

a little book on

ETHEREUM DEVELOPMENT WITH GO

Miguel Mota



Table of Contents

Introduction	1.1
Client	1.2
Setting up the Client	1.2.1
Accounts	1.3
Account Balances	1.3.1
Account Token Balances	1.3.2
Generating New Wallets	1.3.3
Keystores	1.3.4
HD Wallets	1.3.5
Address Check	1.3.6
Transactions	1.4
Querying Blocks	1.4.1
Querying Transactions	1.4.2
Transferring ETH	1.4.3
Transferring Tokens (ERC-20)	1.4.4
Subscribing to New Blocks	1.4.5
Create Raw Transaction	1.4.6
Send Raw Transaction	1.4.7
Smart Contracts	1.5
Smart Contract Compilation & ABI	1.5.1
Deploying a Smart Contract	1.5.2
Loading a Smart Contract	1.5.3
Querying a Smart Contract	1.5.4
Writing to a Smart Contract	1.5.5
Reading Smart Contract Bytecode	1.5.6
Querying an ERC20 Token Smart Contract	1.5.7
Event Logs	1.6
Subscribing to Event Logs	1.6.1
Reading Event Logs	1.6.2
Reading ERC-20 Token Event Logs	1.6.3
Reading 0x Protocol Event Logs	1.6.4
Signatures	1.7
Generating Signatures	1.7.1

Verifying Signatures	1.7.2
Testing	1.8
Faucets	1.8.1
Using a Simulated Client	1.8.2
Swarm	1.9
Setting Up Swarm	1.9.1
Uploading Files to Swarm	1.9.2
Download Files From Swarm	1.9.3
Whisper	1.10
Connecting Whisper Client	1.10.1
Generating Whisper Key Pair	1.10.2
Sending Messages on Whisper	1.10.3
Subscribing to Whisper Messages	1.10.4
Utilities	1.11
Collection of Utility Functions	1.11.1
Glossary	1.12
Resources	1.13

Ethereum Development with Go

This little guide book is to serve as a general help guide for anyone wanting to develop [Ethereum](#) applications using the [Go](#) programming language. It's meant to provide a starting point if you're already pretty familiar with [Ethereum](#) and [Go](#) but don't know where to start on bringing it all together. You'll learn how to interact with smart contracts and perform general [blockchain](#) tasks and queries using [Golang](#).

This book is composed of many examples that I wish I had encountered before when I first started doing [Ethereum](#) development with [Go](#). This book will walk you through most things that you should be aware of in order for you to be a productive [Ethereum](#) developer using [Go](#).

[Ethereum](#) is quickly evolving and things may [go](#) out of date sooner than anticipated. I strongly suggest opening an [issue](#) or making a [pull request](#) if you observe things that can be improved. This book is completely open and free and available on [github](#).

Online

<https://goethereumbook.org>

E-book

The e-book is available in different formats.

- [PDF](#)
- [EPUB](#)
- [MOBI](#)

Introduction

[Ethereum](#) is an open-source, public, [blockchain](#)-based distributed computing platform and operating system featuring [smart contract](#) (scripting) functionality. It supports a modified version of Nakamoto consensus via transaction based state transitions.

-[Wikipedia](#)

[Ethereum](#) is a [blockchain](#) that allows developers to create applications that can be ran completely decentralized, meaning that no single entity can take it down or modify it. Each application deployed to [Ethereum](#) is executed by every single full client on the [Ethereum](#) network.

Solidity

Solidity is a Turing complete programming language for writing smart contracts. Solidity gets compiled to bytecode which is what the [Ethereum](#) virtual machine executes.

go-ethereum

In this book we'll be using the [go-ethereum](#), the official [Ethereum](#) implementation in [Go](#), to interact with the [blockchain](#). [Go-ethereum](#), also known as [geth](#) for short, is the most popular [Ethereum](#) client and because it's in [Go](#), it provides everything we'll ever need for reading and writing to the [blockchain](#) when developing applications using [Golang](#).

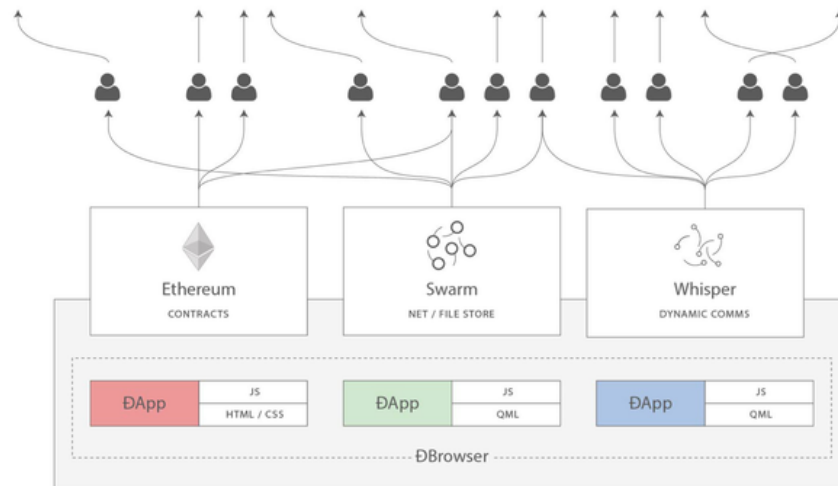
The examples in this book were tested with [go-ethereum](#) version `1.8.10-stable` and [Go](#) version `go1.10.2`.

Block Explorers

[Etherscan](#) is a website for exploring and drilling down on data that lives on the [blockchain](#). These type of websites are known as *Block Explorers* because they allow you to explore the contents of blocks (which contain transactions). Blocks are fundamental components of the [blockchain](#). The block contains the data of all the transactions that have been mined within the allocated block time. The block explorer also allows you to view events that were emitted during the execution of the [smart contract](#) as well as things such as how much was paid for the [gas](#) and amount of [ether](#) was transacted, etc.

Swarm and Whisper

We'll also be diving a little bit into [Swarm](#) and [Whisper](#), a file storage protocol, and a [peer-to-peer](#) messaging protocol respectively, which are the other two pillars required for achieving completely decentralized and distributed applications.



[image credit](#)

Support

Join the [#ethereum](#) channel on the [gophers slack](#) for [Go \(golang\)](#) help.

The [Ethereum StackExchange](#) is also a great place to ask general [Ethereum](#) question and [Go](#) specific questions.

About the Author

This book was written by [Miguel Mota](#), a software developer from Los Angeles working in the [blockchain](#) space. You can find him on Twitter [@miguelmotah](#)

Enough with the introduction, let's get [started](#)!

Client

The client is the entry point to the [Ethereum](#) network. The client is required to broadcast transactions and read [blockchain](#) data. In the [next section](#) will learn how to set up a client in a [Go](#) application.

Setting up the Client

Setting up the [Ethereum](#) client in [Go](#) is a fundamental step required for interacting with the [blockchain](#). First import the `ethclient` [go-ethereum](#) package and initialize it by calling `Dial` which accepts a provider URL.

You can connect to the [infura](#) gateway if you don't have an existing client. [Infura](#) manages a bunch of [Ethereum](#) [[geth](#) and [parity](#)] nodes that are secure, reliable, scalable and lowers the barrier to entry for newcomers when it comes to plugging into the [Ethereum](#) network.

```
client, err := ethclient.Dial("https://mainnet.infura.io")
```

You may also pass the path to the IPC endpoint file if you have a local instance of [geth](#) running.

```
client, err := ethclient.Dial("/home/user/.ethereum/geth.ipc")
```

Using the `ethclient` is a necessary thing you'll need to start with for every [Go Ethereum](#) project and you'll be seeing this step a lot throughout this book.

Using Ganache

[Ganache](#) (formally known as *testrpc*) is an [Ethereum](#) implementation written in [Node.js](#) meant for testing purposes while developing dapps locally. Here we'll walk you through how to install it and connect to it.

First install ganache via [NPM](#).

```
npm install -g ganache-cli
```

Then run the ganache CLI client.

```
ganache-cli
```

Now connect to the ganache RPC host on `http://localhost:8545`.


```
client, err := ethclient.Dial("http://localhost:8545")
if err != nil {
    log.Fatal(err)
}
```

You may also use the same [mnemonic](#) when starting ganache to generate the same sequence of public [addresses](#).

```
ganache-cli -m "much repair shock carbon improve miss forget sock include"
```

I highly recommend getting familiar with ganache by reading their [documentation](#).

Full code

[client.go](#)

```
package main

import (
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("we have a connection")
    _ = client // we'll use this in the upcoming sections
}
```

Accounts

Accounts on [Ethereum](#) are either [wallet addresses](#) or [smart contract addresses](#). They look like

0x71c7656ec7ab88b098defb751b7401b5f6d8976f and they're what you use for sending ETH to another user and also are used for referring to a [smart contract](#) on the [blockchain](#) when needing to interact with it. They are unique and are derived from a [private key](#). We'll [go](#) more in depth into private/public key pairs in later sections.

In order to use account `addresses` with `go-ethereum`, you must first convert them to the `go-ethereum` `common.Address` type.

```
address := common.HexToAddress("0x71c7656ec7ab88b098defb751b7401b5f6d8")
fmt.Println(address.Hex()) // 0x71C7656EC7ab88b098defB751B7401B5f6d8
```

Pretty much you'd use this type anywhere you'd pass an [ethereum](#) address to methods from [go-ethereum](#). Now that you know the basics of accounts and [addresses](#), let's learn how to retrieve the ETH account balance in the next section.

Full code

address.go

```
package main

import (
    "fmt"

    "github.com/ethereum/go-ethereum/common"
)

func main() {
    address := common.HexToAddress("0x71c7656ec7ab88b098defb751b7401b")

    fmt.Println(address.Hex())      // 0x71C7656EC7ab88b098defB751B7401B
    fmt.Println(address.Hash().Hex()) // 0x000000000000000000000000000071c7656ec7ab88b098defb751b7401b
    fmt.Println(address.Bytes())     // [113 199 101 110 199 171 136 176 152 201]
}
```


Account Balances

Reading the balance of an account is pretty simple; call the `BalanceAt` method of the client passing it the account address and optional block number. Setting `nil` as the block number will return the latest balance.

```
account := common.HexToAddress("0x71c7656ec7ab88b098defb751b74011
balance, err := client.BalanceAt(context.Background(), account, nil)
if err != nil {
    log.Fatal(err)
}

fmt.Println(balance) // 25893180161173005034
```

Passing the block number lets you read the account balance at the time of that block. The block number must be a `big.Int`.

```
blockNumber := big.NewInt(5532993)
balance, err := client.BalanceAt(context.Background(), account, blockNumber)
if err != nil {
    log.Fatal(err)
}

fmt.Println(balance) // 25729324269165216042
```

Numbers in [ethereum](#) are dealt using the smallest possible unit because they're fixed-point precision, which in the case of ETH it's *wei*. To read the ETH value you must do the calculation $\text{wei} / 10^{18}$. Because we're dealing with big numbers we'll have to import the native [Go](#) `math` and `math/big` packages. Here's how'd you do the conversion.

```
fbalance := new(big.Float)
fbalance.SetString(balance.String())
ethValue := new(big.Float).Quo(fbalance, big.NewFloat(math.Pow10(18)))

fmt.Println(ethValue) // 25.729324269165216041
```

Pending balance

Sometimes you'll want to know what the pending account balance is, for example after submitting or waiting for a transaction to be confirmed. The client provides a similar method to `BalanceAt` called

PendingBalanceAt which accepts the account address as a parameter.

```
pendingBalance, err := client.PendingBalanceAt(context.Background(), acco
fmt.Println(pendingBalance) // 25729324269165216042
```

Full code

[account_balance.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"
    "math"
    "math/big"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    account := common.HexToAddress("0x71c7656ec7ab88b098defb751b740")
    balance, err := client.BalanceAt(context.Background(), account, nil)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(balance) // 25893180161173005034

    blockNumber := big.NewInt(5532993)
    balanceAt, err := client.BalanceAt(context.Background(), account, blockNumber)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(balanceAt) // 25729324269165216042

    fbalance := new(big.Float)
    fbalance.SetString(balanceAt.String())
    ethValue := new(big.Float).Quo(fbalance, big.NewFloat(math.Pow10(18)))
    fmt.Println(ethValue) // 25.729324269165216041

    pendingBalance, err := client.PendingBalanceAt(context.Background(), account)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(pendingBalance) // 25729324269165216042
}

```

Account Token Balances

To learn how to read account [token](#) (ERC20) balances, head over to the [section on ERC20 token smart contracts](#).

Generating New Wallets

To generate a new [wallet](#) first we need to import the [go-ethereum](#) `crypto` package that provides the `GenerateKey` method for generating a random [private key](#).

```
privateKey, err := crypto.GenerateKey()
if err != nil {
    log.Fatal(err)
}
```

Then we can convert it to bytes by importing the [golang](#) `crypto/ecdsa` package and using the `FromECDSA` method.

```
privateKeyBytes := crypto.FromECDSA(privateKey)
```

We can now convert it to a hexadecimal string by using the [go-ethereum](#) `hexutil` package which provides the `Encode` method which takes a byte slice. Then we strip off the `0x` after it's hex encoded.

```
fmt.Println(hexutil.Encode(privateKeyBytes)[2:]) // fad9c8855b740a0b7ed4c
```

This is the [private key](#) which is used for signing transactions and is to be treated like a password and never be shared, since who ever is in possession of it will have access to all your funds.

Since the public key is derived from the [private key](#), [go-ethereum](#)'s `crypto` [private key](#) has a `Public` method that will return the public key.

```
publicKey := privateKey.Public()
```

Converting it to hex is a similar process that we went through with the [private key](#). We strip off the `0x` and the first 2 characters `04` which is always the EC prefix and is not required.


```
publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
if !ok {
    log.Fatalf("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
}

publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)
fmt.Println(hexutil.Encode(publicKeyBytes)[4:]) // 9a7df67f79246283fdc93af
```

Now that we have the public key we can easily generate the public address which is what you're used to seeing. In order to do that, the [go-ethereum](#) crypto package has a `PubkeyToAddress` method which accepts an ECDSA public key, and returns the public address.

```
address := crypto.PubkeyToAddress(*publicKeyECDSA).Hex()
fmt.Println(address) // 0x96216849c49358B10257cb55b28eA603c874b05E
```

The public address is simply the [Keccak-256](#) hash of the public key, and then we take the last 40 characters (20 bytes) and prefix it with `0x`. Here's how you can do it manually using the `crypto/sha3` `keccak256` function.

```
hash := sha3.NewLegacyKeccak256()
hash.Write(publicKeyBytes[1:])
fmt.Println(hexutil.Encode(hash.Sum(nil)[12:])) // 0x96216849c49358b10257
```

Full code

[generate_wallet.go](#)

```

package main

import (
    "crypto/ecdsa"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/crypto"
    "golang.org/x/crypto/sha3"
)

func main() {
    privateKey, err := crypto.GenerateKey()
    if err != nil {
        log.Fatal(err)
    }

    privateKeyBytes := crypto.FromECDSA(privateKey)
    fmt.Println(hexutil.Encode(privateKeyBytes)[2:]) // fad9c8855b740a0b7ed

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
    }

    publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)
    fmt.Println(hexutil.Encode(publicKeyBytes)[4:]) // 9a7df67f79246283fdc93

    address := crypto.PubkeyToAddress(*publicKeyECDSA).Hex()
    fmt.Println(address) // 0x96216849c49358B10257cb55b28eA603c874b05

    hash := sha3.NewLegacyKeccak256()
    hash.Write(publicKeyBytes[1:])
    fmt.Println(hexutil.Encode(hash.Sum(nil))[12:]) // 0x96216849c49358b102
}

```

Keystores

A [keystore](#) is a file containing an encrypted [wallet private key](#). Keystores in [go-ethereum](#) can only contain one [wallet](#) key pair per file. To generate keystores first you must invoke `NewKeyStore` giving it the directory path to save the keystores. After that, you may generate a new [wallet](#) by calling the method `NewAccount` passing it a password for [encryption](#). Every time you call `NewAccount` it will generate a new [keystore](#) file on disk.

Here's a full example of generating a new [keystore](#) account.

```
ks := keystore.NewKeyStore("./wallets", keystore.StandardScryptN, keystore
password := "secret"
account, err := ks.NewAccount(password)
if err != nil {
    log.Fatal(err)
}

fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426f2
```

Now to import your [keystore](#) you basically need to invoke `NewKeyStore` again as usual and then call the `Import` method which accepts the [keystore](#) JSON data as bytes. The second argument is the password used to encrypt it in order to decrypt it. The third argument is to specify a new [encryption](#) password but we'll use the same one in the example. Importing the account will give you access to the account as expected but it'll generate a new [keystore](#) file! There's no point in having two of the same thing so we'll delete the old one.

Here's an example of importing a [keystore](#) and accessing the account.

```
file := "./wallets/UTC--2018-07-04T09-58-30.122808598Z--20f8d42fb0f667f2e
ks := keystore.NewKeyStore("./tmp", keystore.StandardScryptN, keystore.Sta
jsonBytes, err := ioutil.ReadFile(file)
if err != nil {
    log.Fatal(err)
}

password := "secret"
account, err := ks.Import(jsonBytes, password, password)
if err != nil {
    log.Fatal(err)
}

fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426f2

if err := os.Remove(file); err != nil {
    log.Fatal(err)
}
```

Full code

[keystore.go](https://github.com/ethereum/go-ethereum/blob/master/cmd/evm/cmd/import.go)

```

package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "os"

    "github.com/ethereum/go-ethereum/accounts/keystore"
)

func createKs() {
    ks := keystore.NewKeyStore("./tmp", keystore.StandardScryptN, keystore.
    password := "secret"
    account, err := ks.NewAccount(password)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426
}

func importKs() {
    file := ".tmp/UTC--2018-07-04T09-58-30.122808598Z--20f8d42fb0f667f2e
    ks := keystore.NewKeyStore("./tmp", keystore.StandardScryptN, keystore.
    jsonBytes, err := ioutil.ReadFile(file)
    if err != nil {
        log.Fatal(err)
    }

    password := "secret"
    account, err := ks.Import(jsonBytes, password, password)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426

    if err := os.Remove(file); err != nil {
        log.Fatal(err)
    }
}

func main() {
    createKs()
    //importKs()
}

```


HD Wallet

For creating or using an [HD wallet](#), please refer to the [Go](#) package:
<https://github.com/miguelmota/go-ethereum-hdwallet>

Address Check

This section will describe how to validate an address and determine if it's a [smart contract](#) address.

Check if Address is Valid

We can use a simple regular expression to check if the [ethereum](#) address is valid:

```
re := regexp.MustCompile("^0x[0-9a-fA-F]{40}$")

fmt.Printf("is valid: %v\n", re.MatchString("0x323b5d4c32345ced77393b353"))
fmt.Printf("is valid: %v\n", re.MatchString("0xZYXb5d4c32345ced77393b353"))
```

Check if Address in an Account or a Smart Contract

We can determine if an address is a [smart contract](#) if there's bytecode stored at that address. Here's an example where we fetch the code for a [token smart contract](#) and check the length to verify that it's a [smart contract](#):

```
// 0x Protocol Token (ZRX) smart contract address
address := common.HexToAddress("0xe41d2489571d322189246dafa5ebde")
bytecode, err := client.CodeAt(context.Background(), address, nil) // nil is latest block
if err != nil {
    log.Fatal(err)
}

isContract := len(bytecode) > 0

fmt.Printf("is contract: %v\n", isContract) // is contract: true
```

When there's no bytecode at the address then we know that it's not a [smart contract](#) and it's a standard [ethereum](#) account:

Setting up the Client

```
// a random user account address
address := common.HexToAddress("0x8e215d06ea7ec1fdb4fc5fd21768f4b3
bytecode, err := client.CodeAt(context.Background(), address, nil) // nil is lat
if err != nil {
    log.Fatal(err)
}

isContract = len(bytecode) > 0

fmt.Printf("is contract: %v\n", isContract) // is contract: false
```

Full code

[address_check.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"
    "regexp"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    re := regexp.MustCompile("^0x[0-9a-fA-F]{40}$")

    fmt.Printf("is valid: %v\n", re.MatchString("0x323b5d4c32345ced77393b3"))
    fmt.Printf("is valid: %v\n", re.MatchString("0xZYXb5d4c32345ced77393b3"))

    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // 0x Protocol Token (ZRX) smart contract address
    address := common.HexToAddress("0xe41d2489571d322189246dafa5ebc")
    bytecode, err := client.CodeAt(context.Background(), address, nil) // nil is nil
    if err != nil {
        log.Fatal(err)
    }

    isContract := len(bytecode) > 0

    fmt.Printf("is contract: %v\n", isContract) // is contract: true

    // a random user account address
    address = common.HexToAddress("0x8e215d06ea7ec1fdb4fc5fd21768f4")
    bytecode, err = client.CodeAt(context.Background(), address, nil) // nil is nil
    if err != nil {
        log.Fatal(err)
    }

    isContract = len(bytecode) > 0

    fmt.Printf("is contract: %v\n", isContract) // is contract: false
}

```

Transactions

These sections will discuss how to query and make transactions on [Ethereum](#) using the [go-ethereum](#) `ethclient` package.

Querying Blocks

There's two ways you can query block information as we'll see.

Block header

You can call the client's `HeaderByNumber` to return header information about a block. It'll return the latest block header if you pass `nil`.

```
header, err := client.HeaderByNumber(context.Background(), nil)
if err != nil {
    log.Fatal(err)
}

fmt.Println(header.Number.String()) // 5671744
```

Full block

Call the client's `BlockByNumber` method to get the full block. You can read all the contents and metadata of the block such as block number, block timestamp, block hash, block [difficulty](#), as well as the list of transactions and much much more.

```
blockNumber := big.NewInt(5671744)
block, err := client.BlockByNumber(context.Background(), blockNumber)
if err != nil {
    log.Fatal(err)
}

fmt.Println(block.Number().Uint64()) // 5671744
fmt.Println(block.Time().Uint64()) // 1527211625
fmt.Println(block.Difficulty().Uint64()) // 3217000136609065
fmt.Println(block.Hash().Hex()) // 0x9e8751ebb5069389b855bba72d949
fmt.Println(len(block.Transactions())) // 144
```

Call `TransactionCount` to return just the count of transactions in a block.

```
count, err := client.TransactionCount(context.Background(), block.Hash())
if err != nil {
    log.Fatal(err)
}

fmt.Println(count) // 144
```

In the next section we'll learn how to query transactions in a block.

Full code

blocks.go

```

package main

import (
    "context"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    header, err := client.HeaderByNumber(context.Background(), nil)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(header.Number.String()) // 5671744

    blockNumber := big.NewInt(5671744)
    block, err := client.BlockByNumber(context.Background(), blockNumber)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(block.Number().Uint64()) // 5671744
    fmt.Println(block.Time().Uint64()) // 1527211625
    fmt.Println(block.Difficulty().Uint64()) // 3217000136609065
    fmt.Println(block.Hash().Hex()) // 0x9e8751ebb5069389b855bba72d9
    fmt.Println(len(block.Transactions())) // 144

    count, err := client.TransactionCount(context.Background(), block.Hash())
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(count) // 144
}

```

Querying Transactions

In the [previous section](#) we learned how to read a block and all its data given the block number. We can read the transactions in a block by calling the `Transactions` method which returns a list of `Transaction` type. It's then trivial to iterate over the collection and retrieve any information regarding the transaction.

```
for _, tx := range block.Transactions() {
    fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d
    fmt.Println(tx.Value().String()) // 10000000000000000
    fmt.Println(tx.Gas()) // 105000
    fmt.Println(tx.GasPrice().Uint64()) // 102000000000
    fmt.Println(tx.Nonce()) // 110644
    fmt.Println(tx.Data()) // []
    fmt.Println(tx.To().Hex()) // 0x55fE59D8Ad77035154dDd0AD0388D09C
}
```

In order to read the sender address, we need to call `AsMessage` on the transaction which returns a `Message` type containing a function to return the sender (from) address. The `AsMessage` method requires the EIP155 signer, which we derive the chain ID from the client.

```
chainID, err := client.NetworkID(context.Background())
if err != nil {
    log.Fatal(err)
}

if msg, err := tx.AsMessage(types.NewEIP155Signer(chainID)); err != nil {
    fmt.Println(msg.From().Hex()) // 0xfD081e3Bb178dc45c0cb23202069ddA5
}
```

Each transaction has a receipt which contains the result of the execution of the transaction, such as any return values and logs, as well as the status which will be `1` (success) or `0` (fail).

```
receipt, err := client.TransactionReceipt(context.Background(), tx.Hash())
if err != nil {
    log.Fatal(err)
}

fmt.Println(receipt.Status) // 1
fmt.Println(receipt.Logs) // ...
```

Another way to iterate over transaction without fetching the block is to call the client's `TransactionInBlock` method. This method accepts only the block hash and the index of the transaction within the block. You can call `TransactionCount` to know how many transactions there are in the block.

```
blockHash := common.HexToHash("0x9e8751ebb5069389b855bba72d9490
count, err := client.TransactionCount(context.Background(), blockHash)
if err != nil {
    log.Fatal(err)
}

for idx := uint(0); idx < count; idx++ {
    tx, err := client.TransactionInBlock(context.Background(), blockHash, idx)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d420f
}
```

You can also query for a single transaction directly given the transaction hash by using `TransactionByHash`.

```
txHash := common.HexToHash("0x5d49fcaa394c97ec8a9c3e7bd9e8388d42
tx, isPending, err := client.TransactionByHash(context.Background(), txHash)
if err != nil {
    log.Fatal(err)
}

fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d420f
fmt.Println(isPending)      // false
```

Full code

Setting up the Client

transactions.go

```

package main

import (
    "context"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    blockNumber := big.NewInt(5671744)
    block, err := client.BlockByNumber(context.Background(), blockNumber)
    if err != nil {
        log.Fatal(err)
    }

    for _, tx := range block.Transactions() {
        fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e838
        fmt.Println(tx.Value().String()) // 10000000000000000
        fmt.Println(tx.Gas()) // 105000
        fmt.Println(tx.GasPrice().Uint64()) // 102000000000
        fmt.Println(tx.Nonce()) // 110644
        fmt.Println(tx.Data()) // []
        fmt.Println(tx.To().Hex()) // 0x55fE59D8Ad77035154dDd0AD0388D

        chainID, err := client.NetworkID(context.Background())
        if err != nil {
            log.Fatal(err)
        }

        if msg, err := tx.AsMessage(types.NewEIP155Signer(chainID)); err == nil {
            fmt.Println(msg.From().Hex()) // 0x0fD081e3Bb178dc45c0cb2320206
        }

        receipt, err := client.TransactionReceipt(context.Background(), tx.Hash())
        if err != nil {
            log.Fatal(err)
        }
    }
}

```

```

    fmt.Println(receipt.Status) // 1
}

blockHash := common.HexToHash("0x9e8751ebb5069389b855bba72d94
count, err := client.TransactionCount(context.Background(), blockHash)
if err != nil {
    log.Fatal(err)
}

for idx := uint(0); idx < count; idx++ {
    tx, err := client.TransactionInBlock(context.Background(), blockHash, id
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d4
}

txHash := common.HexToHash("0x5d49fcaa394c97ec8a9c3e7bd9e8388d
tx, isPending, err := client.TransactionByHash(context.Background(), txHa
if err != nil {
    log.Fatal(err)
}

fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d420
fmt.Println(isPending)      // false
}

```

Transferring ETH

In this lesson you'll learn how to transfer ETH from one account to another account. If you're already familiar with [Ethereum](#) then you know that a transaction consists of the amount of [ether](#) you're transferring, the [gas](#) limit, the [gas](#) price, a [nonce](#), the receiving address, and optionally data. The transaction must be signed with the [private key](#) of the sender before it's broadcasted to the network.

Assuming you've already connected a client, the next step is to load your [private key](#).

```
privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad0f:
if err != nil {
    log.Fatal(err)
}
```

Afterwards we need to get the account [nonce](#). Every transaction requires a [nonce](#). A [nonce](#) by definition is a number that is only used once. If it's a new account sending out a transaction then the [nonce](#) will be 0. Every new transaction from an account must have a [nonce](#) that the previous [nonce](#) incremented by 1. It's hard to keep manual track of all the nonces so the [ethereum](#) client provides a helper method `PendingNonceAt` that will return the next [nonce](#) you should use.

The function requires the public address of the account we're sending from -- which we can derive from the [private key](#).

```
publicKey := privateKey.Public()
publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
if !ok {
    log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
}

fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
```

Here, `privateKey.Public()` returns an interface that contains our public key. We perform a type assertion with `publicKey.(expectedType)` to explicitly set the type of our `publicKey` variable, and assign it to `publicKeyECDSA`. This allows us to use it where our program expects an input of type `*ecdsa.PublicKey`.

Now we can read the `nonce` that we should use for the account's transaction.

```
nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
if err != nil {
    log.Fatal(err)
}
```

The next step is to set the amount of ETH that we'll be transferring. However we must convert `ether` to `wei` since that's what the `Ethereum blockchain` uses. `Ether` supports up to 18 decimal places so 1 ETH is 1 plus 18 zeros. Here's a little tool to help you convert between ETH and wei: <https://etherconverter.netlify.com>

```
value := big.NewInt(1000000000000000000) // in wei (1 eth)
```

The `gas` limit for a standard ETH transfer is `21000` units.

```
gasLimit := uint64(21000) // in units
```

The `gas` price must be set in wei. At the time of this writing, a `gas` price that will get your transaction included pretty fast in a block is 30 gwei.

```
gasPrice := big.NewInt(30000000000) // in wei (30 gwei)
```

However, `gas` prices are always fluctuating based on market demand and what users are willing to pay, so hardcoding a `gas` price is sometimes not ideal. The `go-ethereum` client provides the `SuggestGasPrice` function for getting the average `gas` price based on `x` number of previous blocks.

```
gasPrice, err := client.SuggestGasPrice(context.Background())
if err != nil {
    log.Fatal(err)
}
```

We figure out who we're sending the ETH to.

```
toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4fa1
```

Now we can finally generate our unsigned [ethereum](#) transaction by importing the [go-ethereum](#) `core/types` package and invoking `NewTransaction` which takes in the [nonce](#), to address, value, [gas](#) limit, [gas](#) price, and optional data. The data field is `nil` for just sending ETH. We'll be using the data field when it comes to interacting with smart contracts.

```
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, nil)
```

The next step is to sign the transaction with the [private key](#) of the sender. To do this we call the `SignTx` method that takes in the unsigned transaction and the [private key](#) that we constructed earlier. The `SignTx` method requires the EIP155 signer, which we derive the chain ID from the client.

```
chainID, err := client.NetworkID(context.Background())
if err != nil {
    log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKe
```

Now we are finally ready to broadcast the transaction to the entire network by calling `SendTransaction` on the client which takes in the signed transaction.

```
err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s", signedTx.Hash().Hex()) // tx sent: 0x77006fcb3938f6
```

Afterwards you can check the progress on a block explorer such as Etherscan:

<https://rinkeby.etherscan.io/tx/0x77006fcb3938f648e2cc65bafd27dec30b9bfbe9df41f78498b9c8b7322a249e>

Full code

[transfer_eth.go](#)

```

package main

import (
    "context"
    "crypto/ecdsa"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad")
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(1000000000000000000) // in wei (1 eth)
    gasLimit := uint64(21000)                // in units
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f")
    var data []byte
    tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, d

```



```
chainID, err := client.NetworkID(context.Background())
if err != nil {
    log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), private
if err != nil {
    log.Fatal(err)
}

err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s", signedTx.Hash().Hex())
}
```

Transferring Tokens (ERC-20)

This section will walk you through on how to transfer [ERC-20](#) tokens. To learn how to transfer other types of tokens that are non-[ERC-20](#) compliant check out the [section on smart contracts](#) to learn how to interact with smart contracts.

To transfer [ERC-20](#) tokens, we'll need to broadcast a transaction to the [blockchain](#) just like before, but with a few changed parameters:

- Instead of setting a `value` for the broadcasted transaction, we'll need to embed the value of tokens to transfer in the `data` send in the transaction.
- Construct a contract function call and embed it in the `data` field of the transaction we're broadcasting to the [blockchain](#).

We'll assume that you've already completed the previous [section on transferring ETH](#), and have a [Go](#) application that has:

1. Connected a client.
2. Loaded your account [private key](#).
3. Configured the [gas](#) price to use for your transaction.

Creating a Token for testing

You can create a [token](#) using the [Token Factory](#) <https://tokenfactory.surge.sh>, a website for conveniently deploying [ERC-20](#) [token](#) contracts, to follow the examples in this guide.

When you create your [ERC-20](#) [Token](#), be sure to note down the **address of the token contract**.

For demonstration purposes, I've created a [token](#) (HelloToken HTN) using the [Token](#) Factory and deployed it to the [Rinkeby](#) testnet at the [token](#) contract address

```
0x28b149020d2152179873ec60bed6bf7cd705775d .
```

You can check it out with a Web3-enabled browser here (make sure to be connected to the [Rinkeby](#) testnet in MetaMask):

<https://tokenfactory.surge.sh/#/token/0x28b149020d2152179873ec60bed6bf7cd705775d>

ETH value and destination address

First, we'll set a few variables.

Set the `value` of the transaction to 0.

```
value := big.NewInt(0)
```

This `value` is the amount of ETH to be transferred for this transaction, which should be 0 since we're transferring [ERC-20 Tokens](#) and not ETH. We'll set the value of Tokens to be transferred in the `data` field later.

Then, store the address you'll be sending tokens to in a variable.

```
toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4fa1
```

Forming the data field

Now the fun part. We'll need to figure out what goes into the `data` field of the transaction. This is the message that we broadcast to the [blockchain](#) as part of the transaction.

To make a [token](#) transfer, we need to use this data field to invoke a function on the [smart contract](#). For more information on the functions available on an [ERC-20 token](#) contract, see the [ERC-20 Token Standard specification](#).

To transfer tokens from our active account to another, we need to invoke the `transfer()` function in our [ERC-20 token](#) in our transactions data field. We do this by doing the following:

1. Figure out the function [signature](#) of the `transfer()` [smart contract](#) function we'll be calling.
2. Figure out the inputs for the function — the `address` of the [token](#) recipients, and the `value` of tokens to be transferred.
3. Get the first 8 characters (4 bytes) of the Keccak256 hash of that function [signature](#). This is the *method ID* of the contract function we're invoking.
4. Zero-pad (on the left) the inputs of our function call — the `address` and `value`. These input values need to be 256-bits (32 bytes) long.

First, let's assign the [token](#) contract address to a variable.

```
tokenAddress := common.HexToAddress("0x28b149020d2152179873ec60b
```

Next, we need to form the [smart contract](#) function call. The [signature](#) of the function we'll be calling is the `transfer()` function in the [ERC-20](#) specification, and the types of the argument we'll be passing to it. The first argument type is `address` (the address to which we're sending tokens), and the second argument's type is `uint256` (the amount of tokens to send). The result is the string `transfer(address,uint256)` (no spaces!).

We need this function [signature](#) as a byte slice, which we assign to `transferFnSignature` :

```
transferFnSignature := []byte("transfer(address,uint256)") // do not include s
```

We then need to get the `methodID` of our function. To do this, we'll import the `crypto/sha3` to generate the Keccak256 hash of the function [signature](#). The first 4 bytes of the resulting hash is the `methodID` :

```
hash := sha3.NewLegacyKeccak256()
hash.Write(transferFnSignature)
methodID := hash.Sum(nil)[:4]
fmt.Println(hexutil.Encode(methodID)) // 0xa9059cbb
```

Next we'll zero pad (to the left) the account address we're sending tokens. The resulting byte slice must be 32 bytes long:

```
paddedAddress := common.LeftPadBytes(toAddress.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAddress)) // 0x0000000000000000000000000000000000000000000000000000000000000000
```

Next we'll set the value tokens to send as a `*big.Int` number. Note that the denomination used here is determined by the [token](#) contract that you're interacting with, and **not** in ETH or wei.

For example, if we were working with TokenA where 1 [token](#) is set as the smallest unit of TokenA (i.e. the `decimal()` value of the [token](#) contract is 0 ; for more information, see the [ERC-20 Token Standard specification](#)), then `amount := big.NewInt(1000)` would set `amount` to 1000 units of TokenA.

The example [token](#) we're using, HelloToken, uses 18 decimals which is standard practice for [ERC-20](#) tokens. This means that in order to represent 1 [token](#) we have to do the calculation `amount * 10^18`. In this

example we'll use 1,000 tokens so we'll need to calculate $1000 * 10^{18}$ which is $1e+21$ or `1000000000000000000000`. This is the value the [smart contract](#) understands as 1,000 tokens from a user representation.

```
amount := new(big.Int)
amount.SetString("1000000000000000000000", 10) // sets the value to 1000
```

There are utility functions available in the [utils](#) section to easily do these conversions.

Left padding to 32 bytes will also be required for the amount since the [EVM](#) use 32 byte wide data structures.

```
paddedAmount := common.LeftPadBytes(amount.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAmount)) // 0x000000000000000000000000
```

Now we concatenate the method ID, padded address, and padded amount into a byte slice that will be our data field.

```
var data []byte
data = append(data, methodID...)
data = append(data, paddedAddress...)
data = append(data, paddedAmount...)
```

Set gas limit

The [gas](#) limit will depend on the size of the transaction data and computational steps that the [smart contract](#) has to perform. Fortunately the client provides the `EstimateGas` method which is able to estimate the [gas](#) for us based on the most recent state of the [blockchain](#). This function takes a `CallMsg` struct from the `ethereum` package where we specify the data and the address of the [token](#) contract to which we're sending the function call message. It'll return the estimated [gas](#) limit units we'll use to generate the complete transaction.

```
gasLimit, err := client.EstimateGas(context.Background(), ethereum.CallMsg{
    To: &tokenAddress,
    Data: data,
})
if err != nil {
    log.Fatal(err)
}

fmt.Println(gasLimit) // 23256
```

NOTE: The [gas](#) limit set by the `EstimateGas()` method is based on the current state of the [blockchain](#), and is just an *estimate*. If your transactions are constantly failing, or if you prefer to have full control over the amount of [gas](#) your application spends, you may want to set this value manually.

Create transaction

Now we have all the information we need to generate the transaction.

We'll create a transaction similar the one we used in [section on transferring ETH](#), EXCEPT that the `to` field should contain the [token smart contract](#) address, and the value field should be set to `0` since we're not transferring ETH. This is a gotcha that confuses people.

```
tx := types.NewTransaction(nonce, tokenAddress, value, gasLimit, gasPrice,
```

The next step is to sign the transaction with the [private key](#) of the sender. The `SignTx` method requires the EIP155 signer, which we derive the chain ID from the client.

```
chainID, err := client.NetworkID(context.Background())
if err != nil {
    log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKe
```

And finally, broadcast the transaction:

```
err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s", signedTx.Hash().Hex()) // tx sent: 0xa56316b637a94
```

You can check the progress on Etherscan:

<https://rinkeby.etherscan.io/tx/0xa56316b637a94c4cc0331c73ef26389d6c097506d581073f927275e7a6ece0bc>

To learn how to load and interact with an ERC20 [smart contract](#), check out the [section on ERC20 token smart contracts](#).

Full code

[transfer_tokens.go](#)

```

package main

import (
    "context"
    "crypto/ecdsa"
    "fmt"
    "log"
    "math/big"

    "golang.org/x/crypto/sha3"
    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad")
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(0) // in wei (0 eth)
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f")

```



```
tokenAddress := common.HexToAddress("0x28b149020d2152179873ec60")

transferFnSignature := []byte("transfer(address,uint256)")
hash := sha3.NewLegacyKeccak256()
hash.Write(transferFnSignature)
methodID := hash.Sum(nil)[:4]
fmt.Println(hexutil.Encode(methodID)) // 0xa9059cbb

paddedAddress := common.LeftPadBytes(toAddress.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAddress)) // 0x00000000000000000000000000000000

amount := new(big.Int)
amount.SetString("10000000000000000000", 10) // sets the value to 10

paddedAmount := common.LeftPadBytes(amount.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAmount)) // 0x00000000000000000000000000000000

var data []byte
data = append(data, methodID...)
data = append(data, paddedAddress...)
data = append(data, paddedAmount...)

gasLimit, err := client.EstimateGas(context.Background(), ethereum.CallMsg{
    To: &tokenAddress,
    Data: data,
})
if err != nil {
    log.Fatal(err)
}
fmt.Println(gasLimit) // 23256

tx := types.NewTransaction(nonce, tokenAddress, value, gasLimit, gasPrice)

chainID, err := client.NetworkID(context.Background())
if err != nil {
    log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKey)
if err != nil {
    log.Fatal(err)
}

err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
    log.Fatal(err)
}
```

Setting up the Client

```
fmt.Printf("tx sent: %s", signedTx.Hash().Hex()) // tx sent: 0xa56316b637a9  
}
```

Subscribing to New Blocks

In this section we'll go over how to set up a subscription to get events when there is a new block mined. First thing is we need an [Ethereum](#) provider that supports RPC over websockets. In this example we'll use the [infura](#) websocket endpoint.

```
client, err := ethclient.Dial("wss://ropsten.infura.io/ws")
if err != nil {
    log.Fatal(err)
}
```

Next we'll create a new channel that will be receiving the latest block headers.

```
headers := make(chan *types.Header)
```

Now we call the client's `SubscribeNewHead` method which takes in the headers channel we just created, which will return a subscription object.

```
sub, err := client.SubscribeNewHead(context.Background(), headers)
if err != nil {
    log.Fatal(err)
}
```

The subscription will push new block headers to our channel so we'll use a select statement to listen for new messages. The subscription object also contains an error channel that will send a message in case of a failure with the subscription.

```
for {
    select {
    case err := <-sub.Err():
        log.Fatal(err)
    case header := <-headers:
        fmt.Println(header.Hash().Hex()) // 0xbc10defa8dda384c96a17640d84de5
    }
}
```

To get the full contents of the block, we can pass the block header hash to the client's `BlockByHash` function.

```
block, err := client.BlockByHash(context.Background(), header.Hash())
if err != nil {
    log.Fatal(err)
}

fmt.Println(block.Hash().Hex())    // 0xbc10defa8dda384c96a17640d84de5
fmt.Println(block.Number().Uint64()) // 3477413
fmt.Println(block.Time())         // 1529525947
fmt.Println(block.Nonce())        // 130524141876765836
fmt.Println(len(block.Transactions())) // 7
```

As you can see, you can read the entire block's metadata fields, list of transactions, and much more.

Full code

block_subscribe.go

```

package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("wss://ropsten.infura.io/ws")
    if err != nil {
        log.Fatal(err)
    }

    headers := make(chan *types.Header)
    sub, err := client.SubscribeNewHead(context.Background(), headers)
    if err != nil {
        log.Fatal(err)
    }

    for {
        select {
        case err := <-sub.Err():
            log.Fatal(err)
        case header := <-headers:
            fmt.Println(header.Hash().Hex()) // 0xbc10defa8dda384c96a17640d84

            block, err := client.BlockByHash(context.Background(), header.Hash())
            if err != nil {
                log.Fatal(err)
            }

            fmt.Println(block.Hash().Hex()) // 0xbc10defa8dda384c96a17640c
            fmt.Println(block.Number().Uint64()) // 3477413
            fmt.Println(block.Time()) // 1529525947
            fmt.Println(block.Nonce()) // 130524141876765836
            fmt.Println(len(block.Transactions())) // 7
        }
    }
}

```

Create Raw Transaction

If you've read the [previous sections](#), then you know how to load your [private key](#) to sign transactions. We'll assume you know how to do that by now and now you want to get the raw transaction data to be able to broadcast it at a later time.

First construct the transaction object and sign it, for example:

```
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, data)

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKey)
if err != nil {
    log.Fatal(err)
}
```

Now before we can get the transaction in raw bytes format we'll need to initialize a `types.Transactions` type with the signed transaction as the first value.

```
ts := types.Transactions{signedTx}
```

The reason for doing this is because the `Transactions` type provides a `GetRlp` method for returning the transaction in [RLP](#) encoded format. [RLP](#) is a special encoding method [Ethereum](#) uses for serializing objects. The result of this is raw bytes.

```
rawTxBytes := ts.GetRlp(0)
```

Finally we can very easily turn the raw bytes into a hex string.

```
rawTxHex := hex.EncodeToString(rawTxBytes)

fmt.Printf(rawTxHex)
// f86d8202b38477359400825208944592d8f8d7b001e72cb26a73e4fa1806a
```

And now you have the raw transaction data which you can use to broadcast at a future date. In the [next section](#) we'll learn how to broadcast a raw transaction.

Full code

[transaction_raw_create.go](#)

```

package main

import (
    "context"
    "crypto/ecdsa"
    "encoding/hex"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad")
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(10000000000000000) // in wei (1 eth)
    gasLimit := uint64(21000)              // in units
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f")
    var data []byte

```



```
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, d

chainID, err := client.NetworkID(context.Background())
if err != nil {
    log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), private
if err != nil {
    log.Fatal(err)
}

ts := types.Transactions{signedTx}
rawTxBytes := ts.GetRlp(0)
rawTxHex := hex.EncodeToString(rawTxBytes)

fmt.Printf(rawTxHex) // f86...772
}
```

Send Raw Transaction

In the [previous section](#) we learned how to create a raw transaction. Now we'll learn how to broadcast it to the [Ethereum](#) network in order for it to get processed and mined.

First decode the raw transaction hex to bytes format.

```
rawTx := "f86d8202b28477359400825208944592d8f8d7b001e72cb26a73e4
rawTxBytes, err := hex.DecodeString(rawTx)
```

Now initialize a new `types.Transaction` pointer and call `DecodeBytes` from the [go-ethereum](#) `rlp` package passing it the raw transaction bytes and the pointer to the [ethereum](#) transaction type. [RLP](#) is an encoding method used by [Ethereum](#) to serialized and derialized data.

```
tx := new(types.Transaction)
rlp.DecodeBytes(rawTxBytes, &tx)
```

Now we can easily broadcast the transaction with our [ethereum](#) client.

```
err := client.SendTransaction(context.Background(), tx)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0xc429e5f128387d224ba8bed6885e86525e14bfdc2eb24b5e9c3351a1176fd81f
```

You can see the transaction on etherscan:

<https://rinkeby.etherscan.io/tx/0xc429e5f128387d224ba8bed6885e86525e14bfdc2eb24b5e9c3351a1176fd81f>

Full code

[transaction_raw_sendcreate.go](#)

```
package main

import (
    "context"
    "encoding/hex"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/ethclient"
    "github.com/ethereum/go-ethereum/rlp"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    rawTx := "f86d8202b28477359400825208944592d8f8d7b001e72cb26a73"

    rawTxBytes, err := hex.DecodeString(rawTx)

    tx := new(types.Transaction)
    rlp.DecodeBytes(rawTxBytes, &tx)

    err = client.SendTransaction(context.Background(), tx)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0xc429e5f128387d224k
}
```

Smart Contracts

In the next sections we'll learn how to compile, deploy, read, and write to [smart contract](#) using [Go](#).

Smart Contract Compilation & ABI

In order to interact with a [smart contract](#) in a [Go](#) application, we must first generate the ABI (application binary interface) of the contract and compile the ABI to a format that we can import into our [Go](#) application.

Install the Solidity Compiler

The first step is to install the [Solidity compiler](#) (`solc`). In this guide, we're using Solidity 0.4.24.

Solc is available as a snapcraft package for Ubuntu.

```
sudo snap install solc --edge
```

Solc is available as a Homebrew package for macOS.

```
brew update
brew tap ethereum/ethereum
brew install solidity
```

Alternatively, you can run `solc` with Docker using the following command:

```
## Download Docker image
docker pull ethereum/solc:0.4.24

## Usage: docker run --rm -v $(pwd):/root ethereum/solc:0.4.24 <command>
# To generate ABI file from .sol file and send output to ./build folder
docker run --rm -v $(pwd):/root ethereum/solc:0.4.24 --abi /root/Store.sol -o
```

For other platforms or for installing from source, check out the official solidity [install guide](#).

Install `abigen` tool

We also need to install a tool called `abigen` for generating the ABI from a solidity [smart contract](#).

Assuming you have [Go](#) all set up on your computer, simply run the following to install the `abigen` tool.

```
go get -u github.com/ethereum/go-ethereum
cd $GOPATH/src/github.com/ethereum/go-ethereum/
make
make devtools
```

Create smart contract

We'll create a simple [smart contract](#) to test with. More complex smart contracts, and [smart contract](#) development in general is out of scope for this book. I highly recommend checking out [truffle framework](#) for developing and testing smart contracts.

This simple contract will be a key/value store with only 1 external method to set a key/value pair by anyone. We also added an event to emit after the value is set.

```
pragma solidity ^0.4.24;

contract Store {
    event ItemSet(bytes32 key, bytes32 value);

    string public version;
    mapping (bytes32 => bytes32) public items;

    constructor(string _version) public {
        version = _version;
    }

    function setItem(bytes32 key, bytes32 value) external {
        items[key] = value;
        emit ItemSet(key, value);
    }
}
```

Although this [smart contract](#) is simple, it will work for this example.

Create Go contract file

Now we can generate the ABI from a solidity source file.

```
solc --abi Store.sol -o build
```

It'll write it to a file called `./build/Store.abi`

Now let's convert the ABI to a [Go](#) file that we can import. This new file will contain all the available methods the we can use to interact with the [smart contract](#) from our [Go](#) application.

```
abigen --abi=./build/Store.abi --pkg=store --out=Store.go
```

In order to deploy a [smart contract](#) from [Go](#), we also need to compile the solidity [smart contract](#) to [EVM](#) bytecode. The [EVM](#) bytecode is what will be sent in the data field of the transaction. The bin file is required for generating the deploy methods on the [Go](#) contract file.

```
solc --bin Store.sol -o build
```

Now we compile the [Go](#) contract file which will include the deploy methods because we includes the bin file.

```
abigen --bin=./build/Store.bin --abi=./build/Store.abi --pkg=store --out=Store.go
```

That's it for this lesson. In the next lessons we'll learn how to deploy the [smart contract](#), and then interact with it.

Full code

Commands

```
go get -u github.com/ethereum/go-ethereum
cd $GOPATH/src/github.com/ethereum/go-ethereum/
make
make devtools

solc --abi --bin Store.sol -o build
abigen --bin=./build/Store.bin --abi=./build/Store.abi --pkg=store --out=Store.go
```

[Store.sol](#)

```
pragma solidity ^0.4.24;

contract Store {
    event ItemSet(bytes32 key, bytes32 value);

    string public version;
    mapping (bytes32 => bytes32) public items;

    constructor(string _version) public {
        version = _version;
    }

    function setItem(bytes32 key, bytes32 value) external {
        items[key] = value;
        emit ItemSet(key, value);
    }
}
```


Deploying a Smart Contract

If you haven't already, check out the [section on smart contract compilation](#) since this lesson requires knowledge on compiling a solidity smart contract to a Go contract file.

Assuming you've imported the newly created Go package file generated from `abigen`, and set the `ethclient`, loaded your `private key`, the next step is to create a keyed transactor. First import the `accounts/abi/bind` package from `go-ethereum` and then invoke `NewKeyedTransactor` passing in the `private key`. Afterwards set the usual properties such as the `nonce`, `gas` price, `gas` limit, and ETH value.

```
auth := bind.NewKeyedTransactor(privateKey)
auth.Nonce = big.NewInt(int64(nonce))
auth.Value = big.NewInt(0) // in wei
auth.GasLimit = uint64(300000) // in units
auth.GasPrice = gasPrice
```

If you recall in the previous section, we created a very simple `Store` contract that sets and stores key/value pairs. The generated Go contract file provides a `deploy` method. The `deploy` method name always starts with the word *Deploy* followed by the contract name, in this case it's *Store*.

The `deploy` function takes in the keyed transactor, the `ethclient`, and any input arguments that the `smart contract` constructor might takes in. We've set our `smart contract` to take in a string argument for the version. This function will return the `Ethereum` address of the newly deployed contract, the transaction object, the contract instance so that we can start interacting with, and the error if any.

```
input := "1.0"
address, tx, instance, err := store.DeployStore(auth, client, input)
if err != nil {
    log.Fatal(err)
}

fmt.Println(address.Hex()) // 0x147B8eb97fD247D06C4006D269c90C1908F
fmt.Println(tx.Hash().Hex()) // 0xdae8ba5444eefdc99f4d45cd0c4f24056cba6

_ = instance // will be using the instance in the next section
```

Yes it's that simply. You can take the transaction hash and see the deployment status on Etherscan:

[https://rinkeby.etherscan.io/tx/0xdae8ba5444eefdc99f4d45cd0c4f24056c
ba6a02cefbf78066ef9f4188ff7dc0](https://rinkeby.etherscan.io/tx/0xdae8ba5444eefdc99f4d45cd0c4f24056cba6a02cefbf78066ef9f4188ff7dc0)

Full code

Commands

```
solc --abi Store.sol  
solc --bin Store.sol  
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out:
```

[Store.sol](#)

```
pragma solidity ^0.4.24;  
  
contract Store {  
    event ItemSet(bytes32 key, bytes32 value);  
  
    string public version;  
    mapping (bytes32 => bytes32) public items;  
  
    constructor(string _version) public {  
        version = _version;  
    }  
  
    function setItem(bytes32 key, bytes32 value) external {  
        items[key] = value;  
        emit ItemSet(key, value);  
    }  
}
```

[contract_deploy.go](#)

```

package main

import (
    "context"
    "crypto/ecdsa"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad")
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    auth := bind.NewKeyedTransactor(privateKey)
    auth.Nonce = big.NewInt(int64(nonce))
    auth.Value = big.NewInt(0) // in wei
    auth.GasLimit = uint64(300000) // in units

```

```
auth.GasPrice = gasPrice

input := "1.0"
address, tx, instance, err := store.DeployStore(auth, client, input)
if err != nil {
    log.Fatal(err)
}

fmt.Println(address.Hex()) // 0x147B8eb97fD247D06C4006D269c90C190
fmt.Println(tx.Hash().Hex()) // 0xdae8ba5444eefdc99f4d45cd0c4f24056cb,

_ = instance
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Loading a Smart Contract

This section requires knowledge of how to compile a [smart contract's](#) ABI to a [Go](#) contract file. If you haven't already gone through it, please [read the section](#) first.

Once you've compiled your [smart contract's](#) ABI to a [Go](#) package using the `abigen` tool, the next step is to call the "New" method, which is in the format `New<ContractName>`, so in our example if you recall it's going to be `NewStore`. This initializer method takes in the address of the [smart contract](#) and returns a contract instance that you can start interact with it.

```
address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c906")
instance, err := store.NewStore(address, client)
if err != nil {
    log.Fatal(err)
}

_ = instance // we'll be using this in the next section
```

Full code

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out=
```

[Store.sol](#)

```
pragma solidity ^0.4.24;

contract Store {
    event ItemSet(bytes32 key, bytes32 value);

    string public version;
    mapping (bytes32 => bytes32) public items;

    constructor(string _version) public {
        version = _version;
    }

    function setItem(bytes32 key, bytes32 value) external {
        items[key] = value;
        emit ItemSet(key, value);
    }
}
```

[contract_load.go](#)

```
package main

import (
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c9")
    instance, err := store.NewStore(address, client)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("contract is loaded")
    _ = instance
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Querying a Smart Contract

This section requires knowledge of how to compile a [smart contract's](#) ABI to a [Go](#) contract file. If you haven't already gone through it, please [read the section](#) first.

In the previous section we learned how to initialize a contract instance in our [Go](#) application. Now we're going to read the [smart contract](#) using the provided methods by the new contract instance. If you recall we had a global variable named `version` in our contract that was set during deployment. Because it's public that means that they'll be a getter function automatically created for us. Constant and view functions also accept `bind.CallOpts` as the first argument. To learn about what options you can pass checkout the type's [documentation](#) but usually this is set to `nil`.

```
version, err := instance.Version(nil)
if err != nil {
    log.Fatal(err)
}

fmt.Println(version) // "1.0"
```

Full code

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out=
```

[Store.sol](#)


```
pragma solidity ^0.4.24;

contract Store {
    event ItemSet(bytes32 key, bytes32 value);

    string public version;
    mapping (bytes32 => bytes32) public items;

    constructor(string _version) public {
        version = _version;
    }

    function setItem(bytes32 key, bytes32 value) external {
        items[key] = value;
        emit ItemSet(key, value);
    }
}
```

[contract_read.go](#)

```
package main

import (
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c9")
    instance, err := store.NewStore(address, client)
    if err != nil {
        log.Fatal(err)
    }

    version, err := instance.Version(nil)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(version) // "1.0"
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Writing to a Smart Contract

This section requires knowledge of how to compile a [smart contract's](#) ABI to a [Go](#) contract file. If you haven't already gone through it, please [read the section](#) first.

Writing to a [smart contract](#) requires us to sign the transaction with our [private key](#).

```
privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad0f:
if err != nil {
    log.Fatal(err)
}

publicKey := privateKey.Public()
publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
if !ok {
    log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
}

fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
```

We'll also need to figure the [nonce](#) and [gas](#) price.

```
nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
if err != nil {
    log.Fatal(err)
}

gasPrice, err := client.SuggestGasPrice(context.Background())
if err != nil {
    log.Fatal(err)
}
```

Next we create a new keyed transactor which takes in the [private key](#).

```
auth := bind.NewKeyedTransactor(privateKey)
```

Then we need to set the standard transaction options attached to the keyed transactor.

```
auth.Nonce = big.NewInt(int64(nonce))
auth.Value = big.NewInt(0) // in wei
auth.GasLimit = uint64(300000) // in units
auth.GasPrice = gasPrice
```

Now we load an instance of the [smart contract](#). If you recall in the [previous sections](#) we create a contract called *Store* and generated a [Go](#) package file using the `abigen` tool. To initialize it we just invoke the `New` method of the contract package and give the [smart contract](#) address and the `ethclient`, which returns a contract instance that we can use.

```
address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c906")
instance, err := store.NewStore(address, client)
if err != nil {
    log.Fatal(err)
}
```

The [smart contract](#) that we created has an external method called *SetItem* which takes in two arguments (key, value) in the form of solidity `bytes32`. This means that the [Go](#) contract package requires us to pass a byte array of length 32 bytes. Invoking the *SetItem* method requires us to pass the `auth` object we created earlier. Behind the scenes this method will encode this function call with its arguments, set it as the `data` property of the transaction, and sign it with the [private key](#). The result will be a signed transaction object.

```
key := [32]byte{}
value := [32]byte{}
copy(key[:], []byte("foo"))
copy(value[:], []byte("bar"))

tx, err := instance.SetItem(auth, key, value)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0x8d490e535678e9a24360e955d75b27ad307bdfb97a1dca51d0f3035dcee3e870
```

We can see now that the transaction has been successfully sent on the network:

<https://rinkeby.etherscan.io/tx/0x8d490e535678e9a24360e955d75b27ad307bdfb97a1dca51d0f3035dcee3e870>

To verify that the key/value was set, we read the [smart contract](#) mapping value.

```
result, err := instance.Items(nil, key)
if err != nil {
    log.Fatal(err)
}

fmt.Println(string(result[:])) // "bar"
```

There you have it.

Full code

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out=
```

[Store.sol](#)

```
pragma solidity ^0.4.24;

contract Store {
    event ItemSet(bytes32 key, bytes32 value);

    string public version;
    mapping (bytes32 => bytes32) public items;

    constructor(string _version) public {
        version = _version;
    }

    function setItem(bytes32 key, bytes32 value) external {
        items[key] = value;
        emit ItemSet(key, value);
    }
}
```

[contract_write.go](#)

```

package main

import (
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad9
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    auth := bind.NewKeyedTransactor(privateKey)
    auth.Nonce = big.NewInt(int64(nonce))
    auth.Value = big.NewInt(0) // in wei
    auth.GasLimit = uint64(300000) // in units
    auth.GasPrice = gasPrice

    address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c9

```

Setting up the Client

```
instance, err := store.NewStore(address, client)
if err != nil {
    log.Fatal(err)
}

key := [32]byte{}
value := [32]byte{}
copy(key[:], []byte("foo"))
copy(value[:], []byte("bar"))

tx, err := instance.SetItem(auth, key, value)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0x8d490e535678e9a24

result, err := instance.Items(nil, key)
if err != nil {
    log.Fatal(err)
}

fmt.Println(string(result[:])) // "bar"
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Reading Smart Contract Bytecode

Sometimes you'll need to read the bytecode of a deployed [smart contract](#). Since all the [smart contract](#) bytecode lives on the [blockchain](#), we can easily fetch it.

First set up the client and the [smart contract](#) address you want to read the bytecode of.

```
client, err := ethclient.Dial("https://rinkeby.infura.io")
if err != nil {
    log.Fatal(err)
}

contractAddress := common.HexToAddress("0x147B8eb97fD247D06C4006F")
```

Now all you have to is call the `codeAt` method of the client. The `codeAt` method accepts a [smart contract](#) address and an optional block number, and returns the bytecode in bytes format.

```
bytecode, err := client.CodeAt(context.Background(), contractAddress, nil) //
if err != nil {
    log.Fatal(err)
}

fmt.Println(hex.EncodeToString(bytecode)) // 60806...10029
```

See the same bytecode hex on etherscan

<https://rinkeby.etherscan.io/address/0x147b8eb97fd247d06c4006d269c90c1908fb5d54#code>

Full code

[contract_bytecode.go](#)


```
package main

import (
    "context"
    "encoding/hex"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    contractAddress := common.HexToAddress("0x147B8eb97fD247D06C400
    bytecode, err := client.CodeAt(context.Background(), contractAddress, nil)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(hex.EncodeToString(bytecode)) // 60806...10029
}
```

Querying an ERC20 Token Smart Contract

First create an ERC20 [smart contract](#) interface. This is just a contract with the function definitions of the functions that you can call.

```
pragma solidity ^0.4.24;

contract ERC20 {
    string public constant name = "";
    string public constant symbol = "";
    uint8 public constant decimals = 0;

    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint balance);
    function allowance(address tokenOwner, address spender) public constant returns (uint remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns (bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}
```

Then compile the [smart contract](#) to the JSON ABI, and create a [Go token](#) package out of the ABI using `abigen`.

```
solc --abi erc20.sol
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

Assuming we already have [Ethereum](#) client set up as usual, we can now import the new [token](#) package into our application and instantiate it. In this example we'll be using the [Golem token](#).

```
tokenAddress := common.HexToAddress("0xa74476443119A942dE498590F...")
instance, err := token.NewToken(tokenAddress, client)
if err != nil {
    log.Fatal(err)
}
```

We may now call any ERC20 method that we like. For example, we can query the [token](#) balance of a user.

```
address := common.HexToAddress("0x0536806df512d6cdde913cf95c9886f65b1d3462")
bal, err := instance.BalanceOf(&bind.CallOpts{}, address)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("wei: %s\n", bal) // "wei: 74605500647408739782407023"
```

We can also read the public variables of the ERC20 [smart contract](#).

```
name, err := instance.Name(&bind.CallOpts{})
if err != nil {
    log.Fatal(err)
}

symbol, err := instance.Symbol(&bind.CallOpts{})
if err != nil {
    log.Fatal(err)
}

decimals, err := instance.Decimals(&bind.CallOpts{})
if err != nil {
    log.Fatal(err)
}

fmt.Printf("name: %s\n", name)      // "name: Golem Network"
fmt.Printf("symbol: %s\n", symbol)  // "symbol: GNT"
fmt.Printf("decimals: %v\n", decimals) // "decimals: 18"
```

We can do some simple math to convert the balance into a human readable decimal format.

```
fbal := new(big.Float)
fbal.SetString(bal.String())
value := new(big.Float).Quo(fbal, big.NewFloat(math.Pow10(int(decimals))))

fmt.Printf("balance: %f", value) // "balance: 74605500.647409"
```

See the same information on etherscan:

<https://etherscan.io/token/0xa74476443119a942de498590fe1f2454d7d4ac0d?a=0x0536806df512d6cdde913cf95c9886f65b1d3462>

Full code

Commands

```
solc --abi erc20.sol  
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

[erc20.sol](#)

```
pragma solidity ^0.4.24;  
  
contract ERC20 {  
    string public constant name = "";  
    string public constant symbol = "";  
    uint8 public constant decimals = 0;  
  
    function totalSupply() public constant returns (uint);  
    function balanceOf(address tokenOwner) public constant returns (uint balance);  
    function allowance(address tokenOwner, address spender) public constant  
        returns (uint remaining);  
    function transfer(address to, uint tokens) public returns (bool success);  
    function approve(address spender, uint tokens) public returns (bool success);  
    function transferFrom(address from, address to, uint tokens) public returns (bool success);  
  
    event Transfer(address indexed from, address indexed to, uint tokens);  
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);  
}
```

[contract_read_erc20.go](#)

```

package main

import (
    "fmt"
    "log"
    "math"
    "math/big"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"

    token "./contracts_erc20" // for demo
)

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // Golem (GNT) Address
    tokenAddress := common.HexToAddress("0xa74476443119A942dE498590")
    instance, err := token.NewToken(tokenAddress, client)
    if err != nil {
        log.Fatal(err)
    }

    address := common.HexToAddress("0x0536806df512d6cdde913cf95c988")
    bal, err := instance.BalanceOf(&bind.CallOpts{}, address)
    if err != nil {
        log.Fatal(err)
    }

    name, err := instance.Name(&bind.CallOpts{})
    if err != nil {
        log.Fatal(err)
    }

    symbol, err := instance.Symbol(&bind.CallOpts{})
    if err != nil {
        log.Fatal(err)
    }

    decimals, err := instance.Decimals(&bind.CallOpts{})
    if err != nil {
        log.Fatal(err)
    }
}

```

Setting up the Client

```
fmt.Printf("name: %s\n", name)    // "name: Golem Network"
fmt.Printf("symbol: %s\n", symbol) // "symbol: GNT"
fmt.Printf("decimals: %v\n", decimals) // "decimals: 18"

fmt.Printf("wei: %s\n", bal) // "wei: 74605500647408739782407023"

fbal := new(big.Float)
fbal.SetString(bal.String())
value := new(big.Float).Quo(fbal, big.NewFloat(math.Pow10(int(decimals)))

fmt.Printf("balance: %f", value) // "balance: 74605500.647409"
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Events

Smart contracts have the ability to "emit" events during execution. Events are also known as "logs" in [Ethereum](#). The output of the events are stored in transaction receipts under a logs section. Events have become pretty widely used in [Ethereum](#) smart contracts to log when a significant action has occurred, particularly in [token](#) contracts (i.e. [ERC-20](#)) to indicate that a [token](#) transfer has occurred. These sections will walk you through the process of reading events from the [blockchain](#) as well as subscribing to events so that you get notified in real time as the transaction gets mined.

Subscribing to Event Logs

First thing we need to do in order to subscribe to event logs is dial to a websocket enabled [Ethereum](#) client. Fortunately for us, [Infura](#) supports websockets.

```
client, err := ethclient.Dial("wss://rinkeby.infura.io/ws")
if err != nil {
    log.Fatal(err)
}
```

The next step is to create a filter query. In this example we'll be reading all events coming from the example contract that we've created in the previous lessons.

```
contractAddress := common.HexToAddress("0x147B8eb97fD247D06C4006F...")
query := ethereum.FilterQuery{
    Addresses: []common.Address{contractAddress},
}
```

The way we'll be receiving events is through a [Go](#) channel. Let's create one with type of `Log` from the `go-ethereum` `core/types` package.

```
logs := make(chan types.Log)
```

Now all we have to do is subscribe by calling `SubscribeFilterLogs` from the client, which takes in the query options and the output channel. This will return a subscription struct containing `unsubscribe` and `error` methods.

```
sub, err := client.SubscribeFilterLogs(context.Background(), query, logs)
if err != nil {
    log.Fatal(err)
}
```

Finally all we have to do is setup an continous loop with a select statement to read in either new log events or the subscription error.


```
for {
    select {
        case err := <-sub.Err():
            log.Fatal(err)
        case vLog := <-logs:
            fmt.Println(vLog) // pointer to event log
    }
}
```

You'll have to parse the log entries, which we'll learn how to do in the [next section](#).

Full code

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out:
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
    event ItemSet(bytes32 key, bytes32 value);

    string public version;
    mapping (bytes32 => bytes32) public items;

    constructor(string _version) public {
        version = _version;
    }

    function setItem(bytes32 key, bytes32 value) external {
        items[key] = value;
        emit ItemSet(key, value);
    }
}
```

[event_subscribe.go](#)

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("wss://rinkeby.infura.io/ws")
    if err != nil {
        log.Fatal(err)
    }

    contractAddress := common.HexToAddress("0x147B8eb97fD247D06C400
    query := ethereum.FilterQuery{
        Addresses: []common.Address{contractAddress},
    }

    logs := make(chan types.Log)
    sub, err := client.SubscribeFilterLogs(context.Background(), query, logs)
    if err != nil {
        log.Fatal(err)
    }

    for {
        select {
        case err := <-sub.Err():
            log.Fatal(err)
        case vLog := <-logs:
            fmt.Println(vLog) // pointer to event log
        }
    }
}
```

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Reading Event Logs

A [smart contract](#) may optionally emit "events" which get stored as logs as part of the transaction receipt. Reading these events are pretty simple. First we need to construct a filter query. We import the `FilterQuery` struct from the [go-ethereum](#) package and initialize it with filter options. We tell it the range of blocks that we want to filter through and specify the contract address to read this logs from. In this example we'll be reading all the logs from a particular block, from the [smart contract](#) we created in the [smart contract sections](#).

```
query := ethereum.FilterQuery{
    FromBlock: big.NewInt(2394201),
    ToBlock:   big.NewInt(2394201),
    Addresses: []common.Address{
        contractAddress,
    },
}
```

The next step is to call `FilterLogs` from the `ethclient` that takes in our query and will return all the matching event logs.

```
logs, err := client.FilterLogs(context.Background(), query)
if err != nil {
    log.Fatal(err)
}
```

All the logs returned will be ABI encoded so by themselves they won't be very readable. In order to decode the logs we'll need to import our [smart contract ABI](#). To do that, we import our compiled [smart contract Go](#) package which will contain an external property in the name format `<ContractName>ABI` containing our ABI. Afterwards we use the `abi.JSON` function from the [go-ethereum](#) `accounts/abi` [go-ethereum](#) package to return a parsed ABI interface that we can use in our [Go](#) application.

```
contractAbi, err := abi.JSON(strings.NewReader(string(store.StoreABI)))
if err != nil {
    log.Fatal(err)
}
```

Now we can iterate through the logs and decode them into a type we can use. If you recall the logs that our sample contract emitted were of type `bytes32` in Solidity, so the equivalent in Go would be `[32]byte`. We can create an anonymous struct with these types and pass a pointer as the first argument to the `Unpack` function of the parsed ABI interface to decode the raw log data. The second argument is the name of the event we're trying to decode and the last argument is the encoded log data.

```
for _, vLog := range logs {
    event := struct {
        Key [32]byte
        Value [32]byte
    }{}
    err := contractAbi.Unpack(&event, "ItemSet", vLog.Data)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(string(event.Key[:])) // foo
    fmt.Println(string(event.Value[:])) // bar
}
```

Also, the log struct contains additional information such as the block hash, block number, and transaction hash.

```
fmt.Println(vLog.BlockHash.Hex()) // 0x3404b8c050aa0aacd0223e91b5c32fe
fmt.Println(vLog.BlockNumber)    // 2394201
fmt.Println(vLog.TxHash.Hex())   // 0x280201eda63c9ff6f305fcee51d5eb861
```

Topics

If your solidity event contains `indexed` event types, then they become a *topic* rather than part of the data property of the log. In solidity you may only have up to 4 topics but only 3 indexed event types. The first topic is *always* the `signature` of the event. Our example contract didn't contain indexed events, but if it did this is how to read the event topics.

```
var topics [4]string
for i := range vLog.Topics {
    topics[i] = vLog.Topics[i].Hex()
}

fmt.Println(topics[0]) // 0xe79e73da417710ae99aa2088575580a60415d359a
```

As you can see here the first topic is just the hashed event [signature](#).

```
eventSignature := []byte("ItemSet(bytes32,bytes32)")
hash := crypto.Keccak256Hash(eventSignature)
fmt.Println(hash.Hex()) // 0xe79e73da417710ae99aa2088575580a60415d35
```

That's all there is to reading and parsing logs. To learn how to subscribe to logs, read the [previous section](#).

Full code

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out:
```

[Store.sol](#)

```
pragma solidity ^0.4.24;

contract Store {
    event ItemSet(bytes32 key, bytes32 value);

    string public version;
    mapping (bytes32 => bytes32) public items;

    constructor(string _version) public {
        version = _version;
    }

    function setItem(bytes32 key, bytes32 value) external {
        items[key] = value;
        emit ItemSet(key, value);
    }
}
```

[event_read.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"
    "math/big"
    "strings"

    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/accounts/abi"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("wss://rinkeby.infura.io/ws")
    if err != nil {
        log.Fatal(err)
    }

    contractAddress := common.HexToAddress("0x147B8eb97fD247D06C400
    query := ethereum.FilterQuery{
        FromBlock: big.NewInt(2394201),
        ToBlock:   big.NewInt(2394201),
        Addresses: []common.Address{
            contractAddress,
        },
    }

    logs, err := client.FilterLogs(context.Background(), query)
    if err != nil {
        log.Fatal(err)
    }

    contractAbi, err := abi.JSON(strings.NewReader(string(store.StoreABI)))
    if err != nil {
        log.Fatal(err)
    }

    for _, vLog := range logs {
        fmt.Println(vLog.BlockHash.Hex()) // 0x3404b8c050aa0aacd0223e91b5c
        fmt.Println(vLog.BlockNumber)     // 2394201
        fmt.Println(vLog.TxHash.Hex())    // 0x280201eda63c9ff6f305fcee51d5e
    }
}

```

```
event := struct {
    Key  [32]byte
    Value [32]byte
}{}
err := contractAbi.Unpack(&event, "ItemSet", vLog.Data)
if err != nil {
    log.Fatal(err)
}

fmt.Println(string(event.Key[:])) // foo
fmt.Println(string(event.Value[:])) // bar

var topics [4]string
for i := range vLog.Topics {
    topics[i] = vLog.Topics[i].Hex()
}

fmt.Println(topics[0]) // 0xe79e73da417710ae99aa2088575580a60415d
}

eventSignature := []byte("ItemSet(bytes32,bytes32)")
hash := crypto.Keccak256Hash(eventSignature)
fmt.Println(hash.Hex()) // 0xe79e73da417710ae99aa2088575580a60415d
}
```

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Reading ERC-20 Token Event Logs

First create the [ERC-20 smart contract](#) interface for event logs as `erc20.sol` :

```
pragma solidity ^0.4.24;

contract ERC20 {
    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, u
}
```

Then use `abigen` to create the [Go](#) `exchange` package given the abi:

```
solc --abi erc20.sol
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

Now in our [Go](#) application let's create the struct types matching the types of the [ERC-20 event log signature](#):

```
type LogTransfer struct {
    From common.Address
    To common.Address
    Tokens *big.Int
}

type LogApproval struct {
    TokenOwner common.Address
    Spender common.Address
    Tokens *big.Int
}
```

Initialize the [ethereum](#) client:

```
client, err := ethclient.Dial("https://mainnet.infura.io")
if err != nil {
    log.Fatal(err)
}
```

Create a `FilterQuery` passing the [ERC-20 smart contract](#) address and the desired block range. We'll be using the [ZRX token](#) for this example:


```
// 0x Protocol (ZRX) token address
contractAddress := common.HexToAddress("0xe41d2489571d322189246da
query := ethereum.FilterQuery{
    FromBlock: big.NewInt(6383820),
    ToBlock:   big.NewInt(6383840),
    Addresses: []common.Address{
        contractAddress,
    },
}
```

Query the logs with `FilterLogs` :

```
logs, err := client.FilterLogs(context.Background(), query)
if err != nil {
    log.Fatal(err)
}
```

Next we'll parse the JSON abi which we'll use to unpack the raw log data later:

```
contractAbi, err := abi.JSON(strings.NewReader(string(token.TokenABI)))
if err != nil {

    log.Fatal(err)
}
```

In order to filter by certain log type, we need to figure out the keccak256 hash of each event log function [signature](#). The event log function [signature](#) hash is always `topic[0]` as we'll see soon. Here's how to calculate the keccak256 hash using the [go-ethereum](#) `crypto` package:

```
logTransferSig := []byte("Transfer(address,address,uint256)")
LogApprovalSig := []byte("Approval(address,address,uint256)")
logTransferSigHash := crypto.Keccak256Hash(logTransferSig)
logApprovalSigHash := crypto.Keccak256Hash(LogApprovalSig)
```

Now we'll iterate through all the logs and set up a switch statement to filter by event log type:

```

for _, vLog := range logs {
    fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
    fmt.Printf("Log Index: %d\n", vLog.Index)

    switch vLog.Topics[0].Hex() {
    case logTransferSigHash.Hex():
        //
    case logApprovalSigHash.Hex():
        //
    }
}

```

Now to parse the `Transfer` event log we'll use `abi.Unpack` to parse the raw log data into our log type struct. `Unpack` will not parse indexed event types because those are stored under `topics`, so for those we'll have to parse separately as seen in the example below:

```

fmt.Printf("Log Name: Transfer\n")

var transferEvent LogTransfer

err := contractAbi.Unpack(&transferEvent, "Transfer", vLog.Data)
if err != nil {
    log.Fatal(err)
}

transferEvent.From = common.HexToAddress(vLog.Topics[1].Hex())
transferEvent.To = common.HexToAddress(vLog.Topics[2].Hex())

fmt.Printf("From: %s\n", transferEvent.From.Hex())
fmt.Printf("To: %s\n", transferEvent.To.Hex())
fmt.Printf("Tokens: %s\n", transferEvent.Tokens.String())

```

Similarly for the `Approval` event log:

```
fmt.Printf("Log Name: Approval\n")

var approvalEvent LogApproval

err := contractAbi.Unpack(&approvalEvent, "Approval", vLog.Data)
if err != nil {
    log.Fatal(err)
}

approvalEvent.TokenOwner = common.HexToAddress(vLog.Topics[1].Hex())
approvalEvent.Spender = common.HexToAddress(vLog.Topics[2].Hex())

fmt.Printf("Token Owner: %s\n", approvalEvent.TokenOwner.Hex())
fmt.Printf("Spender: %s\n", approvalEvent.Spender.Hex())
fmt.Printf("Tokens: %s\n", approvalEvent.Tokens.String())
```

Putting it all together and running it we'll see the following output:

[illegible]

Compare the parsed log output to what's on etherscan:

<https://etherscan.io/tx/0x0c3b6cf604275c7e44dc7db400428c1a39f33f0c6cbc19ff625f6057a5cb32c0#eventlog>

Full code

Commands

```
solc --abi erc20.sol  
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

[erc20.sol](#)

```
pragma solidity ^0.4.24;  
  
contract ERC20 {  
    event Transfer(address indexed from, address indexed to, uint tokens);  
    event Approval(address indexed tokenOwner, address indexed spender, u  
}  
}
```

[event_read_erc20.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"
    "math/big"
    "strings"

    token "./contracts_erc20" // for demo
    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/accounts/abi"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"
)

// LogTransfer ..
type LogTransfer struct {
    From common.Address
    To common.Address
    Tokens *big.Int
}

// LogApproval ..
type LogApproval struct {
    TokenOwner common.Address
    Spender common.Address
    Tokens *big.Int
}

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // 0x Protocol (ZRX) token address
    contractAddress := common.HexToAddress("0xe41d2489571d322189246")
    query := ethereum.FilterQuery{
        FromBlock: big.NewInt(6383820),
        ToBlock: big.NewInt(6383840),
        Addresses: []common.Address{
            contractAddress,
        },
    }

    logs, err := client.FilterLogs(context.Background(), query)

```

```

if err != nil {
    log.Fatal(err)
}

contractAbi, err := abi.JSON(strings.NewReader(string(token.TokenABI)))
if err != nil {
    log.Fatal(err)
}

logTransferSig := []byte("Transfer(address,address,uint256)")
logApprovalSig := []byte("Approval(address,address,uint256)")
logTransferSigHash := crypto.Keccak256Hash(logTransferSig)
logApprovalSigHash := crypto.Keccak256Hash(logApprovalSig)

for _, vLog := range logs {
    fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
    fmt.Printf("Log Index: %d\n", vLog.Index)

    switch vLog.Topics[0].Hex() {
    case logTransferSigHash.Hex():
        fmt.Printf("Log Name: Transfer\n")

        var transferEvent LogTransfer

        err := contractAbi.Unpack(&transferEvent, "Transfer", vLog.Data)
        if err != nil {
            log.Fatal(err)
        }

        transferEvent.From = common.HexToAddress(vLog.Topics[1].Hex())
        transferEvent.To = common.HexToAddress(vLog.Topics[2].Hex())

        fmt.Printf("From: %s\n", transferEvent.From.Hex())
        fmt.Printf("To: %s\n", transferEvent.To.Hex())
        fmt.Printf("Tokens: %s\n", transferEvent.Tokens.String())

    case logApprovalSigHash.Hex():
        fmt.Printf("Log Name: Approval\n")

        var approvalEvent LogApproval

        err := contractAbi.Unpack(&approvalEvent, "Approval", vLog.Data)
        if err != nil {
            log.Fatal(err)
        }

        approvalEvent.TokenOwner = common.HexToAddress(vLog.Topics[1].Hex())
        approvalEvent.Spender = common.HexToAddress(vLog.Topics[2].Hex())
    }
}

```

Setting up the Client

```
        fmt.Printf("Token Owner: %s\n", approvalEvent.TokenOwner.Hex())
        fmt.Printf("Spender: %s\n", approvalEvent.Spender.Hex())
        fmt.Printf("Tokens: %s\n", approvalEvent.Tokens.String())
    }

    fmt.Printf("\n\n")
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Reading 0x Protocol Event Logs

To read [0x Protocol](#) event logs we must first compile the solidity [smart contract](#) to a [Go](#) package.

Install solc version `0.4.11`

```
npm i -g solc@0.4.11
```

Create the 0x protocol exchange [smart contract](#) interface for event logs as `Exchange.sol` :

```
pragma solidity 0.4.11;

contract Exchange {
    event LogFill(
        address indexed maker,
        address taker,
        address indexed feeRecipient,
        address makerToken,
        address takerToken,
        uint filledMakerTokenAmount,
        uint filledTakerTokenAmount,
        uint paidMakerFee,
        uint paidTakerFee,
        bytes32 indexed tokens, // keccak256(makerToken, takerToken), allows
        bytes32 orderHash
    );

    event LogCancel(
        address indexed maker,
        address indexed feeRecipient,
        address makerToken,
        address takerToken,
        uint cancelledMakerTokenAmount,
        uint cancelledTakerTokenAmount,
        bytes32 indexed tokens,
        bytes32 orderHash
    );

    event LogError(uint8 indexed errorId, bytes32 indexed orderHash);
}
```

Then use `abigen` to create the [Go](#) `exchange` package given the abi:


```
solc --abi Exchange.sol  
abigen --abi="Exchange.sol:Exchange.abi" --pkg=exchange --out=Exchange.
```

Now in our [Go](#) application let's create the struct types matching the types of the 0xProtocol event log [signature](#):

```
type LogFill struct {  
    Maker          common.Address  
    Taker          common.Address  
    FeeRecipient   common.Address  
    MakerToken     common.Address  
    TakerToken     common.Address  
    FilledMakerTokenAmount *big.Int  
    FilledTakerTokenAmount *big.Int  
    PaidMakerFee    *big.Int  
    PaidTakerFee    *big.Int  
    Tokens         [32]byte  
    OrderHash      [32]byte  
}  
  
type LogCancel struct {  
    Maker          common.Address  
    FeeRecipient   common.Address  
    MakerToken     common.Address  
    TakerToken     common.Address  
    CancelledMakerTokenAmount *big.Int  
    CancelledTakerTokenAmount *big.Int  
    Tokens         [32]byte  
    OrderHash      [32]byte  
}  
  
type LogError struct {  
    ErrorID  uint8  
    OrderHash [32]byte  
}
```

Initialize the [ethereum](#) client:

```
client, err := ethclient.Dial("https://mainnet.infura.io")  
if err != nil {  
    log.Fatal(err)  
}
```

Create a `FilterQuery` passing the 0x Protocol [smart contract](#) address and the desired block range:

```
// 0x Protocol Exchange smart contract address
contractAddress := common.HexToAddress("0x12459C951127e0c374FF9109
query := ethereum.FilterQuery{
    FromBlock: big.NewInt(6383482),
    ToBlock:   big.NewInt(6383488),
    Addresses: []common.Address{
        contractAddress,
    },
}
```

Query the logs with `FilterLogs` :

```
logs, err := client.FilterLogs(context.Background(), query)
if err != nil {
    log.Fatal(err)
}
```

Next we'll parse the JSON abi which we'll use to unpack the raw log data later:

```
contractAbi, err := abi.JSON(strings.NewReader(string(exchange.ExchangeA
if err != nil {
    log.Fatal(err)
}
```

In order to filter by certain log type, we need to figure out the keccak256 hash of each event log function [signature](#). The event log function [signature](#) hash is always `topic[0]` as we'll see soon:

```
// NOTE: keccak256("LogFill(address,address,address,address,address,uint2
logFillEvent := common.HexToHash("0d0b9391970d9a25552f37d436d2aae2

// NOTE: keccak256("LogCancel(address,address,address,address,uint256,u
logCancelEvent := common.HexToHash("67d66f160bc93d925d05dae1794c9

// NOTE: keccak256("LogError(uint8,bytes32)")
logErrorEvent := common.HexToHash("36d86c59e00bd73dc19ba3adfe068e
```

Now we'll iterate through all the logs and set up a switch statement to filter by event log type:

```

for _, vLog := range logs {
    fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
    fmt.Printf("Log Index: %d\n", vLog.Index)

    switch vLog.Topics[0].Hex() {
    case logFillEvent.Hex():
        //
    case logCancelEvent.Hex():
        //
    case logErrorEvent.Hex():
        //
    }
}

```

Now to parse `LogFill` we'll use `abi.Unpack` to parse the raw log data into our log type struct. `Unpack` will not parse indexed event types because those are stored under `topics`, so for those we'll have to parse separately as seen in the example below:

```

fmt.Printf("Log Name: LogFill\n")

var fillEvent LogFill

err := contractAbi.Unpack(&fillEvent, "LogFill", vLog.Data)
if err != nil {
    log.Fatal(err)
}

fillEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
fillEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].Hex())
fillEvent.Tokens = vLog.Topics[3]

fmt.Printf("Maker: %s\n", fillEvent.Maker.Hex())
fmt.Printf("Taker: %s\n", fillEvent.Taker.Hex())
fmt.Printf("Fee Recipient: %s\n", fillEvent.FeeRecipient.Hex())
fmt.Printf("Maker Token: %s\n", fillEvent.MakerToken.Hex())
fmt.Printf("Taker Token: %s\n", fillEvent.TakerToken.Hex())
fmt.Printf("Filled Maker Token Amount: %s\n", fillEvent.FilledMakerTokenAn
fmt.Printf("Filled Taker Token Amount: %s\n", fillEvent.FilledTakerTokenAmc
fmt.Printf("Paid Maker Fee: %s\n", fillEvent.PaidMakerFee.String())
fmt.Printf("Paid Taker Fee: %s\n", fillEvent.PaidTakerFee.String())
fmt.Printf("Tokens: %s\n", hexutil.Encode(fillEvent.Tokens[:]))
fmt.Printf("Order Hash: %s\n", hexutil.Encode(fillEvent.OrderHash[:]))

```

Similarly for `LogCancel` :

```

fmt.Printf("Log Name: LogCancel\n")

var cancelEvent LogCancel

err := contractAbi.Unpack(&cancelEvent, "LogCancel", vLog.Data)
if err != nil {
    log.Fatal(err)
}

cancelEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
cancelEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].Hex())
cancelEvent.Tokens = vLog.Topics[3]

fmt.Printf("Maker: %s\n", cancelEvent.Maker.Hex())
fmt.Printf("Fee Recipient: %s\n", cancelEvent.FeeRecipient.Hex())
fmt.Printf("Maker Token: %s\n", cancelEvent.MakerToken.Hex())
fmt.Printf("Taker Token: %s\n", cancelEvent.TakerToken.Hex())
fmt.Printf("Cancelled Maker Token Amount: %s\n", cancelEvent.CancelledM
fmt.Printf("Cancelled Taker Token Amount: %s\n", cancelEvent.CancelledTak
fmt.Printf("Tokens: %s\n", hexutil.Encode(cancelEvent.Tokens[:]))
fmt.Printf("Order Hash: %s\n", hexutil.Encode(cancelEvent.OrderHash[:]))

```

And finally for `LogError` :

```

fmt.Printf("Log Name: LogError\n")

errorID, err := strconv.ParseInt(vLog.Topics[1].Hex(), 16, 64)
if err != nil {
    log.Fatal(err)
}

errorEvent := &LogError{
    ErrorID: uint8(errorID),
    OrderHash: vLog.Topics[2],
}

fmt.Printf("Error ID: %d\n", errorEvent.ErrorID)
fmt.Printf("Order Hash: %s\n", hexutil.Encode(errorEvent.OrderHash[:]))

```

Putting it all together and running it we'll see the following output:

Log Block Number: 6383482
Log Index: 35
Log Name: LogFill
Maker: 0x8dd688660ec0BaBD0B8a2f2DE3232645F73cC5eb
Taker: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Fee Recipient: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Maker Token: 0xD7732e3783b0047aa251928960063f863AD022D8
Taker Token: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
Filled Maker Token Amount: 2400000000000000000000
Filled Taker Token Amount: 6930282000000000000
Paid Maker Fee: 0
Paid Taker Fee: 0
Tokens: 0xf08499c9e419ea8c08c4b991f88632593fb36baf4124c62758acb21
Order Hash: 0x306a9a7ecbd9446559a2c650b4cfc16d1fb615aa2b3f4f63078

Log Block Number: 6383482
Log Index: 38
Log Name: LogFill
Maker: 0x04aa059b2e31B5898fAB5aB24761e67E8a196AB8
Taker: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Fee Recipient: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Maker Token: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
Taker Token: 0xD7732e3783b0047aa251928960063f863AD022D8
Filled Maker Token Amount: 6941718000000000000
Filled Taker Token Amount: 2400000000000000000000
Paid Maker Fee: 0
Paid Taker Fee: 0
Tokens: 0x97ef123f2b566f36ab1e6f5d462a8079fbe34fa667b4eae67194b3f9
Order Hash: 0xac270e88ce27b6bb78ee5b68ebaef666a77195020a6ab89228

Log Block Number: 6383488
Log Index: 43
Log Name: LogCancel
Maker: 0x0004E79C978B95974dCa16F56B516bE0c50CC652
Fee Recipient: 0xA258b39954ceF5cB142fd567A46cDdB31a670124
Maker Token: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
Taker Token: 0x89d24A6b4CcB1B6fAA2625fE562bDD9a23260359
Cancelled Maker Token Amount: 3000000000000000000
Cancelled Taker Token Amount: 7274848425000000000000
Tokens: 0x9dd48110dcc444fdc242510c09bbbbe21a5975cac061d82f7b843b
Order Hash: 0xe43eff38dc27af046bfbd431926926c072bbc7a509d56f6f1a7a

Compare the parsed log output to what's on etherscan:

<https://etherscan.io/tx/0xb73a4492c5db1f67930b25ce3869c1e6b9bdbccb239a23b6454925a5bc0e03c5>

Full code

Commands

```
solc --abi Exchange.sol  
abigen --abi="Exchange.sol:Exchange.abi" --pkg=exchange --out=Exchange.
```

Exchange.sol

```
pragma solidity 0.4.11;  
  
contract Exchange {  
    event LogFill(  
        address indexed maker,  
        address taker,  
        address indexed feeRecipient,  
        address makerToken,  
        address takerToken,  
        uint filledMakerTokenAmount,  
        uint filledTakerTokenAmount,  
        uint paidMakerFee,  
        uint paidTakerFee,  
        bytes32 indexed tokens, // keccak256(makerToken, takerToken), allows  
        bytes32 orderHash  
    );  
  
    event LogCancel(  
        address indexed maker,  
        address indexed feeRecipient,  
        address makerToken,  
        address takerToken,  
        uint cancelledMakerTokenAmount,  
        uint cancelledTakerTokenAmount,  
        bytes32 indexed tokens,  
        bytes32 orderHash  
    );  
  
    event LogError(uint8 indexed errorId, bytes32 indexed orderHash);  
}
```

Setting up the Client

[event_read_0xprotocol.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"
    "math/big"
    "strconv"
    "strings"

    exchange "./contracts_0xprotocol" // for demo
    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/accounts/abi"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/ethclient"
)

// LogFill ...
type LogFill struct {
    Maker          common.Address
    Taker          common.Address
    FeeRecipient    common.Address
    MakerToken      common.Address
    TakerToken      common.Address
    FilledMakerTokenAmount *big.Int
    FilledTakerTokenAmount *big.Int
    PaidMakerFee    *big.Int
    PaidTakerFee    *big.Int
    Tokens          [32]byte
    OrderHash       [32]byte
}

// LogCancel ...
type LogCancel struct {
    Maker          common.Address
    FeeRecipient    common.Address
    MakerToken      common.Address
    TakerToken      common.Address
    CancelledMakerTokenAmount *big.Int
    CancelledTakerTokenAmount *big.Int
    Tokens          [32]byte
    OrderHash       [32]byte
}

// LogError ...
type LogError struct {
    ErrorID uint8
}

```



```

    OrderHash [32]byte
}

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // 0x Protocol Exchange smart contract address
    contractAddress := common.HexToAddress("0x12459C951127e0c374FF91
    query := ethereum.FilterQuery{
        FromBlock: big.NewInt(6383482),
        ToBlock: big.NewInt(6383488),
        Addresses: []common.Address{
            contractAddress,
        },
    }

    logs, err := client.FilterLogs(context.Background(), query)
    if err != nil {
        log.Fatal(err)
    }

    contractAbi, err := abi.JSON(strings.NewReader(string(exchange.Exchange
    if err != nil {
        log.Fatal(err)
    }

    // NOTE: keccak256("LogFill(address,address,address,address,address,uint
    logFillEvent := common.HexToHash("0d0b9391970d9a25552f37d436d2aa
    // NOTE: keccak256("LogCancel(address,address,address,address,uint256
    logCancelEvent := common.HexToHash("67d66f160bc93d925d05dae1794
    // NOTE: keccak256("LogError(uint8,bytes32)")
    logErrorEvent := common.HexToHash("36d86c59e00bd73dc19ba3adfe06

    for _, vLog := range logs {
        fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
        fmt.Printf("Log Index: %d\n", vLog.Index)

        switch vLog.Topics[0].Hex() {
        case logFillEvent.Hex():
            fmt.Printf("Log Name: LogFill\n")

            var fillEvent LogFill

```

```

err := contractAbi.Unpack(&fillEvent, "LogFill", vLog.Data)
if err != nil {
    log.Fatal(err)
}

fillEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
fillEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].Hex())
fillEvent.Tokens = vLog.Topics[3]

fmt.Printf("Maker: %s\n", fillEvent.Maker.Hex())
fmt.Printf("Taker: %s\n", fillEvent.Taker.Hex())
fmt.Printf("Fee Recipient: %s\n", fillEvent.FeeRecipient.Hex())
fmt.Printf("Maker Token: %s\n", fillEvent.MakerToken.Hex())
fmt.Printf("Taker Token: %s\n", fillEvent.TakerToken.Hex())
fmt.Printf("Filled Maker Token Amount: %s\n", fillEvent.FilledMakerTokenAmount.Hex())
fmt.Printf("Filled Taker Token Amount: %s\n", fillEvent.FilledTakerTokenAmount.Hex())
fmt.Printf("Paid Maker Fee: %s\n", fillEvent.PaidMakerFee.String())
fmt.Printf("Paid Taker Fee: %s\n", fillEvent.PaidTakerFee.String())
fmt.Printf("Tokens: %s\n", hexutil.Encode(fillEvent.Tokens[:]))
fmt.Printf("Order Hash: %s\n", hexutil.Encode(fillEvent.OrderHash[:]))

case logCancelEvent.Hex():
    fmt.Printf("Log Name: LogCancel\n")

    var cancelEvent LogCancel

    err := contractAbi.Unpack(&cancelEvent, "LogCancel", vLog.Data)
    if err != nil {
        log.Fatal(err)
    }

    cancelEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
    cancelEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].Hex())
    cancelEvent.Tokens = vLog.Topics[3]

    fmt.Printf("Maker: %s\n", cancelEvent.Maker.Hex())
    fmt.Printf("Fee Recipient: %s\n", cancelEvent.FeeRecipient.Hex())
    fmt.Printf("Maker Token: %s\n", cancelEvent.MakerToken.Hex())
    fmt.Printf("Taker Token: %s\n", cancelEvent.TakerToken.Hex())
    fmt.Printf("Cancelled Maker Token Amount: %s\n", cancelEvent.CancelledMakerTokenAmount.Hex())
    fmt.Printf("Cancelled Taker Token Amount: %s\n", cancelEvent.CancelledTakerTokenAmount.Hex())
    fmt.Printf("Tokens: %s\n", hexutil.Encode(cancelEvent.Tokens[:]))
    fmt.Printf("Order Hash: %s\n", hexutil.Encode(cancelEvent.OrderHash[:]))

case logErrorEvent.Hex():
    fmt.Printf("Log Name: LogError\n")

    errorID, err := strconv.ParseInt(vLog.Topics[1].Hex(), 16, 64)

```

Setting up the Client

```
if err != nil {
    log.Fatal(err)
}

errorEvent := &LogError{
    ErrorID: uint8(errorID),
    OrderHash: vLog.Topics[2],
}

fmt.Printf("Error ID: %d\n", errorEvent.ErrorID)
fmt.Printf("Order Hash: %s\n", hexutil.Encode(errorEvent.OrderHash))
}

fmt.Printf("\n\n")
}
```

solc version used for these examples

```
$ solc --version
0.4.11+commit.68ef5810.Emscripten.clang
```

Signatures

A digital [signature](#) allows non-repudiation as it means the person who signed the message had to be in possession of the [private key](#) and so therefore the message is authentic. Anyone can verify the authenticity of the message as long as they have the hash of the original data and the public key of the signer. Signatures are a fundamental component is [blockchain](#) and we'll learn how to generate and verify signatures in the next few lessons.

Generating a Signature

The components for generating a [signature](#) are: the signers [private key](#), and the hash of the data that will be signed. Any hashing algorithm may be used as long as the output is 32 bytes. We'll be using [Keccak-256](#) as the hashing algorithm which is what [Ethereum](#) prefers to use.

First we'll load [private key](#).

```
privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad0f:
if err != nil {
    log.Fatal(err)
}
```

Next we'll take the [Keccak-256](#) of the data that we wish to sign, in this case it'll be the word *hello*. The [go-ethereum](#) `crypto` package provides a handy `Keccak256Hash` method for doing this.

```
data := []byte("hello")
hash := crypto.Keccak256Hash(data)
fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517b
```

Finally we sign the hash with our private, which gives us the [signature](#).

```
signature, err := crypto.Sign(hash.Bytes(), privateKey)
if err != nil {
    log.Fatal(err)
}

fmt.Println(hexutil.Encode(signature)) // 0x789a80053e4927d0a898db8e065
```

Now that we have successfully generated the [signature](#), in the next section we'll learn how to verify that the [signature](#) indeed was signed by the holder of that [private key](#).

Full code

[signature_generate.go](#)

```
package main

import (
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/crypto"
)

func main() {
    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad")
    if err != nil {
        log.Fatal(err)
    }

    data := []byte("hello")
    hash := crypto.Keccak256Hash(data)
    fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517

    signature, err := crypto.Sign(hash.Bytes(), privateKey)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(hexutil.Encode(signature)) // 0x789a80053e4927d0a898db8eC
}
```

Verifying a Signature

In the previous section we learned how to sign a piece of data with a [private key](#) in order to generate a [signature](#). Now we'll learn how to verify the authenticity of the [signature](#).

We need to have 3 things to verify the [signature](#): the [signature](#), the hash of the original data, and the public key of the signer. With this information we can determine if the [private key](#) holder of the public key pair did indeed sign the message.

First we'll need the public key in bytes format.

```
publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)
```

Next we'll need the original data hashed. In the previous lesson we used [Keccak-256](#) to generate the hash, so we'll do the same in order to verify the [signature](#).

```
data := []byte("hello")
hash := crypto.Keccak256Hash(data)
fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517b
```

Now assuming we have the [signature](#) in bytes format, we can call `Ecrecover` (elliptic curve [signature](#) recover) from the [go-ethereum](#) `crypto` package to retrieve the public key of the signer. This function takes in the hash and [signature](#) in bytes format.

```
sigPublicKey, err := crypto.Ecrecover(hash.Bytes(), signature)
if err != nil {
    log.Fatal(err)
}
```

To verify we simply now have to compare the [signature](#)'s public key with the expected public key and if they match then the expected public key holder is indeed the signer of the original message.

```
matches := bytes.Equal(sigPublicKey, publicKeyBytes)
fmt.Println(matches) // true
```

There's also the `SigToPub` method which does the same thing except it'll return the [signature](#)'s public key in the ECDSA type.

```
sigPublicKeyECDSA, err := crypto.SigToPub(hash.Bytes(), signature)
if err != nil {
    log.Fatal(err)
}

sigPublicKeyBytes := crypto.FromECDSAPub(sigPublicKeyECDSA)
matches = bytes.Equal(sigPublicKeyBytes, publicKeyBytes)
fmt.Println(matches) // true
```

For convenience, the crypto package provides the `VerifySignature` function which takes in the `signature`, hash of the original data, and the public key in bytes format. It returns a boolean which will be true if the public key matches the `signature`'s signer. An important gotcha is that we must first remove the last byte of the signature because it's the ECDSA recover ID which must not be included.

```
signatureNoRecoverID := signature[:len(signature)-1] // remove recovery ID
verified := crypto.VerifySignature(publicKeyBytes, hash.Bytes(), signatureNo
fmt.Println(verified) // true
```

These are the basics in generating and verifying ECDSA signatures with the `go-ethereum` package.

Full code

[signature_verify.go](#)


```

package main

import (
    "bytes"
    "crypto/ecdsa"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/crypto"
)

func main() {
    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad")
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
    }

    publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)

    data := []byte("hello")
    hash := crypto.Keccak256Hash(data)
    fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517

    signature, err := crypto.Sign(hash.Bytes(), privateKey)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(hexutil.Encode(signature)) // 0x789a80053e4927d0a898db8e0

    sigPublicKey, err := crypto.Ecrecover(hash.Bytes(), signature)
    if err != nil {
        log.Fatal(err)
    }

    matches := bytes.Equal(sigPublicKey, publicKeyBytes)
    fmt.Println(matches) // true

    sigPublicKeyECDSA, err := crypto.SigToPub(hash.Bytes(), signature)
    if err != nil {
        log.Fatal(err)
    }

```

Setting up the Client

```
}  
  
sigPublicKeyBytes := crypto.FromECDSAPub(sigPublicKeyECDSA)  
matches = bytes.Equal(sigPublicKeyBytes, publicKeyBytes)  
fmt.Println(matches) // true  
  
signatureNoRecoverID := signature[:len(signature)-1] // remove recovery  
verified := crypto.VerifySignature(publicKeyBytes, hash.Bytes(), signatureNoRecoverID)  
fmt.Println(verified) // true  
}
```

Testing

- [Faucets](#)
- [Using a Simulated Client](#)

Faucets

A faucet is where you can acquire free [testnet] ETH to use while testing.

Below are faucet links to each respective testnet.

- **Ropsten** testnet - <https://faucet.ropsten.be>
- **Rinkeby** testnet - <https://faucet.rinkeby.io>
- **Kovan** testnet - <https://gitter.im/kovan-testnet/faucet>
- **Sokol** testnet - <https://faucet-sokol.herokuapp.com>

Using a Simulated Client

You can use a simulated client for testing your transactions locally quickly and easily, ideal for unit tests. In order to get started we're going to need an account with some initial ETH in it. To do that first generate an account [private key](#).

```
privateKey, err := crypto.GenerateKey()
if err != nil {
    log.Fatal(err)
}
```

Then create a `NewKeyedTransactor` from the `accounts/abi/bind` package passing the [private key](#).

```
auth := bind.NewKeyedTransactor(privateKey)
```

The next step is to create a genesis account and assign it an initial balance. We'll be using the `GenesisAccount` type from the `core` package.

```
balance := new(big.Int)
balance.SetString("1000000000000000000", 10) // 10 eth in wei

address := auth.From
genesisAlloc := map[common.Address]core.GenesisAccount{
    address: {
        Balance: balance,
    },
}
```

Now we pass the genesis allocation struct and a configured block [gas](#) limit to the `NewSimulatedBackend` method from the `accounts/abi/bind/backends` package which will return a new simulated [ethereum](#) client.

```
blockGasLimit := uint64(4712388)
client := backends.NewSimulatedBackend(genesisAlloc, blockGasLimit)
```

You can use this client as you'd normally would. As an example, we'll construct a new transaction and broadcast it.

```

fromAddress := auth.From
nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
if err != nil {
    log.Fatal(err)
}

value := big.NewInt(1000000000000000000) // in wei (1 eth)
gasLimit := uint64(21000)                // in units
gasPrice, err := client.SuggestGasPrice(context.Background())
if err != nil {
    log.Fatal(err)
}

toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4fa1
var data []byte
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, dat
chainID := big.NewInt(1)
signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKe
if err != nil {
    log.Fatal(err)
}

err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s\n", signedTx.Hash().Hex()) // tx sent: 0xec3ceb05642c

```

By now you're probably wondering when will the transaction actually get mined. Well in order to "mine" it, there's one additional important thing you must do; call `Commit` on the client to commit a new mined block.

```
client.Commit()
```

Now you can fetch the transaction receipt and see that it was processed.

```
receipt, err := client.TransactionReceipt(context.Background(), signedTx.Hash)
if err != nil {
    log.Fatal(err)
}
if receipt == nil {
    log.Fatal("receipt is nil. Forgot to commit?")
}

fmt.Printf("status: %v\n", receipt.Status) // status: 1
```

So remember that the simulated client allows you to manually mine blocks at your command using the simulated client's `Commit` method.

Full code

[client_simulated.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/accounts/abi/bind/backends"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/crypto"
)

func main() {
    privateKey, err := crypto.GenerateKey()
    if err != nil {
        log.Fatal(err)
    }

    auth := bind.NewKeyedTransactor(privateKey)

    balance := new(big.Int)
    balance.SetString("1000000000000000000", 10) // 10 eth in wei

    address := auth.From
    genesisAlloc := map[common.Address]core.GenesisAccount{
        address: {
            Balance: balance,
        },
    }

    blockGasLimit := uint64(4712388)
    client := backends.NewSimulatedBackend(genesisAlloc, blockGasLimit)

    fromAddress := auth.From
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(1000000000000000000) // in wei (1 eth)
    gasLimit := uint64(21000)                // in units
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

```



```

}

toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f
var data []byte
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, d

chainID := big.NewInt(1)
signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), private
if err != nil {
    log.Fatal(err)
}

err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tx sent: %s\n", signedTx.Hash().Hex()) // tx sent: 0xec3ceb0564

client.Commit()

receipt, err := client.TransactionReceipt(context.Background(), signedTx.H
if err != nil {
    log.Fatal(err)
}
if receipt == nil {
    log.Fatal("receipt is nil. Forgot to commit?")
}

fmt.Printf("status: %v\n", receipt.Status) // status: 1
}

```

Swarm

[Swarm](#) in [Ethereum](#)'s decentralized and distributed storage solution, comparable to [IPFS](#). [Swarm](#) is a peer to peer data sharing network in which files are addressed by the hash of their content. Similar to Bittorrent, it is possible to fetch the data from many nodes at once and as long as a single [node](#) hosts a piece of data, it will remain accessible everywhere. This approach makes it possible to distribute data without having to host any kind of server - data accessibility is location independent. Other nodes in the network can be incentivised to replicate and store the data themselves, obviating the need for hosting services when the original nodes are not connected to the network.

[Swarm](#)'s incentive mechanism, Swap ([Swarm](#) Accounting Protocol), is a protocol by which peers in the [Swarm](#) network keep track of chunks delivered and received and the resulting (micro-) payments owed. On its own, SWAP can function in a wider context however it's usually presented as a generic micropayment scheme suited for pairwise accounting between peers. while generic by design, the first use of it is for accounting of bandwidth as part of the incentivisation of data transfer in the [Swarm](#) decentralised peer to peer storage network.

Setting up Swarm

To run **swarm** you first need to install **geth** and **bzzd** which is the **swarm** daemon.

```
go install github.com/ethereum/go-ethereum/cmd/geth
go install github.com/ethersphere/swarm/cmd/swarm
```

Now we'll generate a new **geth** account.

```
$ geth account new
```

```
Your new account is locked with a password. Please give a password. Do not
Passphrase:
Repeat passphrase:
Address: {970ef9790b54425bea2c02e25cab01e48cf92573}
```

Export the environment variable **BZZKEY** mapping to the **geth** account address we just generated.

```
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
```

And now run **swarm** with the specified account to be our **swarm** account. **Swarm** by default will run on port **8500**.

```
$ swarm --bzzaccount $BZZKEY
Unlocking swarm account 0x970EF9790B54425BEA2C02e25cAb01E48CF925
Passphrase:
WARN [06-12|13:11:41] Starting Swarm service
```

Now that we have the **swarm** daemon set up and running, let's learn how to upload files to **swarm** in the **next section**.

Full code

Commands

Setting up the Client

```
go install github.com/ethereum/go-ethereum/cmd/geth
go install github.com/ethersphere/swarm/cmd/swarm
geth account new
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
swarm --bzzaccount $BZZKEY
```

Uploading Files to Swarm

In the [previous section](#) we setup a [swarm node](#) running as a daemon on port 8500. Now import the [swarm](#) package ethersphere `swarm/api/client`. I'll be aliasing the package to `bzzclient`.

```
import (
    bzzclient "github.com/ethersphere/swarm/api/client"
)
```

Invoke `NewClient` function passing it the [swarm](#) daemon url.

```
client := bzzclient.NewClient("http://127.0.0.1:8500")
```

Create an example text file `hello.txt` with the content *hello world*. We'll be uploading this to [swarm](#).

```
hello world
```

In our [Go](#) application we'll open the file we just created using `Open` from the client package. This function will return a `File` type which represents a file in a [swarm](#) manifest and is used for uploading and downloading content to and from [swarm](#).

```
file, err := bzzclient.Open("hello.txt")
if err != nil {
    log.Fatal(err)
}
```

Now we can invoke the `Upload` function from our client instance giving it the file object. The second argument is an optional existing manifest string to add the file to, otherwise it'll create one for us. The third argument is if we want our data to be encrypted.

The hash returned is the [swarm](#) hash of a manifest that contains the `hello.txt` file as its only entry. So by default both the primary content and the manifest is uploaded. The manifest makes sure you could retrieve the file with the correct mime type.

```
manifestHash, err := client.Upload(file, "", false)
if err != nil {
    log.Fatal(err)
}

fmt.Println(manifestHash) // 2e0849490b62e706a5f1cb8e7219db7b01677f2
```

Now we can access our file at

bzz://2e0849490b62e706a5f1cb8e7219db7b01677f2a859bac4b5f522afd2a5f02c0 which learn how to do in the [next section](#).

Full code

Commands

```
geth account new
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
swarm --bzzaccount $BZZKEY
```

[hello.txt](#)

```
hello world
```

[swarm_upload.go](#)

```
package main

import (
    "fmt"
    "log"

    bzzclient "github.com/ethersphere/swarm/api/client"
)

func main() {
    client := bzzclient.NewClient("http://127.0.0.1:8500")

    file, err := bzzclient.Open("hello.txt")
    if err != nil {
        log.Fatal(err)
    }

    manifestHash, err := client.Upload(file, "", false, false, false)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(manifestHash) // 2e0849490b62e706a5f1cb8e7219db7b01675
}
```

Downloading Files from Swarm

In the [previous section](#) we uploaded a `hello.txt` file to [swarm](#) and in return we got a manifest hash.

```
manifestHash := "f9192507e2e8e118bfedac428c3aa1dec4ae156e954128ec!
```

Let's inspect the manifest by downloading it first by calling

`DownloadManifest`.

```
manifest, isEncrypted, err := client.DownloadManifest(manifestHash)
if err != nil {
    log.Fatal(err)
}
```

We can iterate over the manifest entries and see what the content-type, size, and content hash are.

```
for _, entry := range manifest.Entries {
    fmt.Println(entry.Hash)    // 42179060941352ba7b400b16c40f1e1290423
    fmt.Println(entry.ContentType) // text/plain; charset=utf-8
    fmt.Println(entry.Path)    // ""
}
```

If you're familiar with [swarm](#) urls, they're in the format

`bzz:./<hash>/<path>`, so in order to download the file we specify the manifest hash and path. The path in this case is an empty string. We pass this data to the `Download` function and get back a file object.

```
file, err := client.Download(manifestHash, "")
if err != nil {
    log.Fatal(err)
}
```

We may now read and print the contents of the returned file reader.


```
content, err := ioutil.ReadAll(file)
if err != nil {
    log.Fatal(err)
}

fmt.Println(string(content)) // hello world
```

As expected, it logs *hello world* which was what our original file contained.

Full code

Commands

```
geth account new
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
swarm --bzzaccount $BZZKEY
```

[swarm_download.go](#)

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"

    bzzclient "github.com/ethersphere/swarm/api/client"
)

func main() {
    client := bzzclient.NewClient("http://127.0.0.1:8500")
    manifestHash := "2e0849490b62e706a5f1cb8e7219db7b01677f2a859bac"
    manifest, isEncrypted, err := client.DownloadManifest(manifestHash)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(isEncrypted) // false

    for _, entry := range manifest.Entries {
        fmt.Println(entry.Hash)    // 42179060941352ba7b400b16c40f1e1290
        fmt.Println(entry.ContentType) // text/plain; charset=utf-8
        fmt.Println(entry.Size)    // 12
        fmt.Println(entry.Path)    // ""
    }

    file, err := client.Download(manifestHash, "")
    if err != nil {
        log.Fatal(err)
    }

    content, err := ioutil.ReadAll(file)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(content)) // hello world
}
```

Whisper

[Whisper](#) is a simple [peer-to-peer](#) identity-based messaging system designed to be a building block in the next generation of decentralized applications. It was designed to provide resilience and privacy at considerable expense. In the upcoming sections we'll set up an [Ethereum node](#) with [whisper](#) support and then we'll learn how to send and receive encrypted messages on the [whisper](#) protocol.

Connecting Whisper Client

To use [whisper](#), we must first connect to an [Ethereum node](#) running [whisper](#). Unfortunately, public gateways such as [infura](#) don't support [whisper](#) because there is no incentive for processing the messages for free. [Infura](#) might support [whisper](#) in the near future but for now we must run our own [geth node](#). Once you [install geth](#), run it with the `--shh` flag on to enable the [whisper](#) protocol, as well as the `--ws` flag to enable websocket support in order to receive messages in real time, and also enable the `--rpc` flag because we'll be communicating over RPC.

```
geth --rpc --shh --ws
```

Now in our [Go](#) application we'll import the [go-ethereum whisper](#) client package found at [whisper/shhclient](#) and initialize the client to connect our local [geth node](#) over websockets using the default websocket port 8546 .

```
client, err := shhclient.Dial("ws://127.0.0.1:8546")
if err != nil {
    log.Fatal(err)
}

_ = client // we'll be using this in the next section
```

Now that we're dialed in let's create a key pair for encrypting the message before we send it in the [next section](#).

Full code

Commands

```
geth --rpc --shh --ws
```

[whisper_client.go](#)

```
package main

import (
    "log"

    "github.com/ethereum/go-ethereum/whisper/shhclient"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    _ = client // we'll be using this in the next section
    fmt.Println("we have a whisper connection")
}
```

Generating Whisper Key Pair

In [whisper](#), messages have to be encrypted with either a symmetric or an asymmetric key to prevent them from being read by anyone other than the intended recipient.

After you've connected to the [whisper](#) client you'll need to call the client's `NewKeyPair` method to generate a new public and private pair that the [node](#) will manage. The result of this function will be a unique ID that references the key pair which we'll be using for encrypting and decrypting the message in the next few sections.

```
keyID, err := client.NewKeyPair(context.Background())
if err != nil {
    log.Fatal(err)
}

fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88b9l
```

Let's learn how to send an encrypted message in the [next section](#).

Full code

Commands

```
geth --rpc --shh --ws
```

[whisper_keypair.go](#)

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/whisper/shhclient"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    keyID, err := client.NewKeyPair(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88k
}
```

Sending Messages on Whisper

Before we're able to create a message, we must first have a public key to encrypt the message. In the [previous section](#) we learned how to generate a public and [private key](#) pair using the `NewKeyPair` function which returned a key ID that references this key pair. We now have to call the `PublicKey` function to read the key pair's public key in bytes format which we'll be using to encrypt the message.

```
publicKey, err := client.PublicKey(context.Background(), keyID)
if err != nil {
    log.Print(err)
}

fmt.Println(hexutil.Encode(publicKey)) // 0x04f17356fd52b0d13e5ede84f998
```

Now we'll construct our [whisper](#) message by initializing the `NewMessage` struct from the [go-ethereum](#) `whisper/whisperv6` package, which requires the following properties:

- `Payload` as the message content in bytes format
- `PublicKey` as the key we'll use for [encryption](#)
- `TTL` as the time-to-live in seconds for the message
- `PowTime` as maximal time in seconds to be spent on [proof of work](#).
- `PowTarget` as the minimal PoW target required for this message.

```
message := whisperv6.NewMessage{
    Payload: []byte("Hello"),
    PublicKey: publicKey,
    TTL: 60,
    PowTime: 2,
    PowTarget: 2.5,
}
```

We can now broadcast to the network by invoking the client's `Post` function giving it the message, will it'll return a hash of the message.


```
messageHash, err := client.Post(context.Background(), message)
if err != nil {
    log.Fatal(err)
}

fmt.Println(messageHash) // 0xdbfc815d3d122a90d7fb44d1fc6a46f3d76ec7
```

In the [next section](#) we'll see how we can create a message subscription to be able to receive the messages in real time.

Full code

Commands

```
geth --shh --rpc --ws
```

[whisper_send.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/whisper/shhclient"
    "github.com/ethereum/go-ethereum/whisper/whisperv6"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    keyID, err := client.NewKeyPair(context.Background())
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88k

    publicKey, err := client.PublicKey(context.Background(), keyID)
    if err != nil {
        log.Print(err)
    }
    fmt.Println(hexutil.Encode(publicKey)) // 0x04f17356fd52b0d13e5ede84f9

    message := whisperv6.NewMessage{
        Payload: []byte("Hello"),
        PublicKey: publicKey,
        TTL:     60,
        PowTime: 2,
        PowTarget: 2.5,
    }
    messageHash, err := client.Post(context.Background(), message)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(messageHash) // 0xdbfc815d3d122a90d7fb44d1fc6a46f3d76e
}

```

Subscribing to Whisper Messages

In this section we'll be subscribing to `whisper` messages over websockets. First thing we need is a channel that will be receiving `whisper` messages in the `Message` type from the `whisper/whisperv6` package.

```
messages := make(chan *whisperv6.Message)
```

Before we invoke a subscription, we first need to determine the criteria. From the `whisperv6` package initialize a new `Criteria` object. Since we're only interested in messages targeted to us, we'll set the `PrivateKeyID` property on the criteria object to the same key ID we used for encrypting messages.

```
criteria := whisperv6.Criteria{
    PrivateKeyID: keyID,
}
```

Next we invoke the client's `SubscribeMessages` method which subscribes to messages that match the given criteria. This method is not supported over HTTP; only supported on bi-directional connections such as websockets and IPC. The last argument is the messages channel we created earlier.

```
sub, err := client.SubscribeMessages(context.Background(), criteria, messages)
if err != nil {
    log.Fatal(err)
}
```

Now that we have our subscription, we can use a `select` statement to read messages as they come in and also to handle errors from the subscription. If you recall from the previous section, the message content is in the `Payload` property as a byte slice which we can convert back to a human readable string.

```
for {
  select {
    case err := <-sub.Err():
      log.Fatal(err)
    case message := <-messages:
      fmt.Printf(string(message.Payload)) // "Hello"
  }
}
```

Check out the full code below for a complete working example. That's all there is to [whisper](#) message subscriptions.

Full code

Commands

```
geth --shh --rpc --ws
```

[whisper_subscribe.go](#)

```

package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "runtime"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/whisper/shhclient"
    "github.com/ethereum/go-ethereum/whisper/whisperv6"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    keyID, err := client.NewKeyPair(context.Background())
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88k

    messages := make(chan *whisperv6.Message)
    criteria := whisperv6.Criteria{
        PrivateKeyID: keyID,
    }
    sub, err := client.SubscribeMessages(context.Background(), criteria, mess
    if err != nil {
        log.Fatal(err)
    }

    go func() {
        for {
            select {
            case err := <-sub.Err():
                log.Fatal(err)
            case message := <-messages:
                fmt.Printf(string(message.Payload)) // "Hello"
                os.Exit(0)
            }
        }
    }()

    publicKey, err := client.PublicKey(context.Background(), keyID)

```

```
if err != nil {
    log.Print(err)
}
fmt.Println(hexutil.Encode(publicKey)) // 0x04f17356fd52b0d13e5ede84f9

message := whisperv6.NewMessage{
    Payload: []byte("Hello"),
    PublicKey: publicKey,
    TTL: 60,
    PowTime: 2,
    PowTarget: 2.5,
}

messageHash, err := client.Post(context.Background(), message)
if err != nil {
    log.Fatal(err)
}
fmt.Println(messageHash) // 0xdbfc815d3d122a90d7fb44d1fc6a46f3d76e

runtime.Goexit() // wait for goroutines to finish
}
```

Utilities

- [Collection of Utility Functions](#)

Collection of Utility Functions

The utility functions' implementation are found below in the [full code](#) section. They are generous in what they accept. Here we'll be showing examples of usage.

Derive the [Ethereum](#) public address from a public key:

```
publicKeyBytes, _ := hex.DecodeString("049a7df67f79246283fdc93af76d4f8...")
address := util.PublicKeyBytesToAddress(publicKeyBytes)
fmt.Println(address.Hex()) // 0x96216849c49358B10257cb55b28eA603c874f...
```

Check if an address is a valid [Ethereum](#) address:

```
valid := util.IsValidAddress("0x323b5d4c32345ced77393b3530b1eed0f3464...")
fmt.Println(valid) // true
```

Check if an address is a zero address.

```
zeroed := util.IsZeroAddress("0x0")
fmt.Println(zeroed) // true
```

Convert a decimal to wei. The second argument is the number of decimals.

```
wei := util.ToWei(0.02, 18)
fmt.Println(wei) // 200000000000000000
```

Convert wei to decimals. The second argument is the number of decimals.

```
wei := new(big.Int)
wei.SetString("200000000000000000", 10)
eth := util.ToDecimal(wei, 18)
fmt.Println(eth) // 0.02
```

Calculate the [gas](#) cost given the [gas](#) limit and [gas](#) price.

Setting up the Client

```
gasLimit := uint64(21000)
gasPrice := new(big.Int)
gasPrice.SetString("2000000000", 10)
gasCost := util.CalcGasCost(gasLimit, gasPrice)
fmt.Println(gasCost) // 42000000000000
```

Retrieve the R, S, and V values from a [signature](#).

```
sig := "0x789a80053e4927d0a898db8e065e948f5cf086e32f9ccaa54c1908e2
r, s, v := util.SigRSV(sig)
fmt.Println(hexutil.Encode(r[:])[2:]) // 789a80053e4927d0a898db8e065e948f
fmt.Println(hexutil.Encode(s[:])[2:]) // 2621578113d5bb62d509bf6049b8fb54
fmt.Println(v) // 28
```

Full code

[util.go](#)

```

package util

import (
    "math/big"
    "reflect"
    "regexp"
    "strconv"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/shopspring/decimal"
)

// IsValidAddress validate hex address
func IsValidAddress(iaddress interface{}) bool {
    re := regexp.MustCompile("^0x[0-9a-fA-F]{40}$")
    switch v := iaddress.(type) {
    case string:
        return re.MatchString(v)
    case common.Address:
        return re.MatchString(v.Hex())
    default:
        return false
    }
}

// IsZeroAddress validate if it's a 0 address
func IsZeroAddress(iaddress interface{}) bool {
    var address common.Address
    switch v := iaddress.(type) {
    case string:
        address = common.HexToAddress(v)
    case common.Address:
        address = v
    default:
        return false
    }

    zeroAddressBytes := common.FromHex("0x00000000000000000000000000000000")
    addressBytes := address.Bytes()
    return reflect.DeepEqual(addressBytes, zeroAddressBytes)
}

// ToDecimal wei to decimals
func ToDecimal(ivalue interface{}, decimals int) decimal.Decimal {
    value := new(big.Int)
    switch v := ivalue.(type) {
    case string:

```

```

    value.SetString(v, 10)
case *big.Int:
    value = v
}

mul := decimal.NewFromFloat(float64(10)).Pow(decimal.NewFromFloat(float64(10)))
num, _ := decimal.NewFromString(value.String())
result := num.Div(mul)

return result
}

// ToWei decimals to wei
func ToWei(iamount interface{}, decimals int) *big.Int {
    amount := decimal.NewFromFloat(0)
    switch v := iamount.(type) {
    case string:
        amount, _ = decimal.NewFromString(v)
    case float64:
        amount = decimal.NewFromFloat(v)
    case int64:
        amount = decimal.NewFromFloat(float64(v))
    case decimal.Decimal:
        amount = v
    case *decimal.Decimal:
        amount = *v
    }

    mul := decimal.NewFromFloat(float64(10)).Pow(decimal.NewFromFloat(float64(10)))
    result := amount.Mul(mul)

    wei := new(big.Int)
    wei.SetString(result.String(), 10)

    return wei
}

// CalcGasCost calculate gas cost given gas limit (units) and gas price (wei)
func CalcGasCost(gasLimit uint64, gasPrice *big.Int) *big.Int {
    gasLimitBig := big.NewInt(int64(gasLimit))
    return gasLimitBig.Mul(gasLimitBig, gasPrice)
}

// SigRSV signatures R S V returned as arrays
func SigRSV(isig interface{}) ([32]byte, [32]byte, uint8) {
    var sig []byte
    switch v := isig.(type) {
    case []byte:

```

```
    sig = v
    case string:
        sig, _ = hexutil.Decode(v)
    }

    sigstr := common.Bytes2Hex(sig)
    rS := sigstr[0:64]
    sS := sigstr[64:128]
    R := [32]byte{}
    S := [32]byte{}
    copy(R[:], common.FromHex(rS))
    copy(S[:], common.FromHex(sS))
    vStr := sigstr[128:130]
    vI, _ := strconv.Atoi(vStr)
    V := uint8(vI + 27)

    return R, S, V
}
```

test file: [util_test.go](#)

Glossary

Addresses

Used to receive and send transactions on the network. An address is a string of alphanumeric characters, but can also be represented as a scannable QR code. They are derived from the public/private ECDSA key pair.

Agreement Ledgers

[Distributed ledgers](#) used by two or more parties to negotiate and reach and agreement.

Altcoin

An abbreviation of "[Bitcoin](#) alternative". Currently, the majority of altcoins are forks of [Bitcoin](#) with usually minor changes to the [proof of work](#) (POW) algorithm of the [Bitcoin blockchain](#). The most prominent [altcoin](#) is [Litecoin](#). [Litecoin](#) introduces changes to the original [Bitcoin](#) protocol such as decreased block generation time, increased maximum number of coins and different hashing algorithm.

Attestation Ledgers

[Distributed ledgers](#) that provide a durable record of agreements, commitments or statements, providing evidence (attestation) that these agreements, commitments or statements were made.

ASIC

An acronym for "Application Specific Integrated Circuit". ASICs are silicon chips specifically designed to do a single task. In the case of [bitcoin](#), they are designed to process SHA-256 hashing problems to mine new bitcoins.

Bitcoin

Currently the most well known [cryptocurrency](#), based on the proof-of-work [blockchain](#).

Blockchain

A type of distributed [ledger](#), comprised of unchangeable, digitally recorded data in packages called blocks (rather like collating them on to a single sheet of paper). Each block is then 'chained' to the next block, using a cryptographic [signature](#). This allows block chains to be used like a [ledger](#), which can be shared and accessed by anyone with the appropriate permissions.

Block Ciphers

A method of encrypting text (to produce ciphertext) in which a cryptographic key and algorithm are applied to a block of data at once as a group rather than to one bit at a time.

Block Height

Refers to the number of blocks connected together in the block chain. For example, Height 0, would be the very first block, which is also called the Genesis Block.

Block Rewards

Rewards given to a miner which has successfully hashed a [transaction block](#). [Block rewards](#) can be a mixture of coins and [transaction fees](#), depending on the policy used by the [cryptocurrency](#) in question, and whether all of the coins have already been successfully mined. The current block reward for the [Bitcoin](#) network is 25 bitcoins for each block.

Central Ledger

Refers to a [ledger](#) maintained by a central agency.

Chain Linking

The process of connecting two blockchains with each other, thus allowing transactions between the chains to take place. This will allow blockchains like [Bitcoin](#) to communicate with other sidechains, allowing the exchange of assets between them

Cipher

The algorithm used for the [encryption](#) and/or [decryption](#) of information. In common language, 'cipher' is also used to refer to an [encryption](#) message, also known as 'code'.

Confirmation

The [blockchain](#) transaction has been verified by the network. This happens through a process known as [mining](#), in a proof-of-work system (e.g. [Bitcoin](#)). Once a transaction is confirmed, it cannot be reversed or double spent. The more confirmations a transaction has, the harder it becomes to perform a [double spend](#) attack.

Consensus Process

A group of peers responsible for maintaining a distributed [ledger](#) use to reach consensus on the [ledger](#)'s contents.

Consortium Blockchain

A [blockchain](#) where the [consensus process](#) is controlled by a pre-selected set of nodes; for example, one might imagine a consortium of 15 financial institutions, each of which operates a [node](#) and of which ten must sign every block for the block to be valid. The right to read the [blockchain](#) may be public or restricted to the participants. There are also hybrid routes such as the root hashes of the blocks being public together with an API that allows members of the public to make a limited number of queries and get back cryptographic proofs of some parts of the [blockchain](#) state. These blockchains may be considered "partially decentralized".

Cryptoanalysis

The study of methods for obtaining the meaning of encrypted information, without access to the secret information that is normally required to do so.

Cryptocurrency

A form of digital currency based on mathematics, where [encryption](#) techniques are used to regulate the generation of units of currency and verify the transfer of funds. Furthermore, cryptocurrencies operate independently of a central bank.

Cryptography

Refers to the process of encrypting and decrypting information.

dApp

A decentralized application that must be completely open-source, it must operate autonomously, and with no entity controlling the majority of its tokens.

DAO

(Decentralized Autonomous Organization) can be thought of as a corporation run without any human involvement under the control of an incorruptible set of business rules.

The DAO

A venture capital fund built on [Ethereum](#) that caused a soft and hark fork.

Decryption

The process of turning [cipher](#)-text back into plaintext

Encryption

The process of turning a clear-text message (plaintext) into a data stream ([cipher](#)-text), which looks like a meaningless and random sequence of bits.

ERC

[ERC](#) stands for [Ethereum](#) Request for Comments. An [ERC](#) is a proposal for [Ethereum](#).

ERC-20

A specification for tokens on [Ethereum](#).

Ether

The native [token](#) of the [Ethereum blockchain](#) which is used to pay for [transaction fees](#), miner rewards and other services on the network.

Ethereum

An open software platform based on [blockchain](#) technology that enables developers to write smart contracts and build and deploy decentralized applications.

Ethereum Classic

A split from an existing [cryptocurrency](#), [Ethereum](#) after a hard [fork](#). To learn more about this, [click here](#).

EVM

The [Ethereum](#) Virtual Machine.

EVM Bytecode

The programming language in which accounts on the [Ethereum blockchain](#) can contain code. The [EVM](#) code associated with an account is executed every time a message is sent to that account, and has the ability to read/write storage and itself send messages.

Digital Commodity

A scarce, electronically transferrable, intangible, with a market value.

Digital Identity

An online or networked identity adopted or claimed in cyberspace by an individual, organization, or electronic device.

Distributed Ledgers

A type of database that are spread across multiple sites, countries or institutions. Records are stored one after the other in a continuous [ledger](#). Distributed [ledger](#) data can be either "permissioned" or "unpermissioned" to control who can view it.

Difficulty

In Proof-of-Work [mining](#), is how hard it is to verify blocks in a [blockchain](#) network. In the [Bitcoin](#) network, the [difficulty](#) of [mining](#) adjusts verifying blocks every 2016 blocks. This is to keep block verification time at ten minutes.

Double Spend

Refers to a scenario, in the [Bitcoin](#) network, where someone tries to send a [bitcoin](#) transaction to two different recipients at the same time. However, once a [bitcoin](#) transaction is confirmed, it makes it nearly impossible to [double spend](#) it. The more confirmations that a particular transaction has, the harder it becomes to [double spend](#) the bitcoins.

Fiat currency

is any money declared by a government to be to be valid for meeting a financial obligation, like USD or EUR.

Fork

The creation of an ongoing alternative version of the [blockchain](#), by creating two blocks simultaneously on different parts of the network. This creates two parallel blockchains, where one of the two is the winning [blockchain](#).

Gas

A measurement roughly equivalent to computational steps (for [Ethereum](#)). Every transaction is required to include a [gas](#) limit and a fee that it is willing to pay per [gas](#); miners have the choice of including the transaction and collecting the fee or not. Every operation has a [gas](#) expenditure; for most operations it is ~3–10, although some expensive operations have expenditures up to 700 and a transaction itself has an expenditure of 21000.

Gas Cost

[Gas](#) cost is the [gas](#) limit multiplied by the [gas](#) price.

Gas Limit

Max number of computational units that the transaction should use up in the smart contract execution.

Gas Price

The price per computational unit.

Geth

An [Ethereum node](#) implementation in [Golang](#).

<https://github.com/ethereum/go-ethereum>

Go

[Go](#) is a programming language created at Google in 2009 by Robert Griesemer, Rob Pike, and Ken Thompson.

Golang

The [Go](#) programming language.

go-ethereum

The [Ethereum](#) implementation in [Golang](#).

Halving

Bitcoins have a finite supply, which makes them a scarce [digital commodity](#). The total amount of bitcoins that will ever be issued is 21 million. The number of bitcoins generated per block is decreased 50% every four years. This is called "[halving](#)". The final [halving](#) will take place in the year 2140.

Hard fork

A change to the [blockchain](#) protocol that makes previously invalid blocks/transactions valid, and therefore requires all users to upgrade their clients.

Hashcash

A proof-of-work system used to limit email spam and denial-of-service attacks, and more recently has become known for its use in [bitcoin](#) (and other cryptocurrencies) as part of the [mining](#) algorithm.

Hashrate

The number of hashes that can be performed by a [bitcoin](#) miner in a given period of time (usually a second).

HD Wallet

An [HD Wallet](#), or Hierarchical Deterministic [wallet](#), is a new-age digital [wallet](#) that automatically generates a hierarchical tree-like structure of private/public [addresses](#) (or keys), thereby addressing the problem of the user having to generate them on his own.

Infura

[Infura](#) provides secure, reliable, and scalable gateways to the [Ethereum](#) network. <https://infura.io/>

Initial Coin Offering

(ICO) is an event in which a new [cryptocurrency](#) sells advance tokens from its overall coinbase, in exchange for upfront capital. ICOs are frequently used for developers of a new [cryptocurrency](#) to raise capital.

IPFS

InterPlanetary File System ([IPFS](#)) is a protocol and network designed to create a content-addressable, [peer-to-peer](#) method of storing and sharing hypermedia in a distributed file system.

Keccak-256

The hashing algorithm used in [Ethereum](#).

Keystore

A file containing an encrypted [wallet](#) private keys and [wallet](#) metadata.

Kovan

A proof-of-authority testnet on the [Ethereum blockchain](#). Supported by [Parity](#) only.

Ledger

An append-only record store, where records are immutable and may hold more general information than financial records.

Litecoin

A [peer-to-peer cryptocurrency](#) based on the Scrypt proof-of-work network. Sometimes referred to as the silver of [bitcoin](#)'s gold.

Mining

The process by which transactions are verified and added to a [blockchain](#). This process of solving cryptographic problems using computing hardware also triggers the release of cryptocurrencies.

Mnemonic

A [mnemonic](#) phrase, [mnemonic](#) recovery phrase or [mnemonic](#) seed is a list of words used as a seed to generate the master [private key](#) and master chain code for an [HD wallet](#).

Multi-signature

(multisig) [addresses](#) allow multiple parties to require more than one key to authorize a transaction. The needed number of signatures is agreed at the creation of the address. Multi [signature addresses](#) have a much greater resistance to theft.

Node

Any computer that connects to the [blockchain](#) network.

Nonce

A number only used once.

Full node

A [node](#) that fully enforces all of the rules of the [blockchain](#).

Parity

An [Ethereum](#) implementation written in the Rust language.

<https://github.com/paritytech/parity>

P2P

[P2P](#) stands for Peer to Peer.

Peer-to-peer

Refers to the decentralized interactions that happen between at least two parties in a highly interconnected network. [P2P](#) participants deal directly with each other through a single mediation point.

Permissioned Ledger

Is a [ledger](#) where actors must have permission to access the [ledger](#). Permissioned ledgers may have one or many owners. When a new record is added, the [ledger](#)'s integrity is checked by a limited [consensus process](#). This is carried out by trusted actors—government departments or banks, for example—which makes maintaining a shared record much simpler than the [consensus process](#) used by [unpermissioned ledgers](#).

Permissioned Blockchains

Provide highly-verifiable data sets because the [consensus process](#) creates a digital [signature](#), which can be seen by all parties.

Private Key

A string of data that shows you have access to bitcoins in a specific [wallet](#). Private keys can be thought of as a password; private keys must never be revealed to anyone but you, as they allow you to spend the bitcoins from your [bitcoin wallet](#) through a cryptographic [signature](#).

Proof of Authority

A consensus mechanism in a private [blockchain](#) which essentially gives one client (or a specific number of clients) with one particular [private key](#) the right to make all of the blocks in the [blockchain](#).

Proof of Stake

An alternative to the proof-of-work system, in which your existing stake in a [cryptocurrency](#) (the amount of that currency that you hold) is used to calculate the amount of that currency that you can mine.

Proof of Work

A system that ties [mining](#) capability to computational power. Blocks must be hashed, which is in itself an easy computational process, but an additional variable is added to the hashing process to make it more difficult. When a block is successfully hashed, the hashing must have taken some time and computational effort. Thus, a hashed block is considered [proof of work](#).

Protocols

Sets of formal rules describing how to transmit or exchange data, especially across a network.

Rinkeby

A proof-of-authority testnet on the [Ethereum blockchain](#). Supported by [Geth](#) only.

RLP

[Recursive Length Prefix \(RLP\)](#) is a standard to encode arbitrarily nested arrays of binary data. [RLP](#) is the main encoding method used to serialize objects in [Ethereum](#).

Ropsten

A proof-of-work testnet on the [Ethereum blockchain](#) which best simulates production environment. Supported by [Geth](#) and [Parity](#).

Script

An alternative [proof of work](#) system to SHA-256, designed to be particularly friendly to CPU and GPU miners, while offering little advantage to [ASIC](#) miners.

SHA256

The cryptographic function used as the basis for [bitcoin's proof of work](#) system.

Signature

A digital [signature](#) is a mathematical scheme for presenting the authenticity of digital messages or documents.

Smart contract

Contracts whose terms are recorded in a computer language instead of legal language. Smart contracts can be automatically executed by a computing system, such as a suitable distributed [ledger](#) system.

Soft fork

A change to the [bitcoin](#) protocol wherein only previously valid blocks/transactions are made invalid. Since old nodes will recognize the new blocks as valid, a softfork is backward-compatible. This kind of [fork](#) requires only a majority of the miners upgrading to enforce the new rules.

Sokol

A proof-of-authority testnet on the [Ethereum blockchain](#). Supported by [Parity](#) only.

Stream ciphers

A method of encrypting text (cyphertext) in which a cryptographic key and algorithm are applied to each binary digit in a data stream, one bit at a time.

Swarm

Decentralized file storage as part of [Ethereum](#).

Token

Is a [digital identity](#) for something that can be owned.

Tokenless Ledger

Refers to a distributed [ledger](#) that doesn't require a native currency to operate.

Transaction Block

A collection of transactions on the [bitcoin](#) network, gathered into a block that can then be hashed and added to the [blockchain](#).

Transaction Fees

Small fees imposed on some transactions sent across the [bitcoin](#) network. The transaction fee is awarded to the miner that successfully hashes the block containing the relevant transaction.

Unpermissioned ledgers

Blockchains that do not have a single owner; they cannot be owned. The purpose of an unpermissioned [ledger](#) is to allow anyone to contribute data to the [ledger](#) and for everyone in possession of the [ledger](#) to have identical copies.

Wallet

A file that contains a collection of private keys.

Whisper

A [peer-to-peer](#) messaging system as part of [Ethereum](#).

Resources

List of resources on [Ethereum](#), Solidity, and [Go](#).

Best Practices

- [Smart Contract Security Best Practices](#)
- [Security Considerations](#)
- [Solidity idiosyncrasies](#)

Help & Support

- StackExchange
 - [Ethereum](#)
- Reddit
 - [ethdev](#)
 - [golang](#)
- Gitter
 - [List of Gitter channels](#)

Community

- Reddit
 - [ethereum](#)
 - [ethtrader](#)
- Twitter
 - [ethereum](#)

Libraries

- [go-ethereum](#)
- [go-solidity-sha3](#)
- [go-ethereum-hdwallet](#)
- [go-ethutil](#)

Developer Tools

- [Truffle](#)
- [Infura](#)
- [Remix IDE](#)
- [Keccak-256 Online](#)

