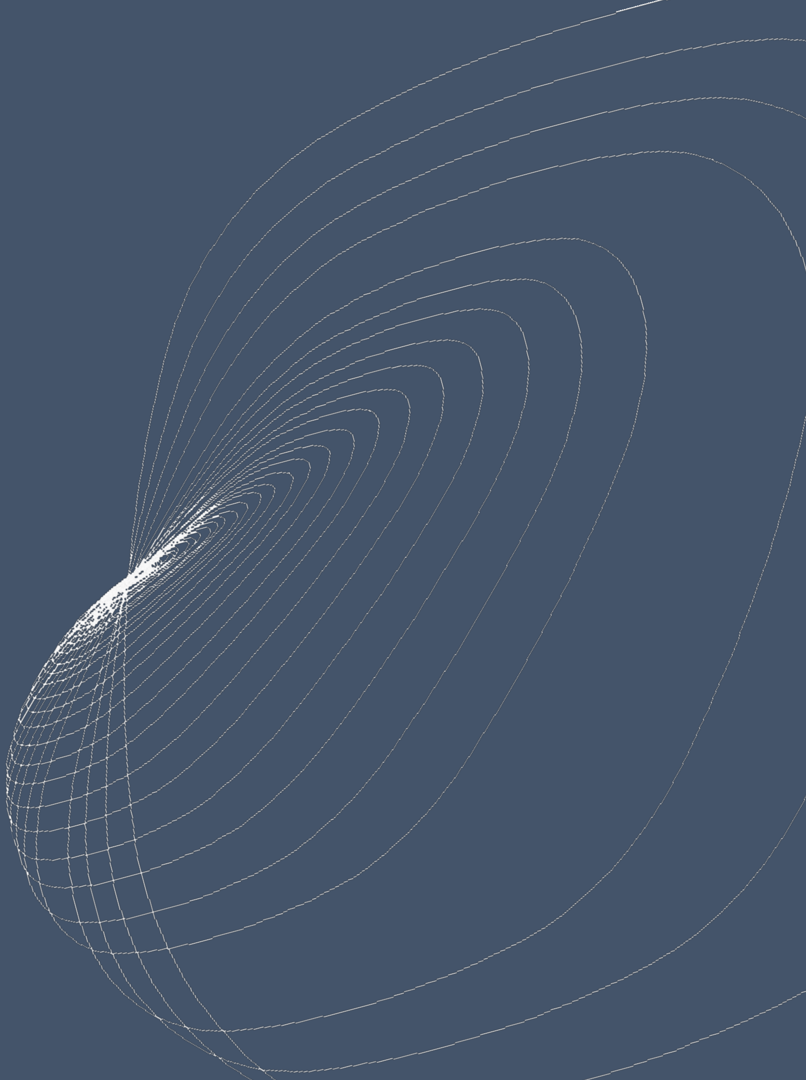


STEP 1
Navigate to `mint.go`

STEP 2
Find the `mintWithBindings` func

STEP 3
Create the `mint` contract transaction

Signing Transactions



PAXOS

Cryptographic Signing Basics

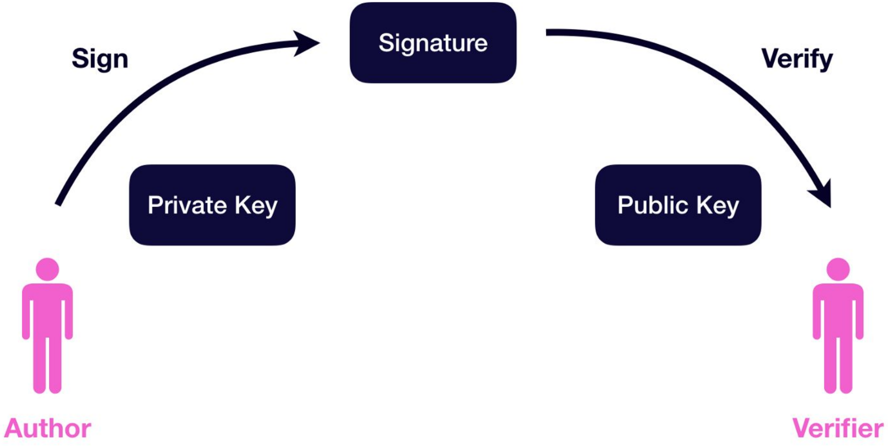
Ethereum signatures uses ECDSA and secp256k1 constants to define the elliptic curve

Asymmetric Key Features

Author can sign a message with their private key

Verifier can use the signature to recover the public key (and the Author's address)

Any change to the message or signature will result in a different public key



Broadcasting A Transaction

You just send the transaction to any Ethereum node and it eventually gets mined!

1. Send the transaction to an Ethereum Node
2. The node communicates to peer nodes who send it throughout the network
3. A miner picks up the transaction and mines it into a block
4. The new block is published back to the network

Blockchain Mining

Blockchain mining is what creates new transactions

- **Fees**. Miners choose transactions that pay them the highest fees. Choosing a higher gas fee leads to faster mining.
- **Hashpower**. Miners try to find the right random hash by creating many hashes until they randomly find one that fits the criteria of the chosen mining problem of the blockchain. The first one to solve it gets the fees and creates the next block

Blockchain mining is brute force protection against changing history

To change history one would have to mine alternative blocks to create a blockchain that makes sense, which would be very expensive in terms of hashpower.

Exercise 3

Explicitly Sign A Mint Transaction

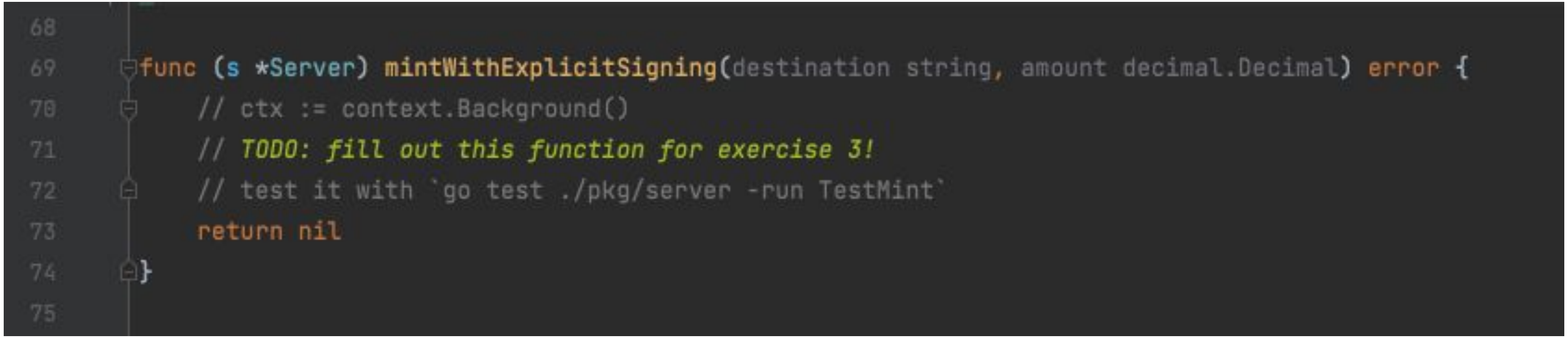
STEP 1
Sign using the test private key

STEP 2
Send to the ganache chain

STEP 3
Ganache will mine it for you!

* See “bonus point” slide towards the end to learn how to deploy to testnet / mainnet

Exercise 3 Code



```
func (s *Server) mintWithExplicitSigning(destination string, amount decimal.Decimal) error {
    // ctx := context.Background()
    // TODO: fill out this function for exercise 3!
    // test it with `go test ./pkg/server -run TestMint`
    return nil
}
```

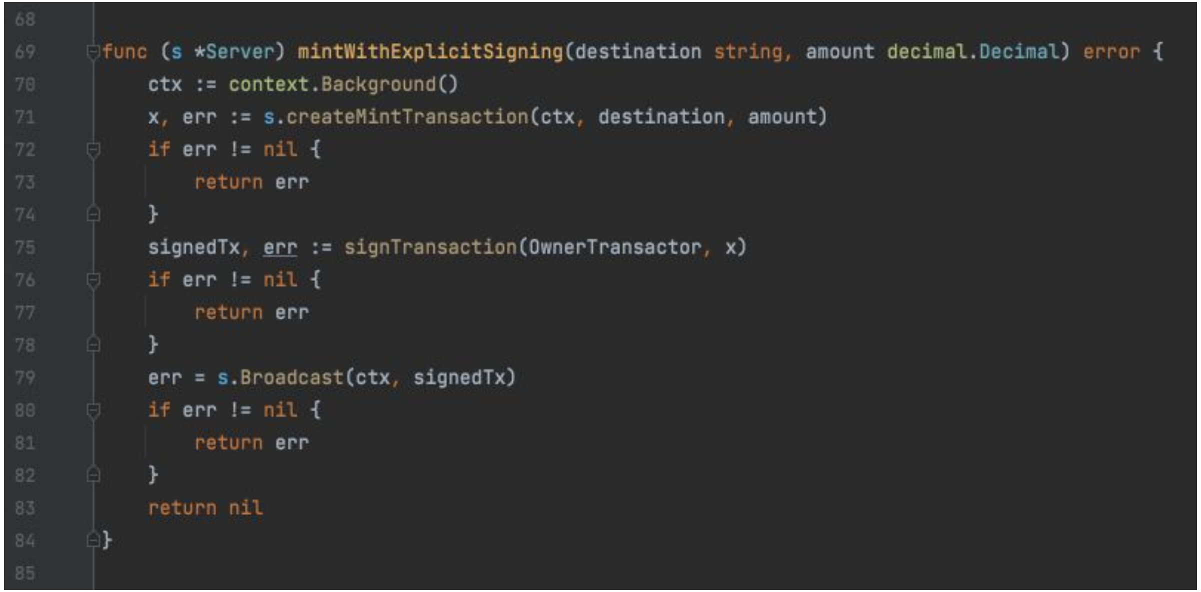
Exercise 3 SOLUTION

Create a Transaction, Sign it, and Broadcast it

To support separate signing

This pattern allows you to take the unsigned transaction and send it over the wire or offline for safer signing.

Cold signing is where you sign in an environment not connected to the internet, using a key that isn't available from the internet.



```go

func (s *Server) mintWithExplicitSigning(destination string, amount decimal.Decimal) error {
	ctx := context.Background()
	x, err := s.createMintTransaction(ctx, destination, amount)
	if err != nil {
		return err
	}
	signedTx, err := signTransaction(OwnerTransactor, x)
	if err != nil {
		return err
	}
	err = s.Broadcast(ctx, signedTx)
	if err != nil {
		return err
	}
	return nil
}

```

Milestone

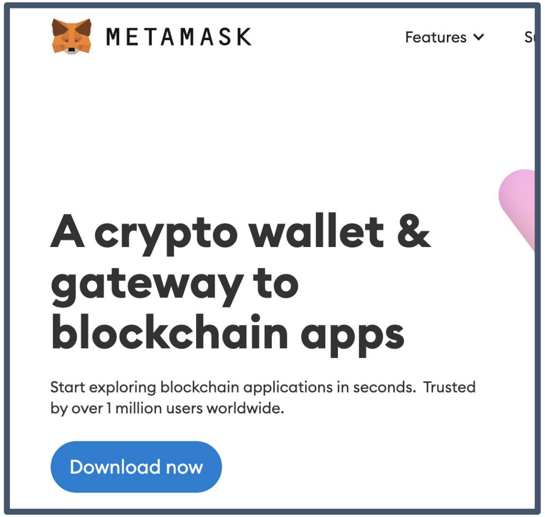
We Created Virtual Dollars!

We have all the pieces to get dollars on the blockchain. Now let’s see them in a user wallet!

Smart Contract	We created a token contract and deployed it.
Transactions	We created token balances by signing and broadcasting a “mint” transaction.
External Wallets	Now we can try looking at the dollars in a metamask wallet and move them around.

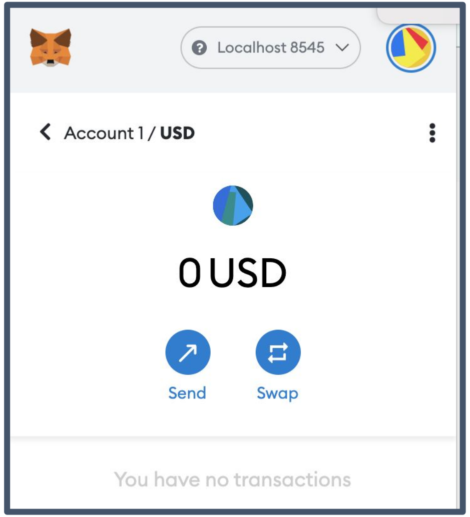
Exercise 4: Use MetaMask To Receive Tokens!

- **Install** for chrome at https://metamask.io/
- **Open the extension** and click “Get Started”
- **Create a Wallet** with a password
- **Switch Network** to Localhost
- **Add the token** in Assets
- **Find Your Address** in Accounts



A crypto wallet & gateway to blockchain apps

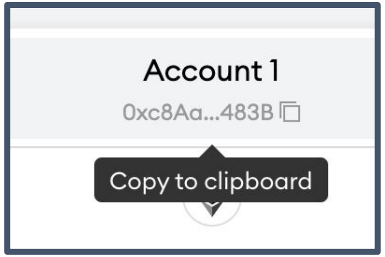
Start exploring blockchain applications in seconds. Trusted by over 1 million users worldwide.



Account 1 / **USD**

0 USD

You have no transactions

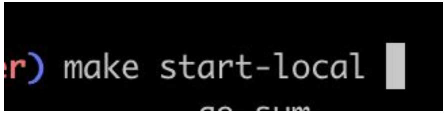


Account 1

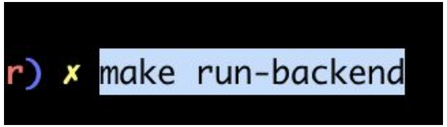
0xc8Aa...483B

Exercise 4: Full Flow

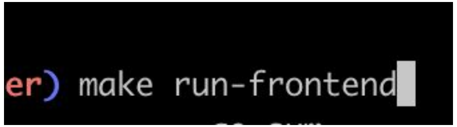
Three Terminals



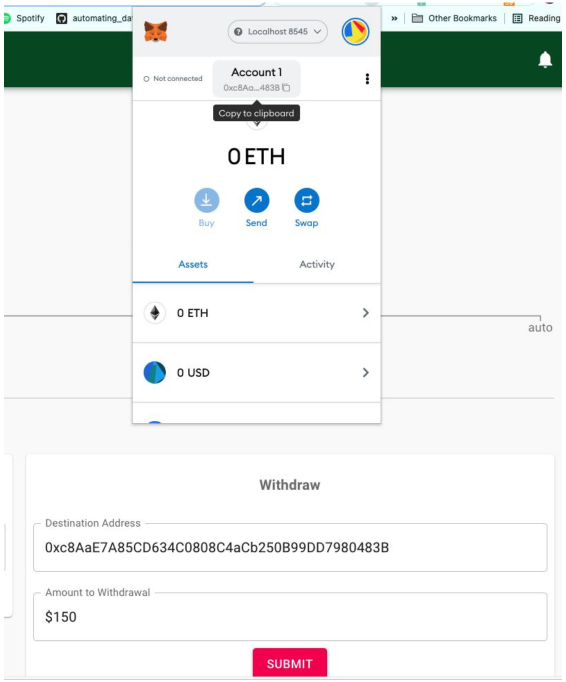
```
r) make start-local
```



```
r) x make run-backend
```



```
er) make run-frontend
```

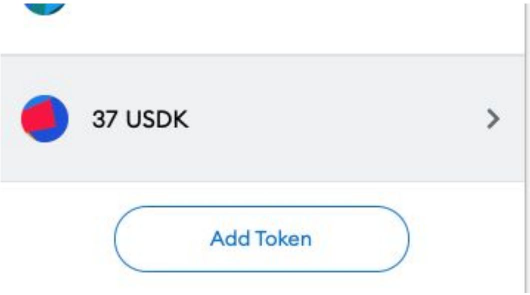


Withdraw

Destination Address: 0xc8AaE7A85CD634C0808C4aCb250B99DD7980483B

Amount to Withdrawal: $150

SUBMIT



Other Topics We didn’t Cover

Handling Token Deposits

At vero eos et accusamus et iusto odio dignissimos ducimus, qui blanditiis praesentium

Reserve Banking

Et harum quidem rerum facilis est et expedita distinctio

Nam libero tempore

Getting on Exchanges

Voluptatum deleniti atque corrupti, qui officia deserunt mollitia animi

Id est laborum et dolorum fuga

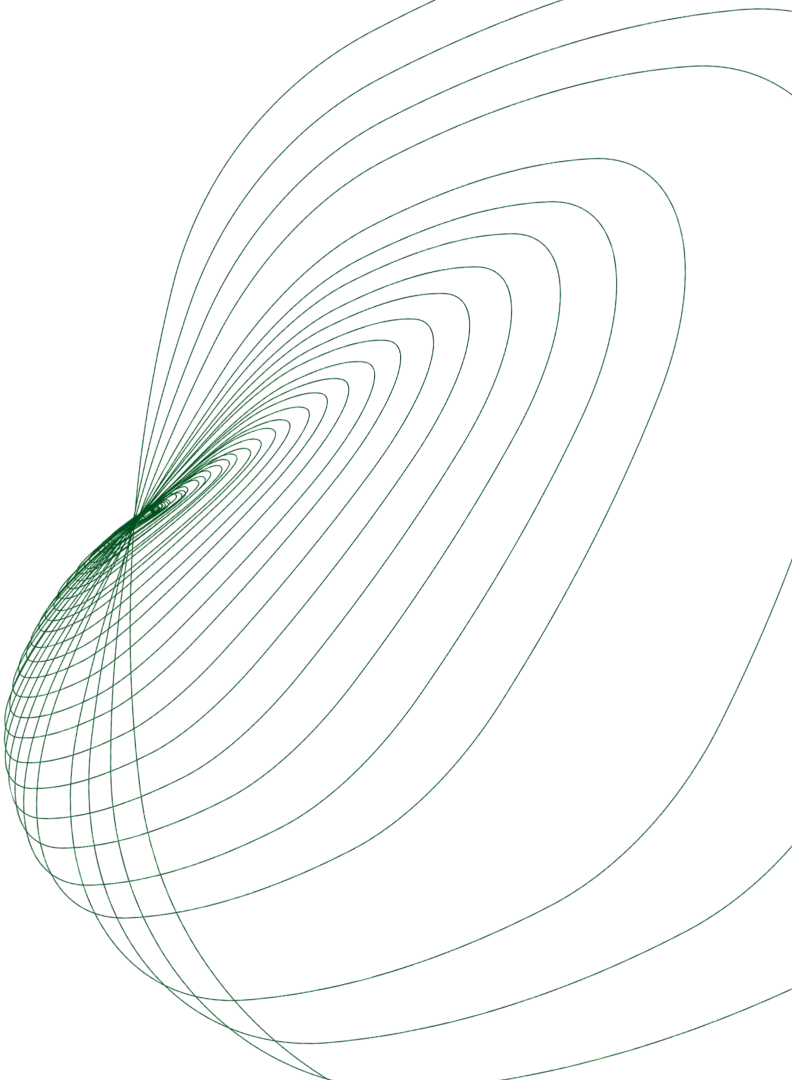
Fiat Network Integrations

Quos dolores et quas molestias excepturi

Obcaecati cupiditate non provident, similique sunt in culpa

PAXOS

Thank you



GET IN TOUCH - We’re Hiring!

Engineering Manager
bperreault@paxos.com

Engineering Manager
igitter@paxos.com