GO (golang): Brief Guide To Programming a Blockchain with GO

A Workshop Using A Practical Example To Make Cryptocurrency Transactions Through Blockchain Technology With A Web Wallet

Jens Schendel

Version 1.0.0, March 2024

Please do **not** print out this PDF.

Table of Contents

Section 1 – Introduction to This Course and the Impact of Blockchain Technology in General

> *The key to growth is the introduction of higher dimensions of consciousness into our awareness.*
>
> *Lao Tzu*

Lecture 1 – The Obligatory About Me - Let Me Introduce myself

Hello and welcome to my workshop

GO (golang): Brief Guide To Programming a Blockchain with GO.

My name is Jens Schendel and I see myself as your tour guide who would like to introduce you to the concepts and principles of blockchain technology.

Originally, I learned Basic and Pascal a long time ago, but I have also worked some more with C and C++, a topic for which I also run what I like to jokingly call the "most insignificant channel on YouTube". Since I got to know Google's programming language GO, I have been enthusiastic about it and try to express this enthusiasm in courses for beginners and fairly advanced programmers here on Udemy.

Blockchain is a relatively new technology that is often overlooked in the shadow of its own main application, the cryptocurrency Bitcoin. Regardless, a decentralized database that enables transactions between participants without a mutual trust relationship with a third entity or even multiple entities may have more use cases.

Perhaps these are just not yet openly visible because the blockchain is largely perceived as "Bitcoin" only. I hope this course, in which you will program and implement blockchain technology yourself, will help you to recognize further possible applications and to understand the possibilities, but also the limits of this newly emerged technology.

I have been dedicated to Bitcoin since around the end of 2013 and since then I have discovered new aspects and mechanisms of this young technology that have come together to result in this first of all cryptocurrencies.

Blockchain itself aggregates a not inconsiderable amount of these sub-technologies and represents a fundamentally new technology in its own right. I never get tired of being fascinated by the precision and flawless uninterrupted continuation of this blockchain as well as of reporting on it in an evangelistic way. It is time to share some of this knowledge and hopefully enthusiasm and pass it on in this course.

I prepared several tasks for you to perform under my guidance. You will begin with developing and implementing your own source code from scratch. In the further course you will start several nodes, let them find each other over spawning a peer-to-peer network, get to know and create private and public keys, generate blockchain addresses, perform mining, strive for and achieve consensus on a net of distributed blockchain nodes and finally even start an online wallet and transfer cryptocurrency values from one wallet to another. Wow! And you will do all that using mainly

Google's programming language go. As you see there is a lot of fun waiting for you, let's get started!

Lecture 2 – The Impact of Bitcoin, Cryptocurrencies and Blockchain Technology on Society

In this lecture, you will take a nosedive into a topic that is not only turning upside down the world of finance but also shaping the very fabric of our digital society. I'm talking about the world of Bitcoin, cryptocurrencies, and blockchain technology in particular.

Now, you might be wondering, "What's the big deal about these digital coins and blockchain?" Let us hurry through some impressions of the impact of Bitcoin, other cryptocurrencies, and blockchain technology on our society.

Let's start with the rock star of the digital currency world – Bitcoin. Imagine a world where you can send money anywhere, anytime, without the need for intermediaries like banks. Bitcoin makes this possible. It's decentralized, borderless, and operates on a peer-to-peer network. The impact? Financial inclusion of the participants like never before. Take the example of remittances – Bitcoin allows people to send and receive money across borders without the hefty fees and delays associated with traditional banking systems.

Beyond Bitcoin, there are thousands of other digital currencies, some with its unique features. Ethereum, for instance, introduced the concept of smart contracts. These are self-executing contracts with the terms directly written into code. Think of it as a digital agreement that automatically enforces itself. This has enormous potential, from decentralized finance (DeFi) to supply chain management and beyond.

But what about the technology that makes all this possible? Enter blockchain. Blockchain is the underlying technology that powers most cryptocurrencies. It's a decentralized ledger that records transactions across a network of computers, ensuring transparency and security. One of the most exciting aspects is its potential to disrupt industries beyond finance. Consider supply chain management – with blockchain, we can trace the origin and journey of products, ensuring authenticity and reducing fraud.

In conclusion, the impact of Bitcoin, cryptocurrencies, and blockchain technology on society is profound. From financial inclusion to smart contracts you are witnessing a digital revolution. As students of this programming course, you're not just witnesses; you are on the way to become an architect of this new era.

Lecture 3 – A Side Note on the Term web3

I want to take a moment to demystify the buzz around "web3". It's the latest hot topic, but is it just a buzzword, or is there substance behind the hype?

Web3, often touted as the next evolution of the internet, promises a decentralized, user-centric online experience. However, let's not get too carried away with the buzz. The term "web3" is, in many cases, more of a marketing catchphrase than a clearly defined concept.

What is crucial to understand is that the real powerhouse behind these discussions is the underlying technology – blockchain. Blockchain is the backbone of cryptocurrencies like Bitcoin and Ethereum, providing a secure and transparent way to record transactions. But here's the kicker – blockchain is still a widely misunderstood or underexplored technology.

Before we rush into inventing arbitrary use cases to implement something called “web3”, it is imperative that we delve deeper into understanding the intricacies of blockchain. Its potential goes beyond the financial realm; it's a transformative force across various industries.

Don’t be swayed solely by the buzz around web3. Instead, invest your time and energy in unraveling the complexities of blockchain technology. By doing so, you position yourself to explore genuine innovations and contribute meaningfully to the evolution of the digital landscape.

Remember, the power of blockchain is not just in the promise of web3, but in the countless possibilities waiting to be unlocked through a thorough understanding of this groundbreaking technology.

Lecture 4 – Blockchain’s Effects on the Economy as Well as the Programmer HR Market

Look at the profound impact that blockchain is having on our economy and, more specifically, on the programmer HR market.

[https://www.linkedin.com/pulse/how-blockchain-can-benefit-us-your-daily-life-be-earning/]

Now, why should you care about blockchain? Let me tell you – it's not just about the cryptocurrencies; it's about a revolutionary technology reshaping the landscape of our economy.

Understanding blockchain is not merely a technological luxury but a strategic advantage. Consider the example of supply chain management. Blockchain's decentralized and transparent nature ensures an unalterable record of every transaction in a supply chain.

[https://www.xilinx.com/products/design-tools/resources/the-developers-guide-to-blockchain-development.html]

This not only reduces fraud but also enhances trust and efficiency. Imagine you, as an entrepreneur, realizing the potential to build a transparent and efficient supply chain solution using blockchain. That's a game-changer!

Even if you're not dreaming of starting your own venture, as an individual programmer, your world is about to witness a seismic shift. Understanding blockchain is becoming increasingly vital, and the demand for blockchain-savvy professionals is soaring.

[www.glassdoor.com]

A quick’n dirty research on www.glassdoor.com only with the search term “blockchain” brings up around 1700 results in the Unites States only.

It includes all kind of blockchain-related jobs from blockchain developer, blockchain architects, over analysts and researchers, to blockchain consultants. Employing companies like Coinbase, JP Morgan Chase become visible, and many salaries range around 3-digit numbers in Dollars.

Similar in Europe.

- France 835 Job Options
- Germany 572
- Spain 427
- Nigeria 40
- South Africa 57

But in Asia it looks more promising:

- India 1688
- Taiwan/China: 150 with names like Binance showing up

Singapore: 617, but let’s deduct the IBM jobs. They only show up here because they have the blockchain as a buzzword here in the company introduction. So around 50 deducted means still more than 550 Job opportunities.

That are not just numbers; that are lucrative opportunities knocking on your door!

[https://www.forbes.com/advisor/investing/cryptocurrency/crypto-market-outlook-forecast/]

I admit that in the past that looked already even much better. But keep in mind that this is 2024 after a long period of cryptocurrency winter, but spring is waiting just around the corner. The underlying technology in general will also benefit from the increasing acceptance and adoption of cryptocurrency payments. The demand will rise and the companies will choose the ones who are best prepared.

As an ordinary programmer, you might be wondering how this impacts you. Well, with rising number of requests for blockchain expertise in the future, companies are willing to pay a premium for skilled individuals. Picture this – higher salaries, more benefits, and exciting new projects.

[https://www.geeksforgeeks.org/what-is-blockchain-ecosystem/]

Moreover, blockchain is not just a technology; each blockchain application comes with an ecosystem. Smart contracts, decentralized applications, and token economies are becoming mainstream. As a programmer, embracing blockchain opens up a world of opportunities beyond traditional coding. Entrepreneurs, programmers – understanding blockchain is not just an option; it's a strategic necessity. The economy is evolving, and those who grasp the power of blockchain will lead the way.

Lecture 5 – Course Outline

Download Course Outline

Lecture 6 – Objectives: Executing Transactions, Mining, Wallet, Verification of Transactions

What you will achieve in this course is the programming of a

- simple web wallet and

- a basic blockchain node

For that you will use the GO programming language.

The final demonstration will include

- transferring values with a transaction from one user's wallet to a second user's wallet,
- confirming transactions by mining through the involved nodes
- implementation of a competitive proof of work algorithm,
- proper creating and validating transactions,
- storing the transactions in blocks,
- distributing a blockchain to each node,
- achieving consensus,

and much more.

Lecture 7 – Prerequisites for This Course (An IDE, GO, Compiling Code, etc)

To follow the course and to benefit from the coding experience you need to have Google's programming language GO installed on your computer and be ready to compile and execute code. To install GO follow the installation instructions from their website matching your operating system and architecture.

Highly recommended is an IDE supporting Golang like

- GoLand from JetBrains which you have to pay for, or
- you can setup your own IDE around your favorite text editor like Atom or Notepad++,
- or you just rely on Visual Studio Code with the GO extensions installed as I do in this course and which is completely free of charge.

It doesn't matter if you are on Windows Machine, or macOS or on Linux, you need to have access to some text shell like the command prompt or PowerShell on Windows, or the bash in a terminal window on both macOS and Linux.

It may be an advantage to have a free github account to setup a repository for your experiments to benefit from an online version control system. In that case you may also want to have git, a distributed version control system, installed on your local computer as well.

To download the releases from the course accompanying repository neither a github account nor git installed is mandatory.

Please note that I cannot provide support with the installation, instructions or setup of the required software mentioned here or used during the course. Please visit the websites of the corresponding providers for assistance.

Links

- GO download

- GO installation instructions
- GoLand - JetBrains IDE (charged)
- Visual Studio Code (free)
- github
- git download

Section 2 – What Is Blockchain and What Can One Do With It?

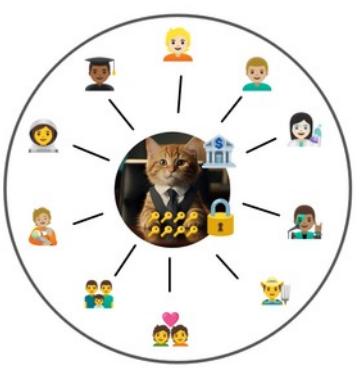
> *It is a global distributed database, with additions to the database by consent of the majority.*
>
> *Satoshi Nakamoto*

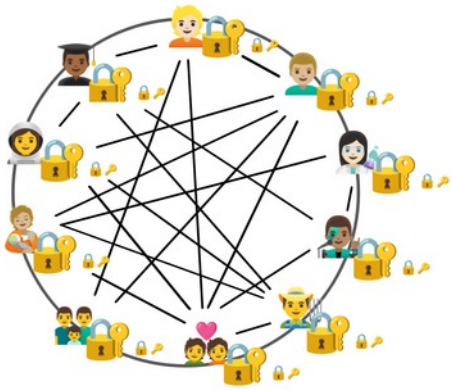
Lecture 8 – What is blockchain and what are the differences to a trivial database

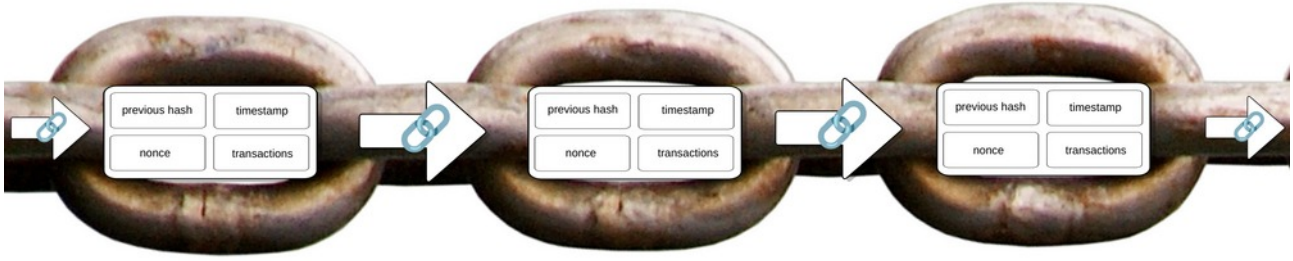
The TL;DR; is simple: a blockchain is a chain of blocks containing information.

But this definition is not enough to justify a course like this. You have to take a look at a few of the blockchain's own features.

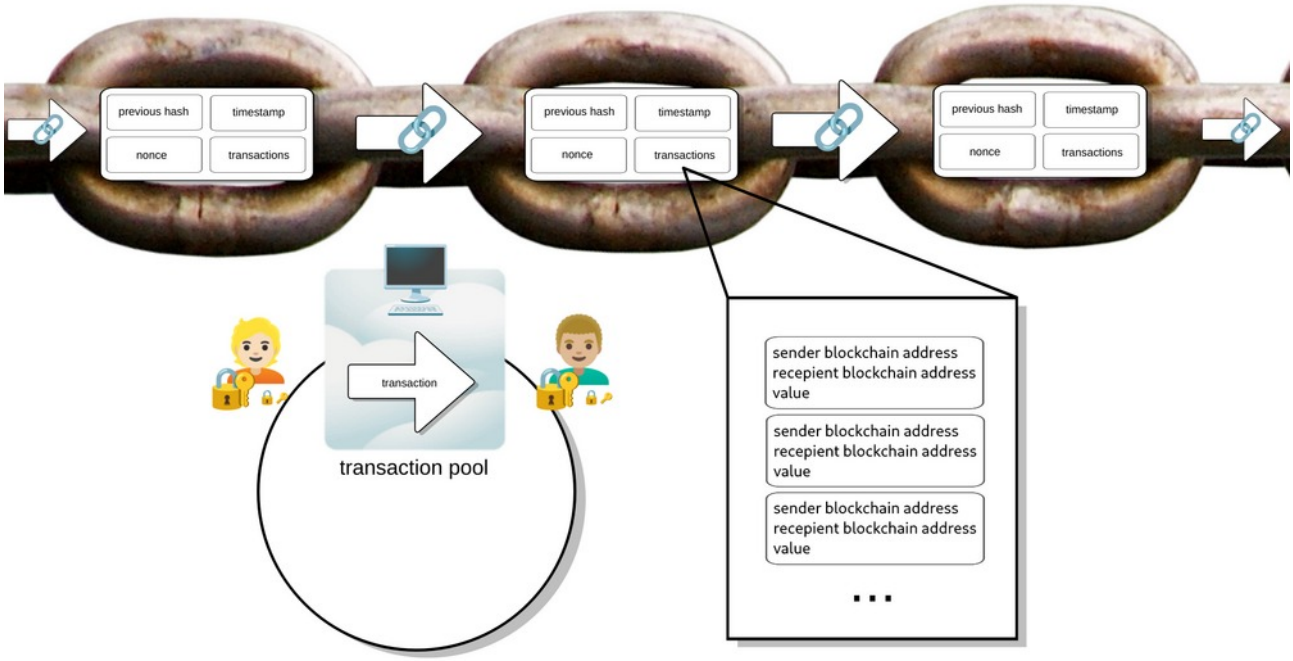
Blockchain is a decentralized and distributed ledger technology that securely records and verifies transactions across a network of computers. Um, digest that!







Unlike a traditional database, which is centralized and controlled by a single authority, a blockchain is operating on a peer-to-peer network, allowing multiple participants to have a copy of the entire ledger. Each transaction, bundled into a block, is cryptographically linked to the previous one, forming an unalterable and immutable chain.



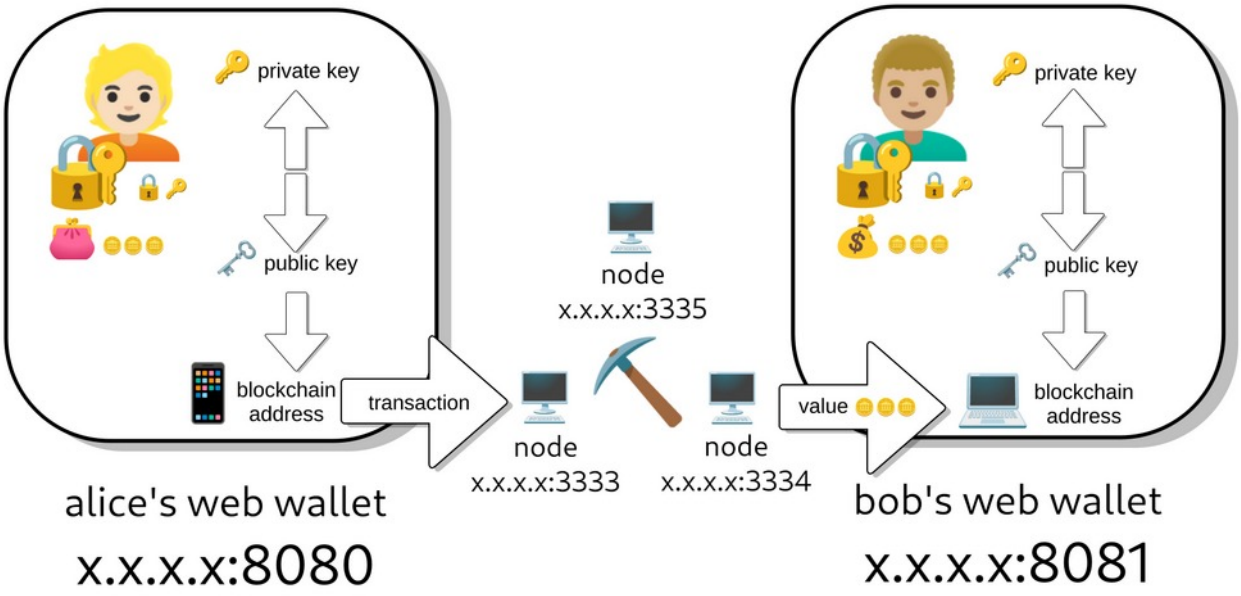
The key differences from a trivial database lie in

- decentralization,
- transparency, and
- immutability.

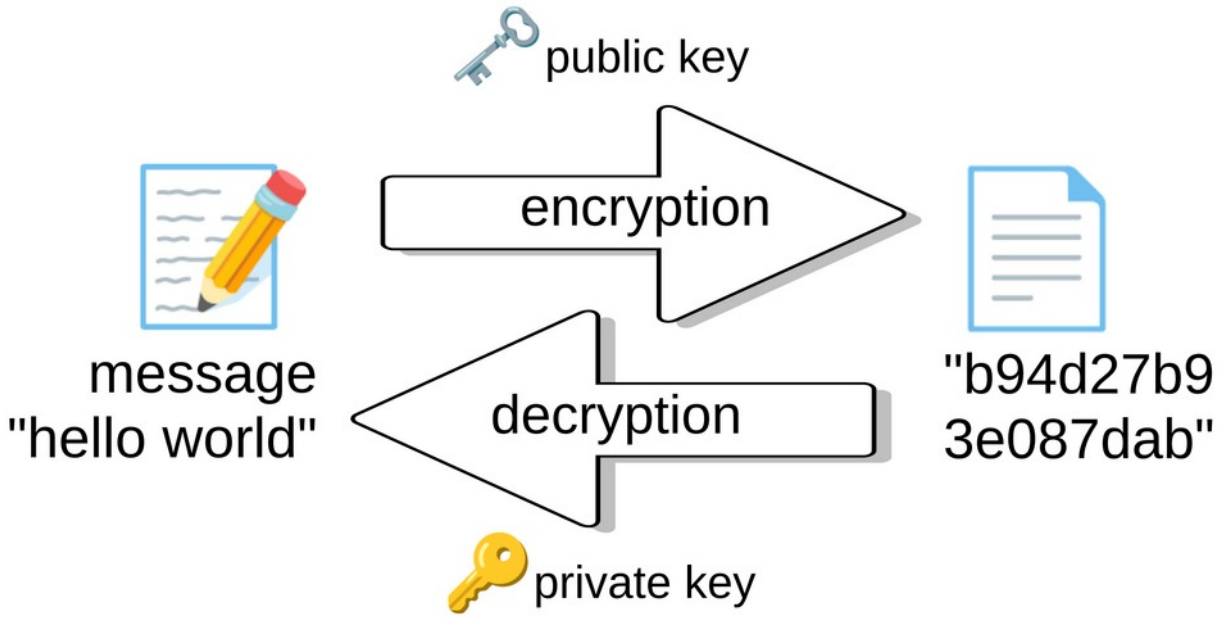
In a blockchain, no single entity has control, eliminating the need for a central authority. Transparency is achieved through a shared ledger visible to all participants, enhancing trust. Immutability ensures that once a block is added, it cannot be changed or deleted, providing a secure and tamper-resistant record.

In contrast, a conventional database relies on a central authority for control, lacks the inherent transparency of a blockchain, and can be vulnerable to unauthorized alterations. The decentralized, transparent, and immutable nature of blockchain makes it a perfect match for applications extending far beyond the realm of finance, into areas like supply chain management, voting systems, and more.

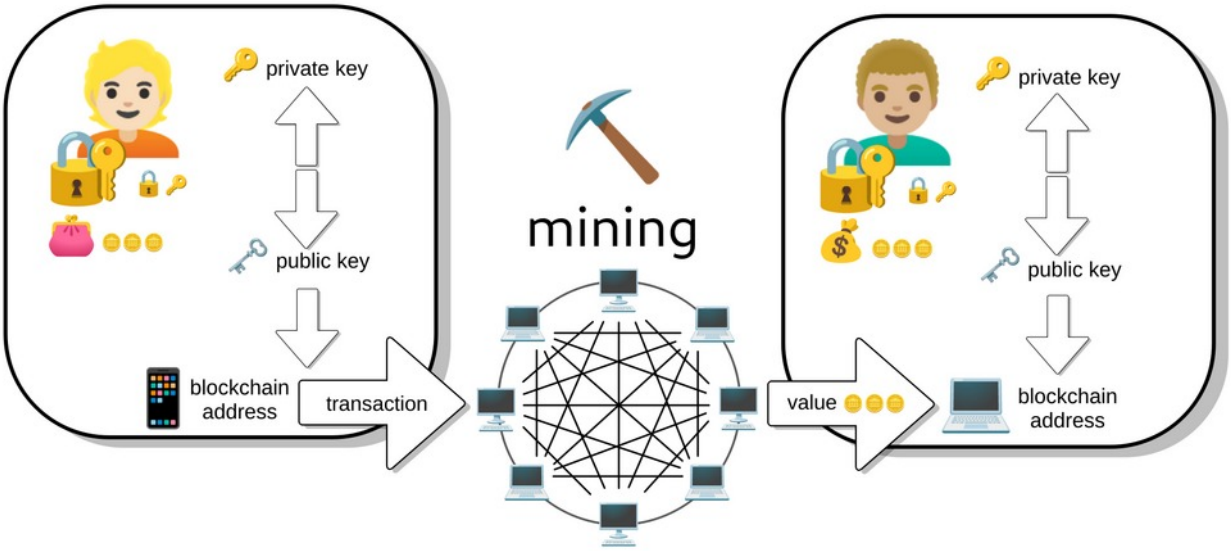
Lecture 9 – The Top Use Case: Blockchain as the Platform for Processing Payment Transactions



Blockchain as most people know it serves as a simple platform for processing payment transactions. In this use case, blockchain ensures secure, transparent, and swift transactions without the need for intermediaries like banks. Smart contracts, self-executing agreements with predefined rules, automate payment processes, reducing delays and human errors.



Blockchain's decentralized nature eliminates the risk of a single point of failure and enhances security through cryptographic validation. This use case offers borderless transactions, fostering financial inclusion. Additionally, blockchain's transparency ensures that all transaction details are visible and traceable, boosting trust in the payment ecosystem.



Overall, utilizing blockchain as a payment transaction platform streamlines processes, enhances security, and promotes a global, decentralized financial system. This transformative use case extends beyond cryptocurrencies, influencing the broader financial industry and paving the way for innovative approaches to payment processing.

Lecture 10 – Other Types of Blockchain and Other Possible Use Cases

Blockchains can be broadly categorized based on several criteria, including their access permissions, consensus mechanisms, and purpose. Here are some of the key types of blockchains:

1. **Public Blockchains:**
    - **Permissionless:** Anyone can participate, validate transactions, and join the network. Examples include Bitcoin and Ethereum.

2. **Private Blockchains:**
    - **Permissioned:** Access to the network is restricted, and participants are usually known entities. Suitable for enterprise applications, these blockchains offer more control and privacy.

3. **Consortium Blockchains:**
    - **Semi-Permissioned:** Shared control among a group of organizations. Useful for industries where collaboration is necessary but complete decentralization is not feasible.

4. **Hybrid Blockchains:**
    - Combine elements of public and private blockchains, offering flexibility and scalability. Certain parts of the blockchain may be public, while others are private.

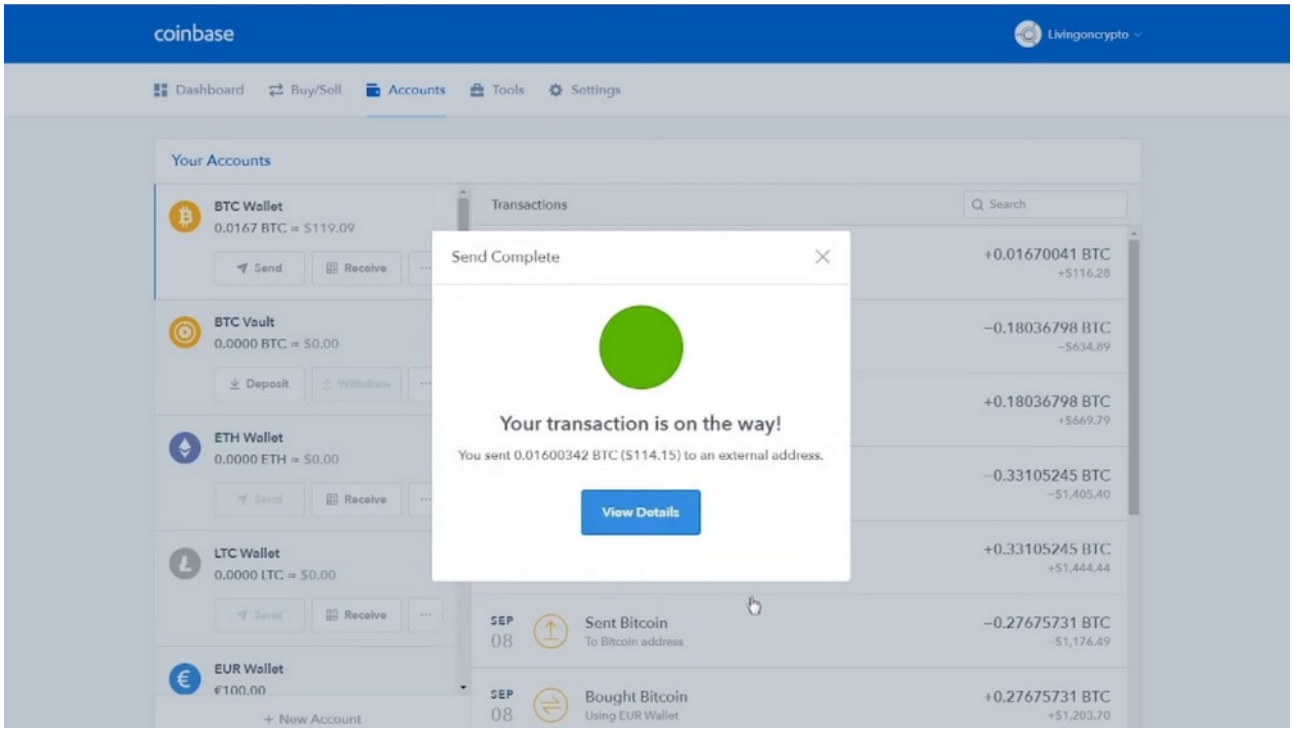
5. **Based on Consensus Mechanism:**

- **Proof of Work (PoW):** Requires participants (miners) to solve complex mathematical problems to validate transactions. Energy-intensive but secure (e.g., Bitcoin).
- **Proof of Stake (PoS):** Validators are chosen based on the amount of cryptocurrency they hold and are willing to "stake" as collateral. More energy-efficient than PoW.
- **Delegated Proof of Stake (DPoS):** Token holders vote for a limited number of delegates to validate transactions. Enhances scalability and efficiency (e.g., EOS).
- **Proof of Authority (PoA):** Validators are chosen based on their reputation and authority. Suitable for private or consortium blockchains.

6. **Purpose-Based Blockchains:**
    - **Smart Contract Platforms:** Designed to execute programmable contracts automatically (e.g., Ethereum, Binance Smart Chain).
    - **Supply Chain Blockchains:** Focus on enhancing transparency and traceability in supply chain management (e.g., VeChain).
    - **Identity Blockchains:** Aim to secure and manage digital identities (e.g., Sovrin).

These categories showcase the diversity of blockchains, each tailored to specific use cases and requirements. As the technology evolves, new hybrid models and consensus mechanisms continue to emerge, further expanding the blockchain landscape.

Lecture 11 – Example of a Transfer of Value as a Transaction on the Bitcoin Network



Please see how a real transfer of cryptocurrency from one wallet to another as an ispiration what this course is aiming at to teach you to build on your own.

Lecture 12 – Definitions of Terms in the Realm of Blockchain: Value, Hash, Timestamp, etc

For those of you who are not familiar with the terminology of blockchain and cryptocurrencies, I have provided a small unsorted glossary of terms used throughout this course in the course overview.

This PDF with the course overview is part of the course and I recommend that you keep the PDF to hand throughout the course. Please think of our environment and refrain from printing out the PDF. Thank you!

1. **Address:**

A Bitcoin address is a representation of a user's public key and is used for receiving funds. It is a unique string of characters that identifies the destination for Bitcoin transactions.

2. **Blockchain:**

The blockchain is a decentralized, distributed ledger that records all transactions made with Bitcoin. It consists of a chain of blocks, where each block contains a list of transactions. The blockchain is maintained by a network of nodes, ensuring transparency, security, and immutability.

Understanding these terms is essential for navigating the world of blockchain technology, whether you're a user, investor, or developer.

3. **Consensus:**

Consensus in the Bitcoin network refers to the agreement among nodes on the validity of transactions and the order of blocks in the blockchain. Bitcoin uses a consensus mechanism called Proof of Work to achieve this agreement. Miners compete to solve complex mathematical puzzles to add new blocks, and the longest valid chain is considered the true blockchain.

4. **Hash:**

A hash in Bitcoin is a cryptographic function that takes an input (data) and produces a fixed-size string of characters, which is a unique identifier for that input. Hashes are crucial for data integrity and security in the blockchain.

5. **Private Key:**

The private key is a secret, alphanumeric string known only to the owner. It is used to sign transactions and access the Bitcoin associated with a specific address. Protecting the private key is crucial for the security of one's Bitcoin holdings.

6. **Public Key:**

The public key is derived from the private key and serves as an address to which others can send Bitcoin. It is openly shared and associated with the owner's identity. However, it's computationally infeasible to reverse-engineer the private key from the public key.

7. **Signature:**

A digital signature is created using the private key and attached to a transaction. It provides cryptographic proof that the transaction was authorized by the rightful owner of the bitcoins.

8. **Timestamp:**

Timestamp in blockchain is a record of the date and time when a particular transaction is added to the blockchain. It helps establish the chronological order of transactions.

9. **Transactions:**

Transactions in Bitcoin involve the transfer of value from one Bitcoin address to another. They are recorded on the blockchain and include details like sender, receiver, amount, and cryptographic signatures for security.

10. **Validation:**

Validation in Bitcoin refers to the process by which transactions are verified and added to the blockchain. It involves consensus mechanisms, such as proof of work, where miners compete to solve complex mathematical problems to validate transactions.

11. **Value:**

In the context of Bitcoin, "value" refers to the amount of the cryptocurrency associated with a specific transaction or address. It represents the economic worth of the Bitcoin being transferred.

Section 3 – Build a Blockchain: The Easiest Way to Understand a Blockchain Is to Create One

> *For the things we have to learn before we can do them, we learn by doing them.*
>
> *Aristotle*

Lecture 13 – What You Need to Get Started Besides Your Favorite Soft Drink

If you want to make use of github to store your experiments in a repository consider to setup a github account and login to that account. Create a new repository with the following settings:

- public,
- add readme,
- add gitignore file for go
- add licence

[Change gitignore file content!]

After the repository is created you will be able to clone the repository, basically to copy it to your local computer by selecting the https or ssh link.

In the desired working directory you execute

```
git clone git@github.com:jagottsicher/myGoBlockchain.git
```

Let’s test if we can access the repository by creating a file, adding it to the versioning, write a commit and push the whole repository back to the server:

```
cd myGoBlockchain
touch test.txt
git add .
git commit -m “just an initial push”
git push origin main
```

Please be aware that you need to deploy a SSH key properly to access the ssh link for a push,

You need to create ssh keys locally on your computer and deploy it by clicking in the upper right corner of the github website on your

account icon → settings → left side menu approx middle:→ SSH and PGP keys → button right upper corner: New SSH key

to deploy a new ssh keys. Further help can be found on https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

If you want to clone your repository by using the https link, because you are behind a strict firewall or accessing github through a proxy you need to provide a so called access token instead of a password. That is because since 2021 for security reasons it is no longer possible to access your repository with your login credentials directly.

You can create a general personal access token by clicking in the upper right corner of the github website on your

account icon → settings → left side menu bottom:→ developer settings → Personal access tokens → Tokens (classic) – button right upper corner: Generate new token → Generate new token (classic)

After authenticating your identity a personal access token can be generated, which is basically a text string and serves as a password.

Please see https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for first clues to solve access and permission issues.

If you have problems to commit and push your work to github please contact https://docs.github.com/en for advice and assistance. This is not part of this course.

Now that your repository is setup and you made sure you can commit your work and push it to github you can delete the `test.txt` file if you are still in the directory.

```
rm test.txt
```

First you want to initialize your go project to use a feature called go modules which will take care about the dependencies or your code. It is not only good practice but highly recommended to use it.

```
go mod init github.com/jagottsicher/myGoBlockchain
```

To be clear here as domain you can setup whatever you like but for importing dependencies later and for an easy workflow for me as an instructor I use the path to my github repository here. At this point I can start my IDE directly within this folder here. If you are on a windows machine you may start Visual Basic Code first and then choose “open folder” from the “file” menu. I do

```
code .
```

on the terminal.

You need to create a new file like `blockchain.go` and can start editing. You start with a simple hello-world-file like this:

```
package main

import "fmt"

func main() {
     fmt.Println("Hello world")
}
```

If you setup Visual Studio Code properly you can open a local terminal of your choice from the terminal menu within Visual Studio Code. If you prefer an external terminal window you can also go that way.

My command prompt here shows the path I am in and here I can compile and run my code directly with typing

```
go run .
```

You see the code is executed and the output “hello world” is written to the terminal.

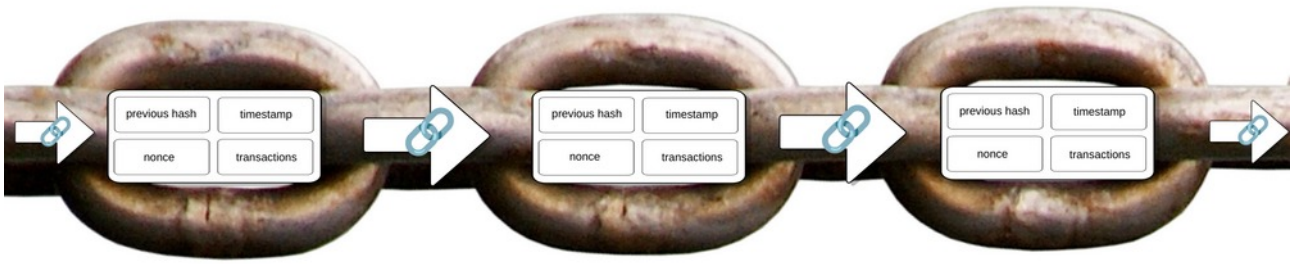
Now I will once push this to github. Please be aware that this is something you have to decide for your project, when it is a good time to push your newest versions to github. In the next lectures I will do that from time to time off-screen to provide you with different states of the code presented here.

```
git add .
git commit -m “added a hello-world-like first draft”
git push origin main
```

And now it's time to take your hands out of your pockets, release the handbrake and program a blockchain. Get started in the next lesson!

Code Lecture 13

Lecture 14 – Create a Block! And Satoshi Said: “Let There Be Block!”



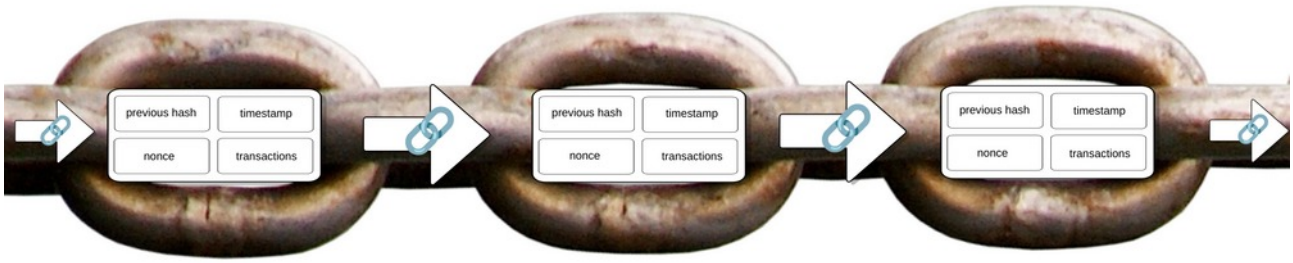

```
package main

import (
     "fmt"
     "log"
     "time"
)

type Block struct {
     nonce        int
     previousHash string
     timestamp    int64
     transactions []string
}

func NewBlock(nonce int, previousHash string) *Block {
     b := new(Block)
     b.timestamp = time.Now().UnixNano()
     b.nonce = nonce
     b.previousHash = previousHash
     return b
}

func (b *Block) Print() {
     fmt.Printf("timestamp       %d\n", b.timestamp)
     fmt.Printf("nonce           %d\n", b.nonce)
     fmt.Printf("previous_hash   %s\n", b.previousHash)
     fmt.Printf("transactions    %s\n", b.transactions)
```

```
}

func init() {
     log.SetPrefix("Blockchain Node: ")
}

func main() {
     b := NewBlock(0, "init hash")
     b.Print()
}
```

In this lesson, you created a struct with the elements of the data types required to create a block for a blockchain.

Code Lecture 14

Lecture 15 – Create a Blockchain Struct. Give Your Blocks a Structure!



Now you will learn how to chain these blocks into a blockchain.

[ADD TO BLOCKCHAIN.GO]

```
type Blockchain struct {
     transactionPool []string
     chain           []*Block
}

func NewBlockchain() *Blockchain {
```

```
	bc := new(Blockchain)
	bc.CreateBlock(0, "hash #0 genesis block")
	return bc
}

func (bc *Blockchain) CreateBlock(nonce int, previousHash string)
*Block {
	b := NewBlock(nonce, previousHash)
	bc.chain = append(bc.chain, b)
	return b
}

func (bc *Blockchain) Print() {
	for i, block := range bc.chain {
		fmt.Printf("%s Block %d %s\n", strings.Repeat("=", 10),
i, strings.Repeat("=", 10))
		block.Print()
	}
	fmt.Printf("%s\n", strings.Repeat("#", 27))
}

// func main() {
//   blockChain := NewBlockchain()
//   fmt.Println(Blockchain)
// }

func main() {
	blockChain := NewBlockchain()
	blockChain.Print(Blockchain)
	blockChain.CreateBlock(23, "hash #1")
	blockChain.Print()
	blockChain.CreateBlock(42, "hash #2")
	blockChain.Print()
}
```

You have successfully started a blockchain and created three blocks already. This is all still quite improvised and if you want to connect the blocks with each other, you have to create hash values and use these as connectors between the blocks.

Code Lecture 15

Lecture 16 – How to Generate the Hash Value of a Block

At the moment, your hash values are basically random strings. Before you write the code now, you should realize a few things about creating hash values. Take a look in GO's standard library. You can find the standard library on

[https://pkg.go.dev/std]

You probably know that a widely tested and secure algorithm called sha256 is often used to create hash values in blockchains.

You may be looking in the `hash` directory, but you won't find it there. sha256 is an encryption algorithm invented by the NSA. The implementation in GO was in the development branch x, for experimental, for a long time. Nowadays you can now find the sha256 package in the `crypto` directory.

[RUN CRYPTO/SHA256/FUNCTIONS/SUM256 EXAMPLES]

You will quickly find what you are looking for in the functions. There resides a function called Sum256, and if you look at an example application here, you will see that this function provides exactly what we will need next.

The great thing about the website here is that you can run the examples immediately, and even edit them and then run them.

[RUN CRYPTO/SHA256/FUNCTIONS/SUM256 EXAMPLES EDITED]

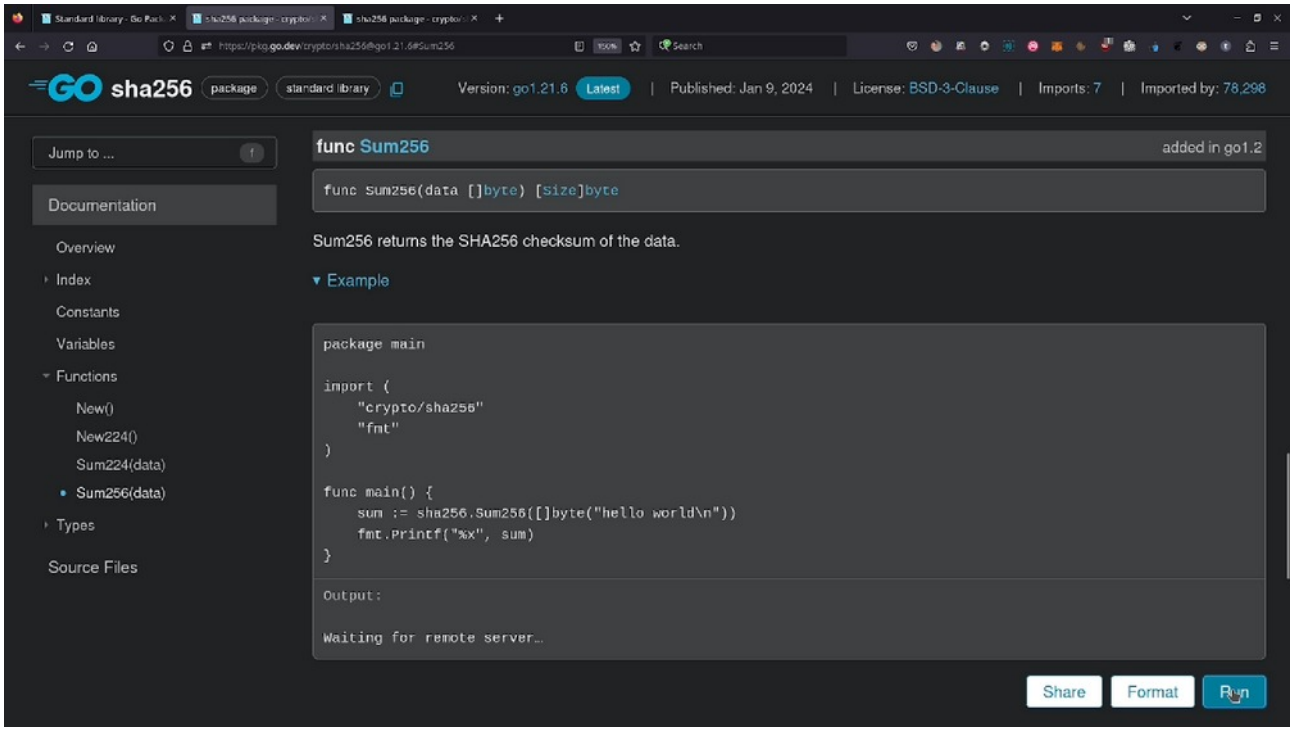
But, what is returned here is basically a slice of byte. You can easily check this by adjusting the output here:

```
fmt.Printf("%v\n", sum)
fmt.Printf("%x\n", sum)
```

Please take a look at this.

```
sum := sha256.Sum256([]byte("Litte Green Man\n"))
fmt.Printf("%x\n", sum)
sum2 := sha256.Sum256([]byte("Litte Green Man\n"))
fmt.Printf("%x\n", sum2)
```

The crucial factor with this type of encryption is that the same input values always provide the same output values. However, it is not possible to deduce the original input values from the output values.



Conversely, this means that if the data has been changed, a different hash value exists for it. This is the great benefit for the blockchain: without having to check two complete data sets of input data, it is possible to determine whether the data was exactly the same by comparing two hash values.

You make use of this as follows: You create a hash value over the data of each block and include the result as part of the next block. This is the goal you want to achieve in the short term.

You just need to write a method that returns a hash value. Our hash value has a width of 32 bytes and you can implement it as a method with receiver type pointer to Block.

```
func (b *Block) Hash() [32]byte {
     m, _ := json.Marshal(b)
     fmt.Println(m)
     return sha256.Sum256([]byte(m))
}
```

Your hash value has a width of 32 bytes and this is exactly what your hash method should return for a block. Your input data will be exactly this block, but you will convert the data into JavaScript object notation, JSON, and create the hash value of the JSON object after converting the JSON into a slice of byte as expected by function Sum256.

Note to all of you coming from other Programming langugaes: In GO, casting is referred to as conversion. Forget the wizardry of spellcasting or casting for Hollywood movies – GO programmers, gophers, use built-in functions to convert data types. No magic, just explicit

conversions. Unlike some languages, there's no implicit casting in GO. So, let's call it what it is – a conversion.

```
func main() {
     // blockChain := NewBlockchain()
     // blockChain.Print(Blockchain)
     // blockChain.CreateBlock(23, "hash #1")
     // blockChain.Print()
     // blockChain.CreateBlock(42, "hash #2")
     // blockChain.Print()

     block := &Block{nonce: 1}
     fmt.Printf("%x\n", block.Hash())
}
```

[RUN]

As you can see, you can easily create a unique sha256 hash value for a block, which in this case only contains a single value, nonce equals one. Works smoothly though.

Perhaps it will become clearer what is happening here. To do this, adjust the output in the Hash method to convert m slice of bytes to type string.

```
     fmt.Println(string(m))
```

[RUN]

Now you can also implement a new method to marshal JSON of a block.

```
func (b *Block) MarshalJSON() ([]byte, error) {
     return json.Marshal(struct {
          Timestamp    int64    `json:"timestamp"`
          Nonce        int      `json:"nonce"`
          PreviousHash string   `json:"previous_hash"`
          Transactions []string `json:"transactions"`
     }{
          Timestamp:    b.timestamp,
          Nonce:        b.nonce,
          PreviousHash: b.previousHash,
          Transactions: b.transactions,
```

```
    })
```

[RUN]

You may also have noticed that a hash value is not actually of the string type. It is a 32 byte long slice of byte. You should take this into account and adjust the data type in the struct.

```
type Block struct {
    nonce        int
    previousHash [32]byte
    timestamp    int64
    transactions []string
}
```

and here

```
func NewBlock(nonce int, previousHash [32]byte) *Block {
```

and here as well

```
func (b *Block) MarshalJSON() ([]byte, error) {
    return json.Marshal(struct {
        Timestamp    int64    `json:"timestamp"`
        Nonce        int      `json:"nonce"`
        PreviousHash [32]byte `json:"previous_hash"`
        Transactions []string `json:"transactions"`
    }{
    ...
```

and then here:

```
func (b *Block) Print() {
    fmt.Printf("timestamp       %d\n", b.timestamp)
    fmt.Printf("nonce           %d\n", b.nonce)
```

```
    fmt.Printf("previous_hash   %x\n", b.previousHash)
    fmt.Printf("transactions    %s\n", b.transactions)
}
```

and here again:

```
func (bc *Blockchain) CreateBlock(nonce int, previousHash
[32]byte) *Block {
```

and finally here

```
func NewBlockchain() *Blockchain {
    b := &Block{}
    bc := new(Blockchain)
    bc.CreateBlock(0, b.Hash())
    return bc
}
```

If you have the feeling that this all went a bit fast or you may have made a mistake, take a look at the source code associated with this lesson. Basically, the underlying data type of an element of the block structs was changed and this required the adaptation of all methods that somehow refer to this instance.

GO is a strictly typing programming language and that's a good thing. Your IDE will help you to find the necessary changes, respectively Visual Studio Code as well as Goland by JetBrains can also rename complete instances and change all references.

You always want to add one block after the other to the blockchain by including the hash from the previous block in the current block. To do this, you now write a method to determine the last block, basically the block that was last appended to the blockchain and that precedes the current block.

```
func (bc *Blockchain) LastBlock() *Block {
    return bc.chain[len(bc.chain)-1]
}
```

The great thing is that our blockchain is a slice of pointers to `Blocks`. The length, which you can determine with the `len()` function, reflects the current number of elements in the slice. You only

need to subtract one from this and you can return the pointer to the last block before it. It's that straightforward.

Let’s check, if your method returns the expected values

```
func main() {
	// blockChain := NewBlockchain()
	// blockChain.Print(Blockchain)
	// blockChain.CreateBlock(23, "hash #1")
	// blockChain.Print()
	// blockChain.CreateBlock(42, "hash #2")
	// blockChain.Print()

	// block := &Block{nonce: 1}
	// fmt.Printf("%x\n", block.Hash)

	blockChain := NewBlockchain()
	blockChain.Print()

	previousHash := blockChain.LastBlock().Hash()
	blockChain.CreateBlock(23, previousHash)
	blockChain.Print()

	previousHash = blockChain.LastBlock().Hash()
	blockChain.CreateBlock(42, previousHash)
	blockChain.Print()
}
```

[RUN]

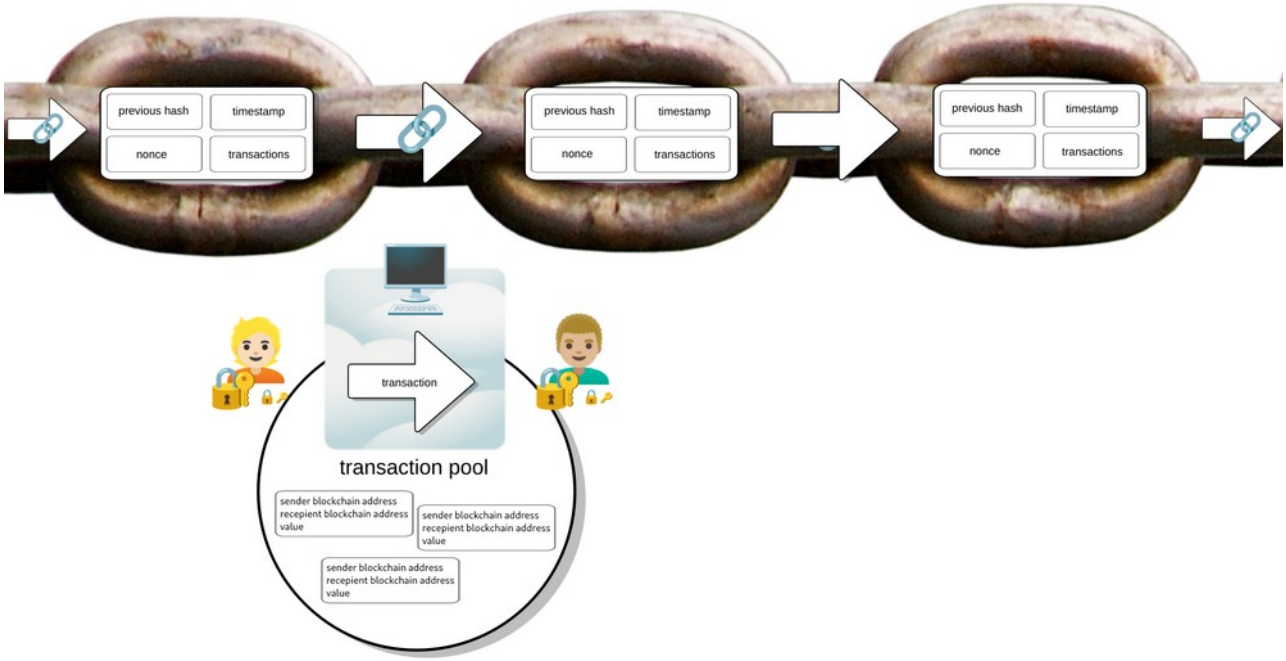
As you can see from the output, your code seems to work. You have successfully created a blockchain and connected the three existing blocks by creating a hash value of an object in JSON of the previous block.

If you are now confused, need further review of the code or your own code does not work, download the source code using the link to the current release of this lesson from Github. Try to understand how the code works.

Code Lecture 16

Lecture 17 – Add Transactions - The Payload Data in the Blocks of the Blockchain

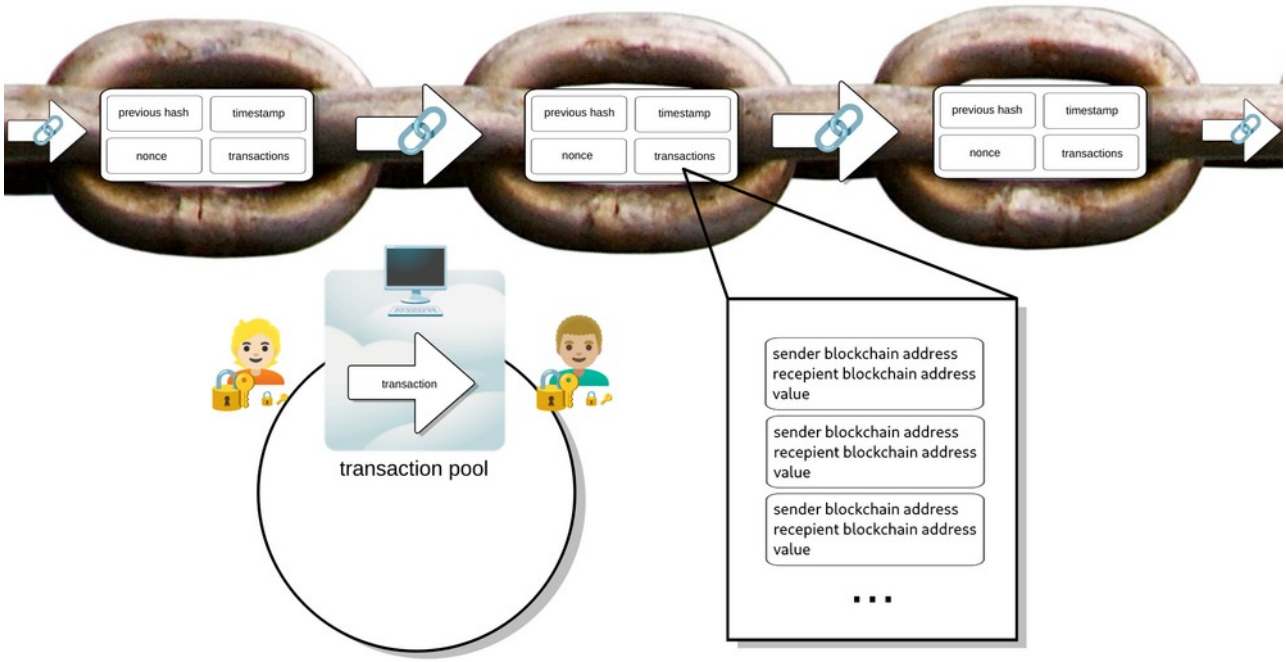
Each block of your blockchain contain four elements by now:



1. the hash value of the previous block,
2. a timestamp
3. a nonce, you shall think about of as a solution to a riddle, and
4. a slice of transaction

where each of the transactions of that slice consists of three elements so far:

1. the sender's blockchain address
2. the recipient's blockchain address
3. a value of type float32



It is time to implement transaction handling into your code. At the end of this lecture the blocks of your blockchain will be able to to take in and process transactions if there are some available in some kind of pool when it comes to block building.

[BEFORE FUNC INIT()]

```
type Transaction struct {
     senderBlockchainAddress    string
     recipientBlockchainAddress string
     value                      float32
}

func NewTransaction(sender string, recipient string, value
float32) *Transaction {
     return &Transaction{sender, recipient, value}
}

func (t *Transaction) Print() {
     fmt.Printf("%s\n", strings.Repeat("-", 40))
     fmt.Printf(" sender_blockchain_address      %s\n",
t.senderBlockchainAddress)
     fmt.Printf(" recipient_blockchain_address   %s\n",
t.recipientBlockchainAddress)
```

```
	fmt.Printf(" value                          %.1f\n", t.value)
}

func (t *Transaction) MarshalJSON() ([]byte, error) {
	return json.Marshal(struct {
		Sender    string  `json:"sender_blockchain_address"`
		Recipient string  `json:"recipient_blockchain_address"`
		Value     float32 `json:"value"`
	}{
		Sender:    t.senderBlockchainAddress,
		Recipient: t.recipientBlockchainAddress,
		Value:     t.value,
	})
}
```

[TO TOP EDIT]

```
type Block struct {
	nonce        int
	previousHash string
	timestamp    int64
	transactions []*Transaction
}

func NewBlock(nonce int, previousHash [32]byte, transactions
[]*Transaction) *Block {
b := new(Block)
	b.timestamp = time.Now().UnixNano()
	b.nonce = nonce
	b.previousHash = previousHash
	b.transactions = transactions
	return b
}
```

```
func (b *Block) Print() {
     fmt.Printf("timestamp       %d\n", b.timestamp)
     fmt.Printf("nonce           %d\n", b.nonce)
     fmt.Printf("previous_hash   %x\n", b.previousHash)
     for _, t := range b.transactions {
          t.Print()
     }
}

func (b *Block) MarshalJSON() ([]byte, error) {
     return json.Marshal(struct {
          Timestamp    int64    `json:"timestamp"`
          Nonce        int      `json:"nonce"`
          PreviousHash string   `json:"previous_hash"`
          Transactions []*Transaction `json:"transactions"`
     }{
          Timestamp:    b.timestamp,
          Nonce:        b.nonce,
          PreviousHash: b.previousHash,
          Transactions: b.transactions,
     })

type Blockchain struct {
     transactionPool []*Transaction
     chain           []*Block
}
```

The pool needs to be emptied after transactions are on transferred to the chain and stored in a block.

```
func (bc *Blockchain) CreateBlock(nonce int, previousHash string)
*Block {
     b := NewBlock(nonce, previousHash, bc.transactionPool)
     bc.chain = append(bc.chain, b)
     bc.Transactionpool = []*Transactions{}
```

```
    return b
}
```

Please see the new method `AddTransaction` to the Type `Blockchain` to add a transaction

```
func (bc *Blockchain) AddTransaction(sender string, recipient
string, value float32) bool {
    t := NewTransaction(sender, recipient, value)
    bc.transactionPool = append(bc.transactionPool, t)
    return false
}

func main() {
    // blockChain := NewBlockchain()
    // blockChain.Print(Blockchain)
    // blockChain.CreateBlock(23, "hash #1")
    // blockChain.Print()
    // blockChain.CreateBlock(42, "hash #2")
    // blockChain.Print()

    // block := &Block{nonce: 1}
    // fmt.Printf("%x\n", block.Hash)

    blockChain := NewBlockchain()
    blockChain.Print()

    blockChain.AddTransaction("Rick", "Morty", 137)
    previousHash := blockChain.LastBlock().Hash()
    blockChain.CreateBlock(23, previousHash)
    blockChain.Print()

    previousHash = blockChain.LastBlock().Hash()
    blockChain.CreateBlock(42, previousHash)
    blockChain.Print()
```

```
}
```

[RUN]

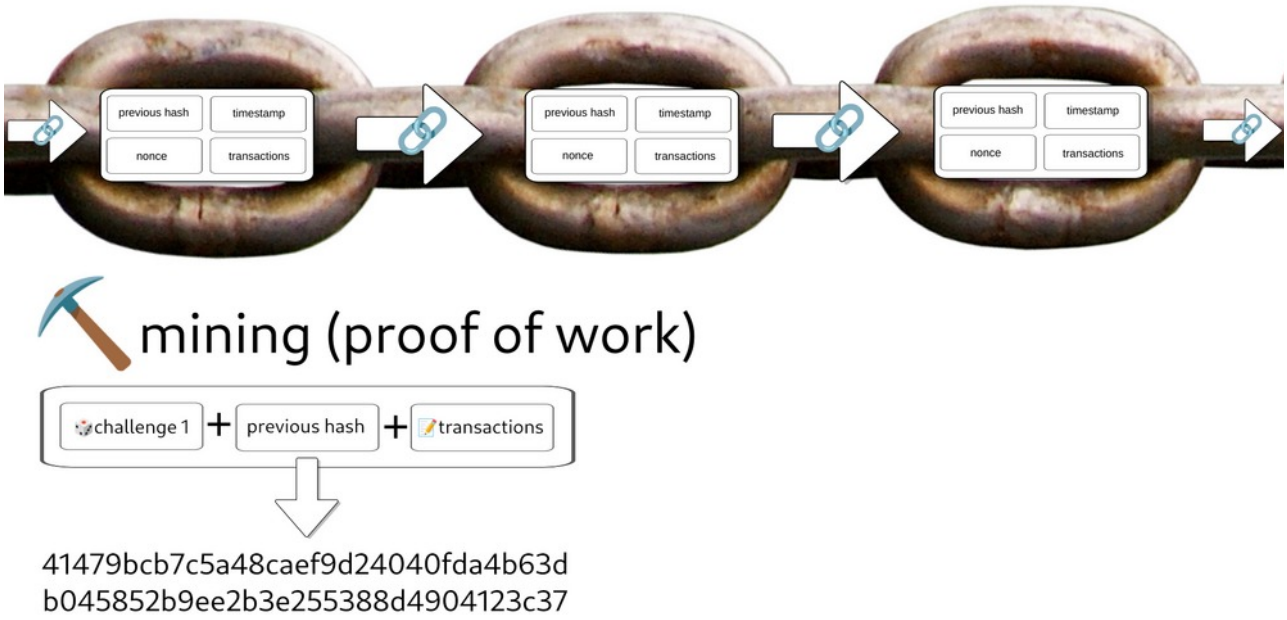
```
    blockchain.AddTransaction("Rick", "Morty", 137)
    previousHash := blockChain.LastBlock().Hash()
    blockChain.CreateBlock(23, previousHash)
    blockChain.Print()

    blockchain.AddTransaction("Stan", "Francine", 23)
    blockchain.AddTransaction("Fry", "Bender", 42)
    previousHash = blockChain.LastBlock().Hash()
    blockChain.CreateBlock(42, previousHash)
    blockChain.Print()
```

[RUN]

In this lecture you learned how to create transactions and place them in a transaction pool. From there they are taken, added to a block and removed from the pool, respectively the pool is flushed. If you have problems to understand what is going on please check out the source code for this lecture on Github.

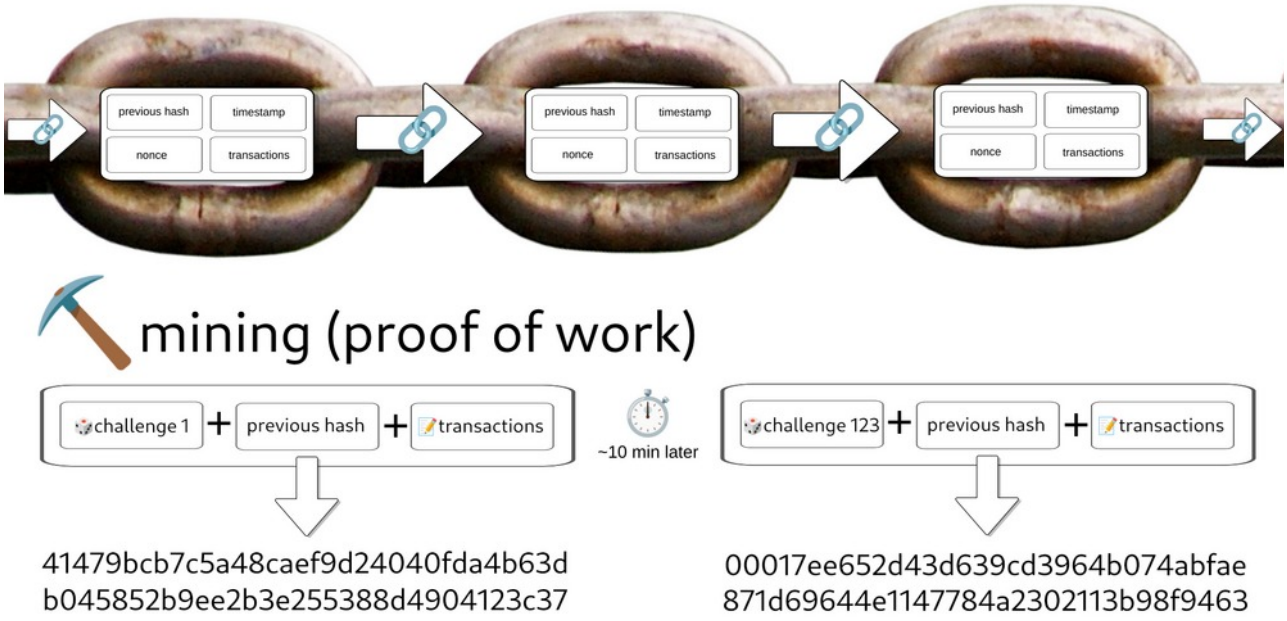
Code Lecture 17

Lecture 18 – Always Something New! What Are Proof Of Work, Mining, Difficulty and the Nonce?



A block on your blockchain already has a lot of nice features. You have a reasonable value as hash of the previous block, a timestamp of when the block was created and you can pack transactions into a block.

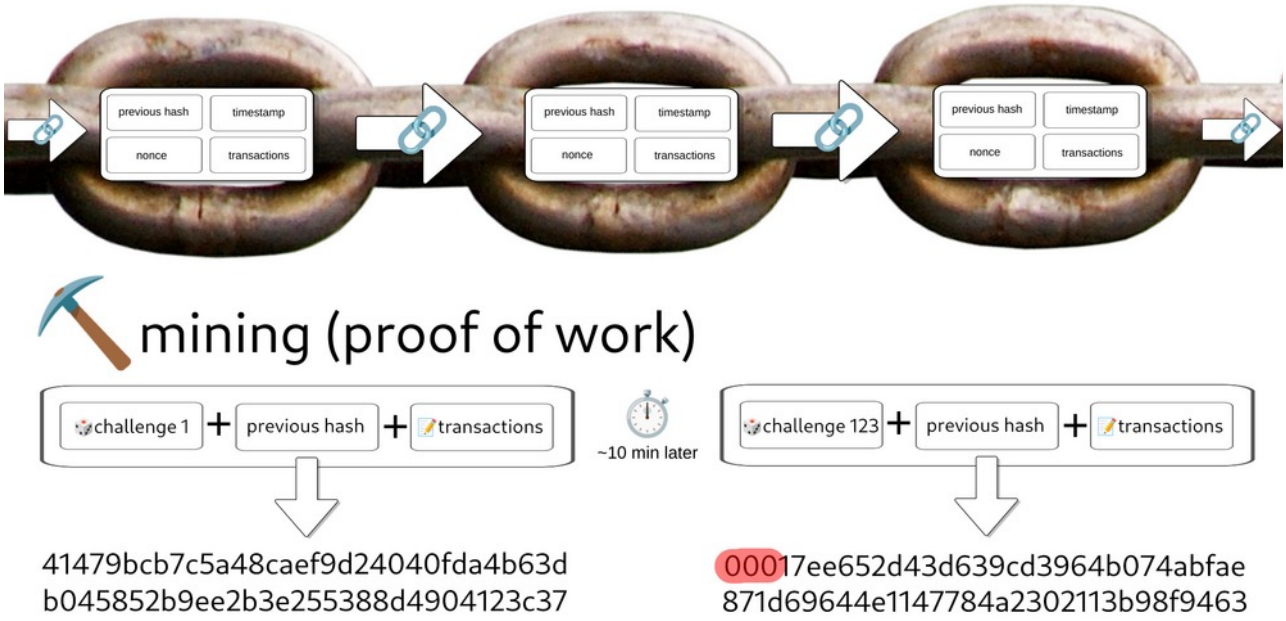
What is still missing is a meaningful value for nonce. A nonce is basically a value of some kind that is added to a cryptographic function once only, so to speak, in order to make it unique.



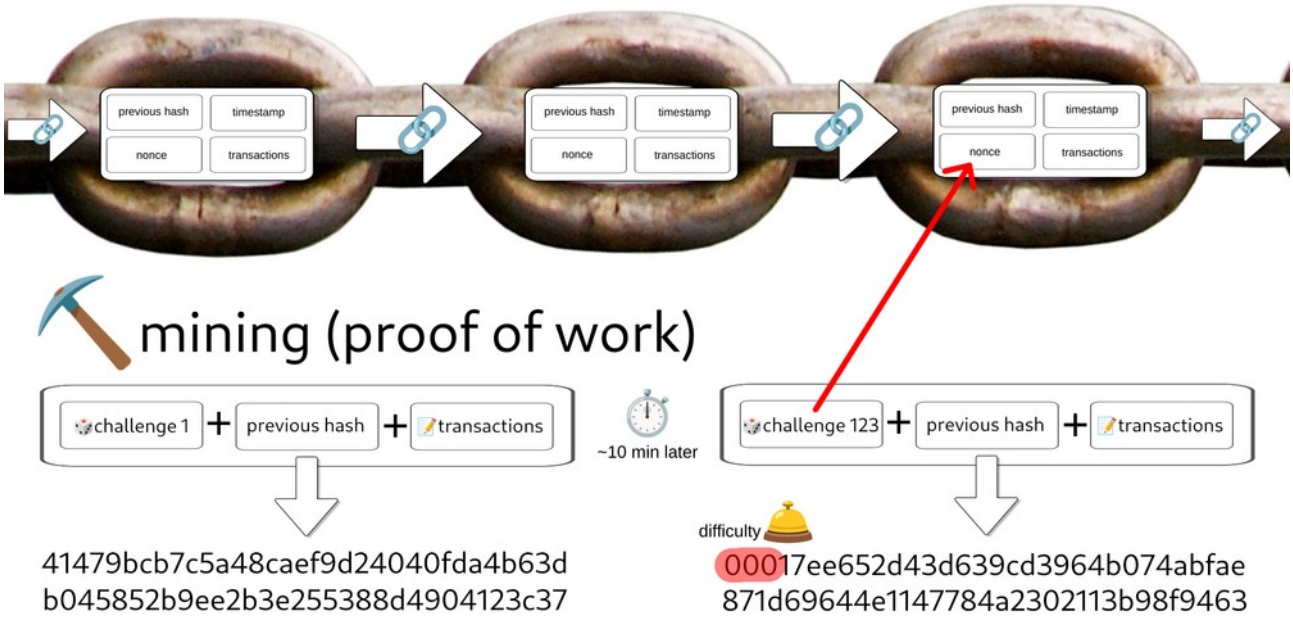
On the following slides the nonce during the process of finding the hash value is varied. As long the Block is not written you can call it a challenge.

Why is this used as a value in a block for the blockchain in the first time? This course is largely based on the blockchain as it is structured for Bitcoin. In Bitcoin, the creation of the blocks and thus the confirmation of the transaction is a competitive process among different Bitcoin nodes, which form blocks from a largely identical pool of transactions and create hash values from them.

If one of these so-called miners is the first to do this, he is rewarded with a previously agreed reward and by receiving the fees contained in the transactions. Each miner therefore tries to solve the task as quickly as possible and add their result to the blockchain. You may have heard this described as a mathematical riddle that has to be solved and requires a lot of computing power. This is only half the truth. cal riddle that has to be solved and requires a lot of computing power. This is only half the truth.



The real task is to create a hash value for the block and this is basically a matter of fractions of a second. However, the hash value must fulfill a condition that artificially increases the time to find one. Basically, the condition serves to ensure that not only the node with the most computing power or the best internet connection solving all blocks, but that the task is forced to be decentralized. The condition is linked to the number of leading zeros of the hash value.



You simply have to try out different hash values to find one matching that condition. The more zeros are demanded, the longer it takes to find a corresponding hash value, which is known as the difficulty. And in order to be able to vary this trial and error, you need a value that you can change constantly. This will become the nonce. Basically, from a value that is simply incremented. You could also choose it randomly, or rely on enough variation of transactions from a large pool and on different timestamps in nanoseconds. In this slide you see the challenge just reaching a value of 123 and with that fulfilling the condition.

Such a condition like the number of leading zeros to a hash value can control the average duration for block creation very finely. Bitcoin adjusts the difficulty every 2016 blocks (approximately every 14 days) and thus ensures that block creation takes around ten minutes despite a growing or fluctuating number of miners.

In our code though, you will set the difficulty fixed to three so that we can achieve a fast transaction and block creation rate withing seconds. We will simply simulate a stable duration by triggering block creation only every few seconds.

Lecture 19 – Where Does the Nonce Derive From? Don’t Get Upset, Count Slowly to ...

You are taking remarkably big steps towards a much misunderstood field of blockchain: mining. In one of the following lectures we will have a closer look, but for now let’s start with where the thing called nonce comes from and how we do something with it.

```
const MINING_DIFFICULTY = 3
```

A constant in Go is basically not of a defined type and during compilation is is decided and from the context where the constant is used the data type is decided.

```
func (b *Block) Hash() [32]byte {
```

```
    m, _ := json.Marshal(b)
    fmt.Println(string(m))
    return sha256.Sum256([]byte(m))
}
```

[BELOW ADDTRANSACTION()]

If we are going to start mining the first step is to take a copy of all transactions which are available in the transaction pool and use that copy to stuff them into a new block to test a lot of hashes of that block. In a real blockchain scenario the mining node would choose the most promising transactions only and prepare a block of these. After a block is confirmed the transactions can be deleted from the pool.

But in your little example code here it works some simplified. You don't make a pre-selection here and take just all available transactions. Later we flush the pool anyway if the transactions are stored in the blockchain.

Nevertheless, at this point you cannot just flush the transactions pool already. That was like juggling with eggs risking the eggs are lost. Instead you just make a copy of the complete transaction pool and work with that instead. That's really easy.

```
func (bc *Blockchain) CopyTransactionPool() []*Transaction {
    transactions := make([]*Transaction, 0)
    for _, t := range bc.transactionPool {
        transactions = append(transactions,
            NewTransaction(t.senderBlockchainAddress,
                t.recipientBlockchainAddress,
                t.value))
    }
    return transactions
}
```

Now you can start to implement the condition and return true if it is fulfilled. You need our difficulty constant and use that to check if the hash starts with the required number of zeros. Basically you test the validity of a hash of a potential block of all available transactions if the condition of having difficulty number of zeros in the front is true.

Note that there are probably much faster ways to check this than building a string from the hash value and doing some string-substring comparison but for the sake of easy understanding what's going on this is fair enough for your code.

```
func (bc *Blockchain) ValidProof(nonce int, previousHash [32]byte,
transactions []*Transaction, difficulty int) bool {
     zeros := strings.Repeat("0", difficulty)
     guessBlock := Block{0, nonce, previousHash, transactions}
     guessHashStr := fmt.Sprintf("%x", guessBlock.Hash())
     return guessHashStr[:difficulty] == zeros
}
```

Note that you must not have to put a valid timestamp in here. You had to update this all the time and after a block is found hashing the block would only work, if exactly the timestamp was used while creating it. But here you want your block validity to depend on the nonce only.

Now that you have a method proofing the validity of a block you need to proof that your work has been done with determinating the nonce for which the condition is true, so to say, what the nonce of the valid block was. This will contain the loop to vary the nonce until a valid block hash is found.

```
func (bc *Blockchain) ProofOfWork() int {
     transactions := bc.CopyTransactionPool()
     previousHash := bc.LastBlock().Hash()
     nonce := 0
     for !bc.ValidProof(nonce, previousHash, transactions,
MINING_DIFFICULTY) {
          nonce += 1
     }
     return nonce
}
```

Time to checkout if mining and our proof of work algorithm work as expected

```
func main() {
     blockChain := NewBlockchain()
     blockChain.Print()

     blockChain.AddTransaction("Rick", "Morty", 137)
     previousHash := blockChain.LastBlock().Hash()
     nonce := blockChain.ProofOfWork()
```

```
	blockChain.CreateBlock(nonce, previousHash)
	blockChain.Print()

	blockChain.AddTransaction("Stan", "Francine", 23)
	blockChain.AddTransaction("Fry", "Bender", 42)
	previousHash = blockChain.LastBlock().Hash()
	nonce := blockChain.ProofOfWork()
	blockChain.CreateBlock(nonce, previousHash)
	blockChain.Print()
}
```

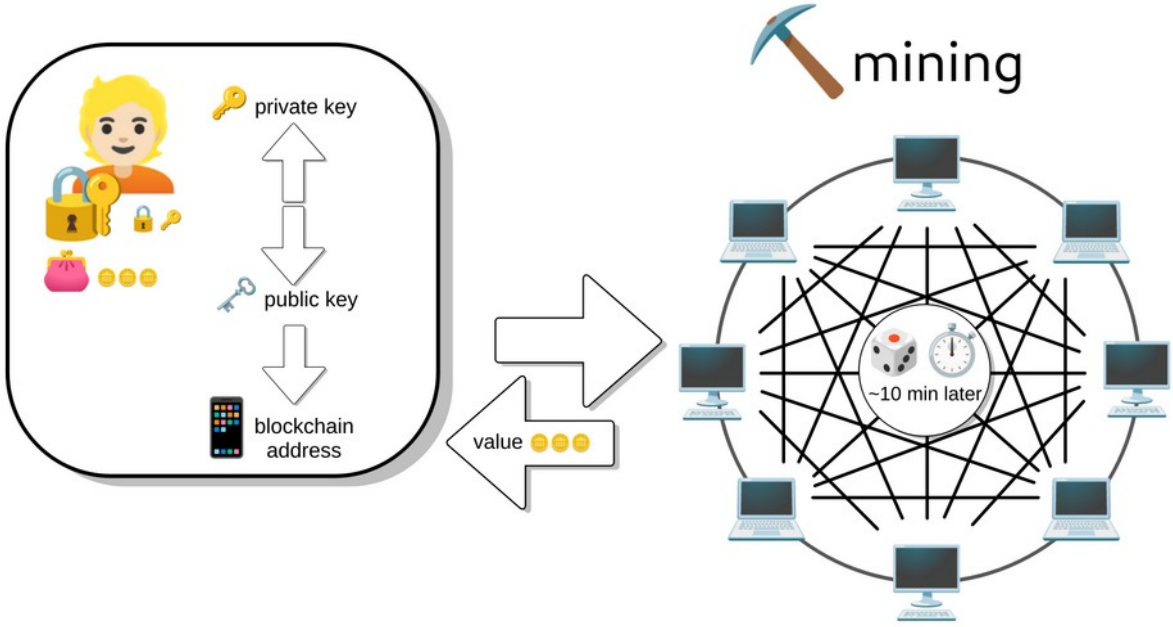
[RUN CHECK OUTPUT AND SEE NONCE VALUES]

As you see you start with an initial genesis block and a nonce of zero but from then on you get a nonce different from zero for every new block found. If you have problems to understand how the proof of work algorithm works it may help to see some interim results for the hashes which are checked generated.

```
func (bc *Blockchain) ValidProof(nonce int, previousHash [32]byte, transactions []*Transaction, difficulty int) bool {
	zeros := strings.Repeat("0", difficulty)
	guessBlock := Block{0, nonce, previousHash, transactions}
	guessHashStr := fmt.Sprintf("%x", guessBlock.Hash())
	fmt.Println(guessHashStr)
	return guessHashStr[:difficulty] == zeros
}
```

[RUN CHECK OUTPUT AND SEE NONCE VALUES]

You see lots of hashes and I want to draw your attention to the last one. Here you can clearly see that this one is the first hash which fulfills the requirement to has the constant difficulty number of zeros in the front.

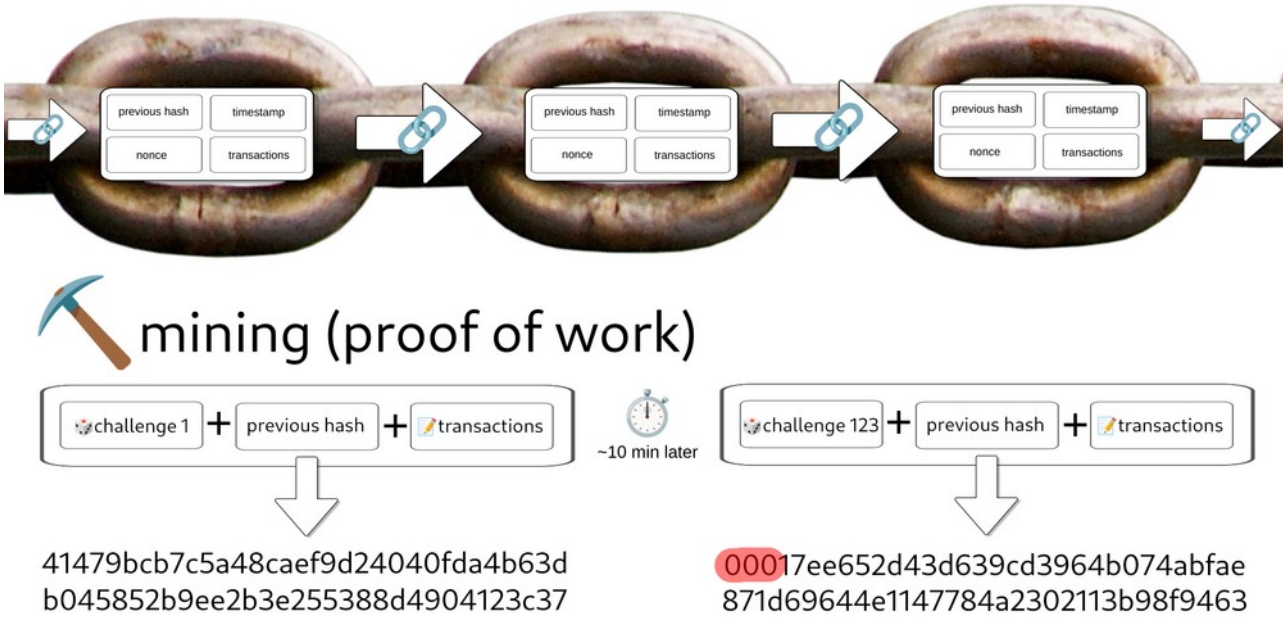


This proof-of-work algorithm is very similar to what thousands of millions of miners do all the time in a kind of competition. The solution to the task can practically only be found by guessing and trial and error, but the solution itself can be checked in a fraction of a second. This way it is proofed that a work has been done which ensures validity of all transactions included.

You can imagine it like children all throwing dice at the same time until a six is rolled. Only much more complex.

[Code Lecture 19]

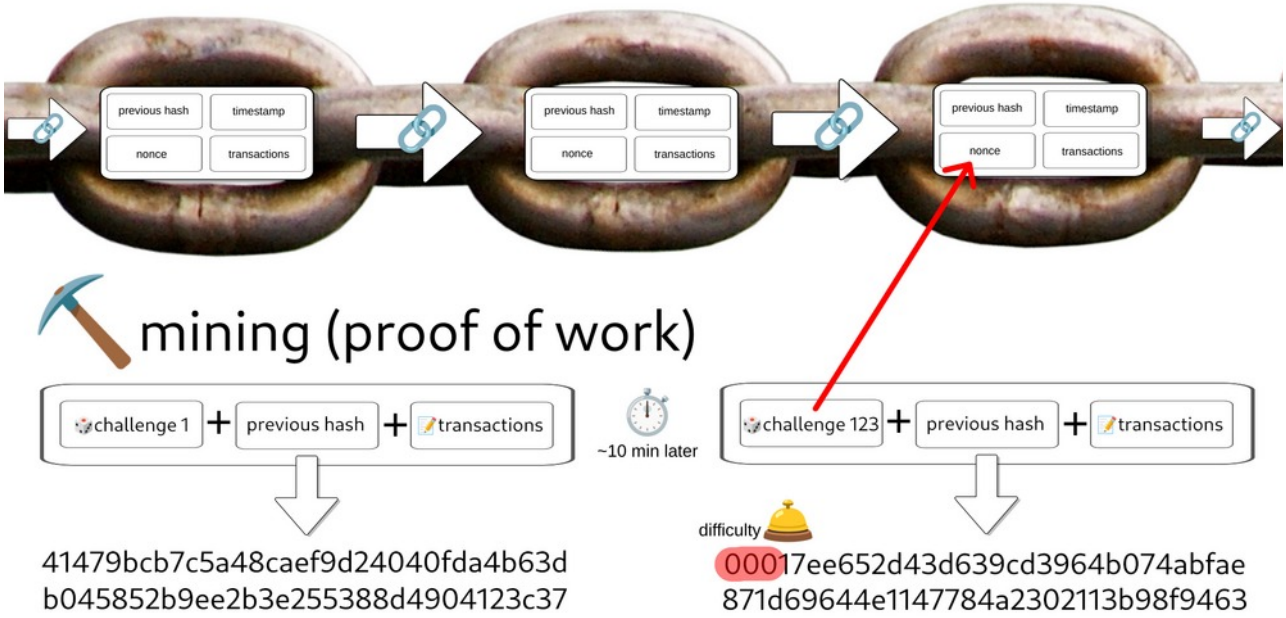
Lecture 20 – All About Mining in This Model (And a Side Note on Minting New Coins/Values)



Mining is often misunderstood, mainly because of the unscripted name, which implies that you uncover hidden values in a similar way to digging for gold. But this is not the case.

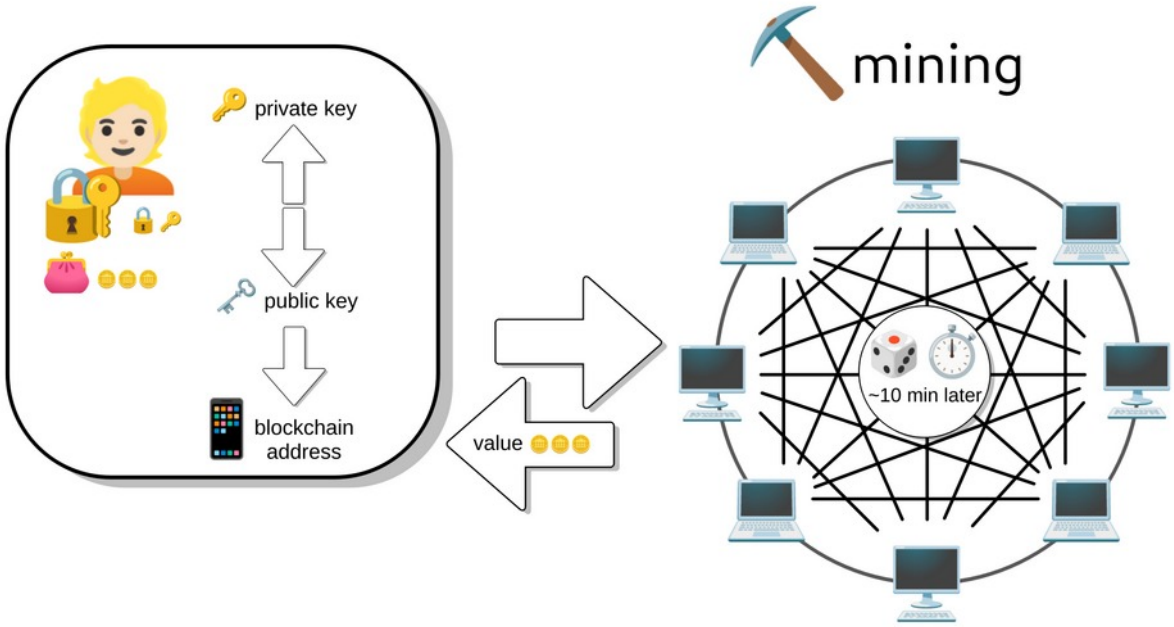
Mining has the intended side effect of decentralized money issuance though. If you look at mining from this perspective, then it is more of issuing coins or banknotes and should therefore be called minting. However, the main task of mining is not the distributed circulation of new cryptocurrency, but the confirmation of the validity of transactions and the packaging of these in a block with a time stamp.

The proof that this work has been done can be found in the hash value and can be checked by any node in a fraction of a second. We therefore focussed on this when implementing the nonce and now when implementing a reward system that generates coins from thin air, so to speak, similar like the initial transaction of each block in which the miner assigns itself the reward - the so called coinbase after which the crypto currency exchange and wallet is named.



For this course, I propose a simplified system for the entire field of "mining".

Every node that has a transaction pool should also be able to mine. We completely ignore the fees that serve as an incentive to confirm transactions and verify blocks with each transaction. And the reward that the miner receives is practically created out of thin air. But the source itself must be that of a registered blockchain participant. We treat it like a "black box". Whenever mining is carried out successfully, a reward comes from there. I call it the "BLOCKCHAIN REWARD SYSTEM (e.g. minting & fees)". That should be enough for our simplified model.



Remember how to get a value for nonce by changing a value within a block until you find a hash value that has the required number of leading zeros. This is exactly mining and now we want to make the process more formal so that you can perform mining in the form of a function call.

[VS CODE]

```
const (
	MINING_DIFFICULTY = 3
	MINING_SENDER     = "BLOCKCHAIN REWARD SYSTEM (e.g. minting & fees)"
	MINING_REWARD     = 1.0
)
```

```
type Blockchain struct {
	transactionPool   []*Transaction
	chain             []*Block
	blockchainAddress string
}
```

The reason for introducing a blockchain address as an element of this structs is clear. This is the reward or minting system and basically the black box from which all crypto assets are issued and put into circulation.

```
func NewBlockchain(blockchainAddress string) *Blockchain {
```

```
    b := &Block{}
    bc := new(Blockchain)
    bc.blockchainAddress = blockchainAddress
    bc.CreateBlock(0, b.Hash())
    return bc
}
```

This ensures that mining and the rewards system are available in every blockchain.

```
func (bc *Blockchain) Mining() bool {
    bc.AddTransaction(MINING_SENDER, bc.blockchainAddress,
MINING_REWARD)
    nonce := bc.ProofOfWork()
    previousHash := bc.LastBlock().Hash()
    bc.CreateBlock(nonce, previousHash)
    log.Println("action=mining, status=success")
    return true
}
```

This is your mining method for your blockchain.

You retrieve the reward and add it as a transaction to the transaction pool.

You then determine a value for nonce by running the proof-of-work algorithm until a hash of the matching the difficulty condition is found. You create the block, write a message to the log and report back success with the return value true. That’s the whole magic.

```
func main() {
    minerBlockchainAddress := "miners_blockchain_address"
    blockChain := NewBlockchain(minerBlockchainAddress)
    blockChain.Print()

    blockChain.AddTransaction("Rick", "Morty", 137)
    blockchain.Mining()
    blockChain.Print()

    blockChain.AddTransaction("Stan", "Francine", 23)
```

```
    blockChain.AddTransaction("Fry", "Bender", 42)
    blockchain.Mining()
    blockChain.Print()
}
```

As you can see, this allows you to consolidate several steps into one function call for mining.

[GO RUN .]

[CHECK OUTPUT EXPLAIN REWARDS AND MINING]

I hope and think that you can follow what is happening, but I would still like to clarify something.

You are currently working on the code of only one node and are creating different transactions from and to different participants. Rick, Morty, Stan, Francine, Fry and Bender. This does not necessarily mean that all these entities share a wallet or operate the same node or are connected to the same node. And then there's a reward system, which does not really exist, and a minerBlockchainAddress, which could basically be anyone and is probably also connected to a completely different node. Nevertheless, every participant can, but does not have to, be a miner if they are mining.

This will become clearer later in the course when we separate the wallets from the nodes, each node generates real blockchain addresses from key pairs and you start several nodes that competitively race to be the first to mine a block and validate transactions.

Code Lecture 20

Lecture 21 – Determine the Total Value of an Address. The Receiver Gets, the Sender Gives!

While you are here implementing methods for the blockchain, you can also do one thing that you will need to do at some point anyway—at the latest in the next section, when it comes to creating wallets.

We are talking about the total amount stored as a value for a blockchain address. The blockchain is basically an account book, a ledger. If you want to determine the total amount stored under an address at the time of the query, you need to offset all inputs and all outputs against each other.

```
func (bc *Blockchain) CalculateTotalAmount(blockchainAddress
string) float32 {
    var totalAmount float32 = 0
    for _, b := range bc.chain {
```

```
            for _, t := range b.transactions {
                value := t.value
                if blockchainAddress == t.recipientBlockchainAddress
{
                    totalAmount += value
                }

                if blockchainAddress == t.senderBlockchainAddress {
                    totalAmount -= value
                }
            }
        }
        return totalAmount
}
```

There is certainly a better or at least more elegant way to determine the total sum of a blockchain address, but I think this method is excellent for demonstrating how the values of a cryptocurrency are stored on a blockchain.

```
func main() {
        minerBlockchainAddress := "miners_blockchain_address"
        blockChain := NewBlockchain(minerBlockchainAddress)
        blockChain.Print()

        blockChain.AddTransaction("Rick", "Morty", 137)
        blockchain.Mining()
        blockChain.Print()

        blockChain.AddTransaction("Stan", "Francine", 23)
        blockChain.AddTransaction("Fry", "Bender", 42)
        blockchain.Mining()
        blockChain.Print()

        fmt.Printf("Miner %.1f\n",
blockChain.CalculateTotalAmount(minerBlockchainAddress))
```

```
    fmt.Printf("Stan %.1f\n",
blockChain.CalculateTotalAmount("Stan"))
    fmt.Printf("Francine %.1f\n",
blockChain.CalculateTotalAmount("Francine"))
}
```

[RUN]

As you can see from your output here:

- the miner had to create two blocks and was rewarded with a value of one each. This makes a total amount of two.
- Stan has sent 23 to Francine. That puts him 23 in the red. Of course, this is not possible with a real blockchain but at the moment you can just allow that.
- Francine, on the other hand, has received 23 from Stan and the total amount stored under your blockchain address is 23.

Please also note that whenever you want to query the value stored on an address, you must always check the blockchain from the beginning of its existence to the current time.

In a real blockchain application, you would of course cache the values you found and then only update them when your node is synchronized with the blockchain at the current transaction status.

Code Lecture 21

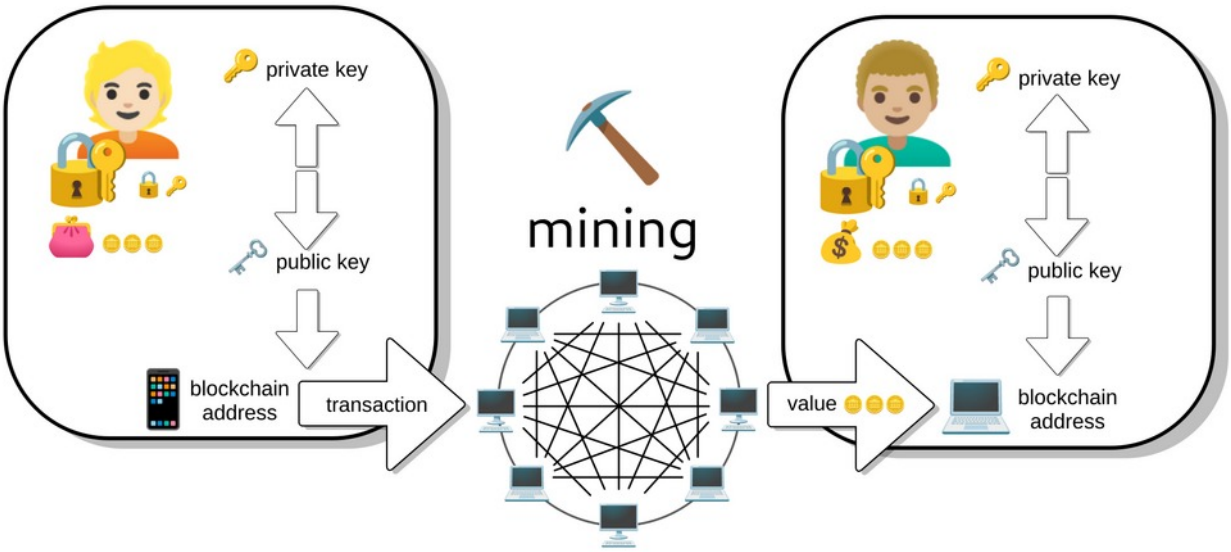
Section 4 – Program a Wallet: This Is GO, So It Will Be an Online Wallet!

I Left My Wallet in El Segundo ...
A Tribe Called Quest

Lecture 22 – The Elliptic Curve Digital Signature Algorithm (ECDSA)

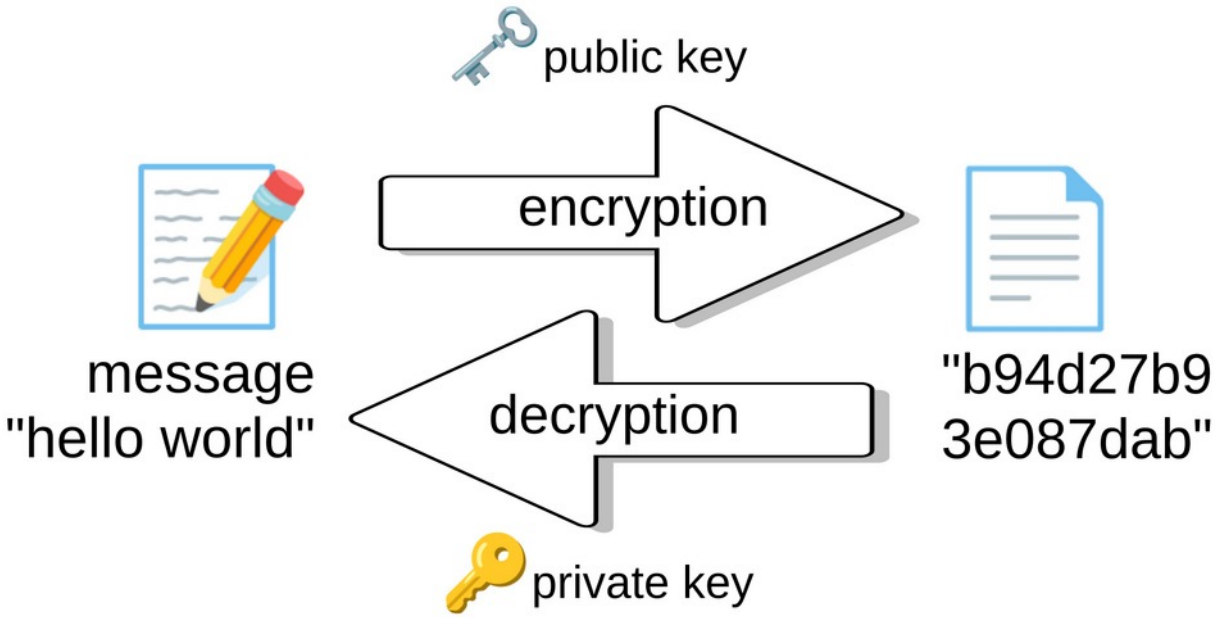
This section gets a little mathematical, but I'll keep the basics relatively abstract here and only go into them to the extent necessary to create a wallet, respectively a blockchain address from a pair of keys.

And this lesson is about private keys, public keys, encryption and decryption, signatures and elliptic curves, which help to make the transactions on your blockchain secure!



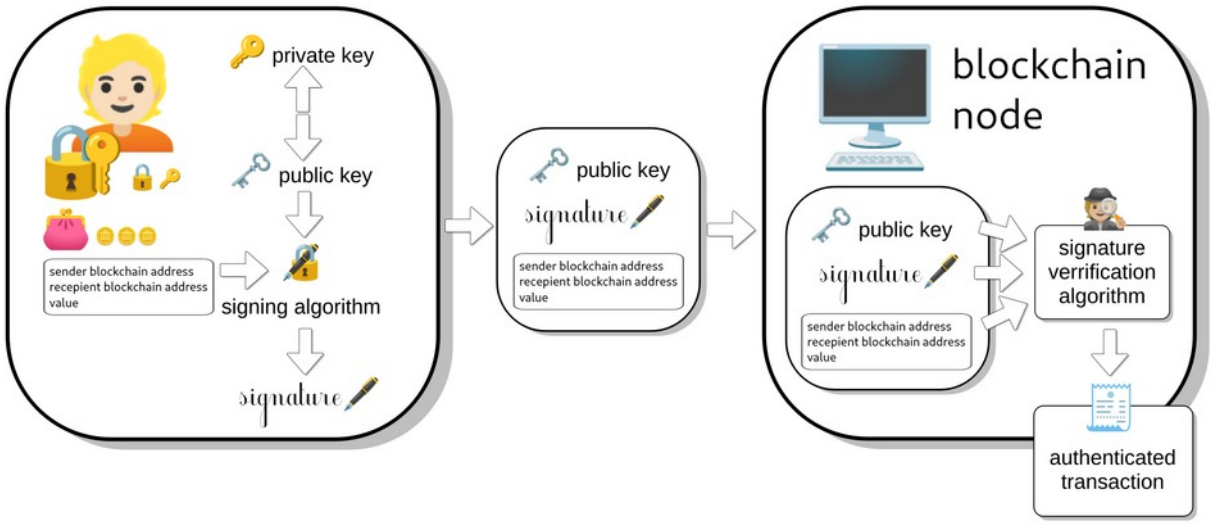
As you can see, wallets are separate from the actual blockchain node.

A wallet is a user interface that mediates between the user and the blockchain node. Instructions that are sent to the node must be legitimized. The legitimization that allows an address to be used results from the combination of a private key and a public key. This can be used to generate a blockchain address that is accepted by every participant in the network.



In simple terms, you could say that a message can be encrypted with the public key, but can only be decrypted, in other words made readable again, with the private key. As you can see, the concept of keys in cryptography always works as a pair.

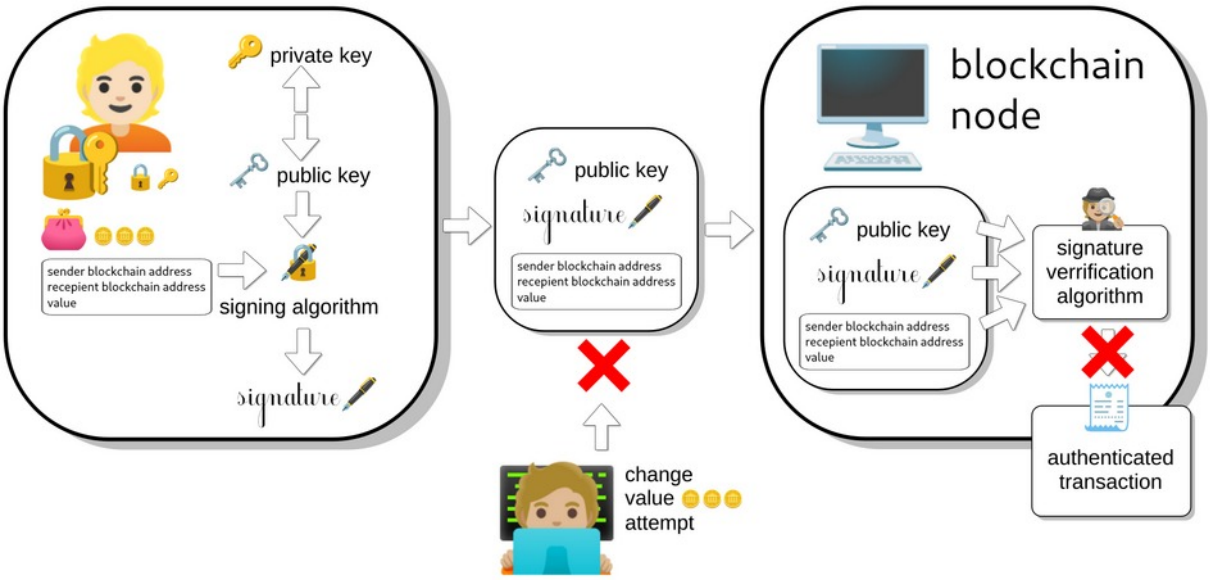
The first step to participating in the blockchain network is to create a suitable key pair consisting of a secret private key and a public key. From this you can derive a blockchain address. Messages or instructions consisting of the public key and a signature can only be created using the private key.



Here you can see how a wallet communicates with a blockchain node. A signature for a transaction is created with the key pair and the transaction is signed on wallet side.

The signature is transmitted to the blockchain node together with the public key and the message, in this case the transaction.

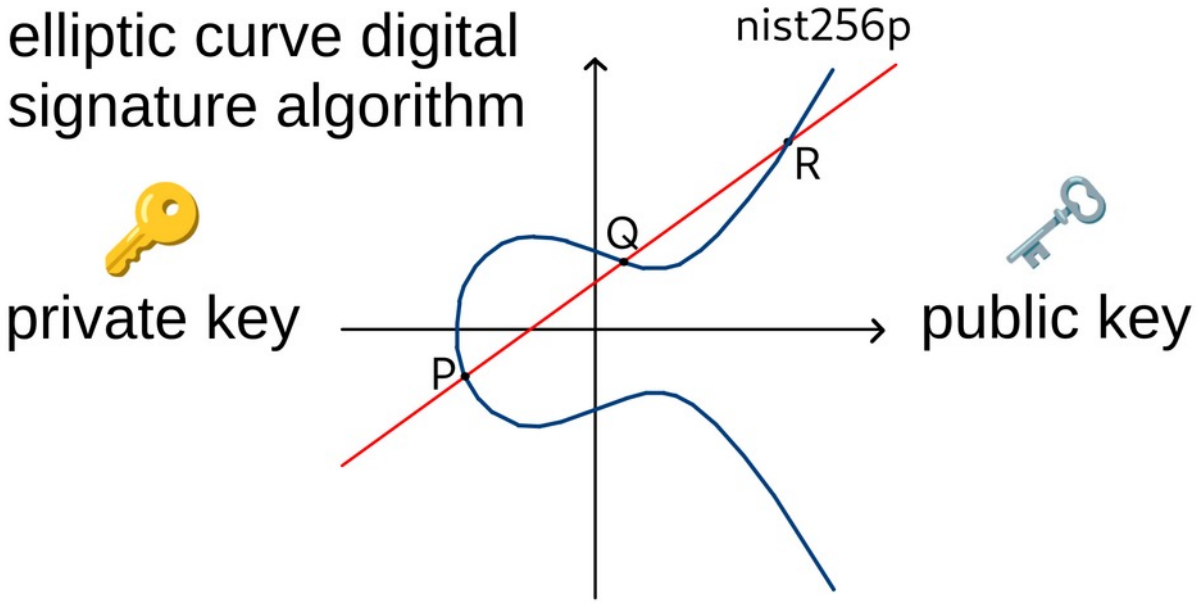
This transmitted data can be checked very easily by the blockchain node, and also by any other node, and verified as valid or rejected as invalid. You will see a signing algorithm on the wallet side and a signature verification algorithm on the blockchain node.



A potential attacker could try to intercept the message and change a value or the receiving address while the data is being transmitted from the wallet to the blockchain node.

But he does not have the private key that was used when the signature was crated. Due to the cryptographic features of the signature and public key, the node cannot draw any conclusions about the private key.

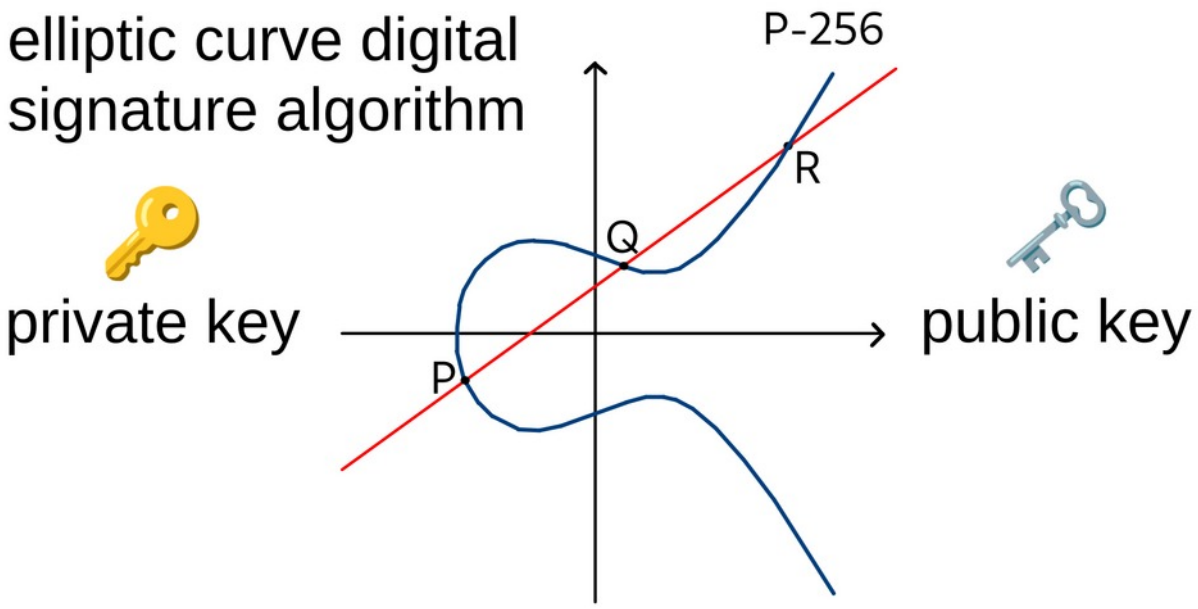
Therefore, any subsequently modified transaction that is not validly signed is rejected by the blockchain node. Even if the attacker himself were to operate a blockchain node that accepts the compromised transaction, the transaction would be rejected by the next blockchain node to which it is transmitted at the next.



A so-called man-in-the-middle attack is doomed to failure.

For the sake of completeness, I would like to outline here at least how the signature algorithm is structured. It is an Elliptic Curve Digital Signature Algorithm (ECDSA). ECDSA offers a variant of the Digital Signature Algorithm (DSA) which includes elliptic-curve cryptography.

I will gladly spare you the mathematical-cryptographic background here. You can read about it on the Wikipedia page for ECDSA. But basically, you encrypt a message with a combination of different components that all lie on an elliptic curve, which you have to agree on beforehand. You often read NIST256p or P-256 in this context, but both basically mean the same thing.



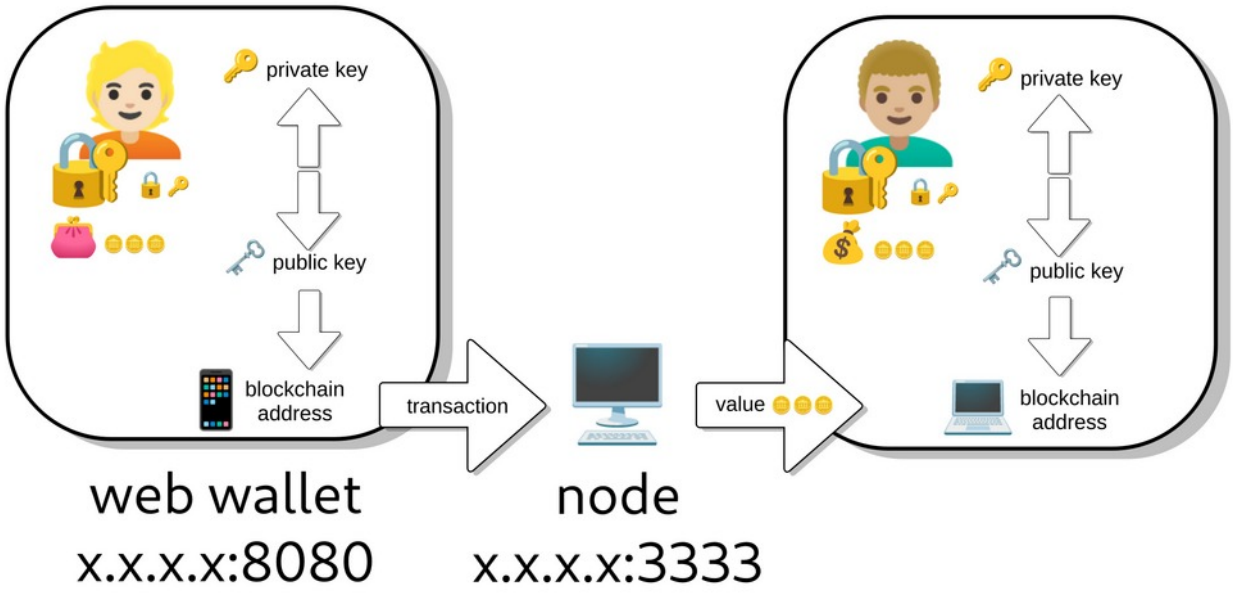
In very simplified terms, however, it works like this: you look for a point P on an elliptical curve and consider it to be the private key. And now draw a straight line that intersects the elliptical curve at further points Q and R. If you now choose this point Q somewhere else here, there will always be a different point R. If you now use any point Q to encrypt a message, there is practically no practicable way to determine P from it.

All cases in which encryption based on elliptic curves was broken were due to faulty implementations or incorrect use of ECDSA. For example, the signatures used by Sony to encrypt the operating system of the PlayStation 3 were broken this way.

The current implementation of ECDSA in the GO Standard Library is considered secure. In the following lessons, you will generate secure key pairs, create blockchain addresses, sign transactions and verify the transaction requests that are transmitted with public keys and signatures.

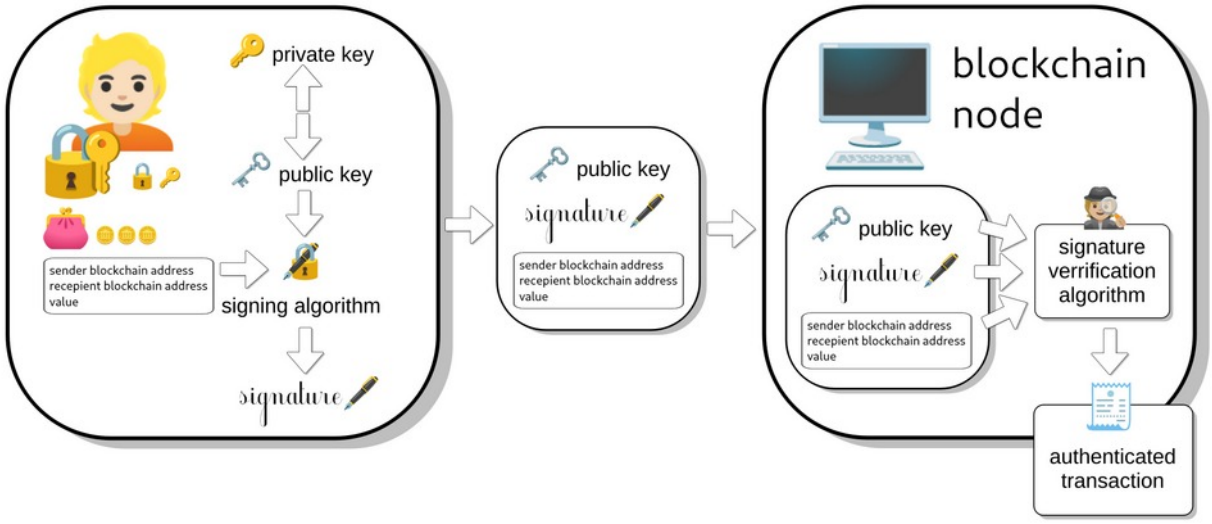
Wallet Development's History

Lecture 23 – Build Just a Very Basic Wallet. A Whole Wallet Is Too Much for This Example



As previously mentioned, you will need to separate the blockchain node and wallet. This is also a good occasion to subdivide the blockchain package somewhat.

```
myGoBlockchain
|-- LICENSE
|-- README.md
|-- blockchain
|   |-- block.go
|   |-- blockchain.go
|   `-- transaction.go
|-- go.mod
|-- main.go
`-- wallet
    `-- wallet.go
```



```
package wallet

import (
        "crypto/ecdsa"
        "crypto/elliptic"
        "crypto/rand"
        "fmt"
)

type Wallet struct {
        privateKey *ecdsa.PrivateKey
        publicKey  *ecdsa.PublicKey
}

func NewWallet() *Wallet {
        w := new(Wallet)
        privateKey, _ := ecdsa.GenerateKey(elliptic.P256(),
rand.Reader)
        w.privateKey = privateKey
        w.publicKey = &w.privateKey.PublicKey
        return w
}

func (w *Wallet) PrivateKey() *ecdsa.PrivateKey {
        return w.privateKey
}

func (w *Wallet) PrivateKeyStr() string {
        return fmt.Sprintf("%x", w.privateKey.D.Bytes())
}

func (w *Wallet) PublicKey() *ecdsa.PublicKey {
        return w.publicKey
}
```

```
func (w *Wallet) PublicKeyStr() string {
        return fmt.Sprintf("%x%x", w.publicKey.X.Bytes(),
w.publicKey.Y.Bytes())
}
```

```
package main

import (
     "fmt"
     "log"

     "github.com/jagottsicher/myGoBlockchain/wallet"
)

func init() {
     log.SetPrefix("Blockchain: ")
}

func main() {
     w := wallet.NewWallet()
     fmt.Println(w.PrivateKeyStr())
     fmt.Println(w.PublicKeyStr())
}
```

https://pkg.go.dev/crypto/ecdsa

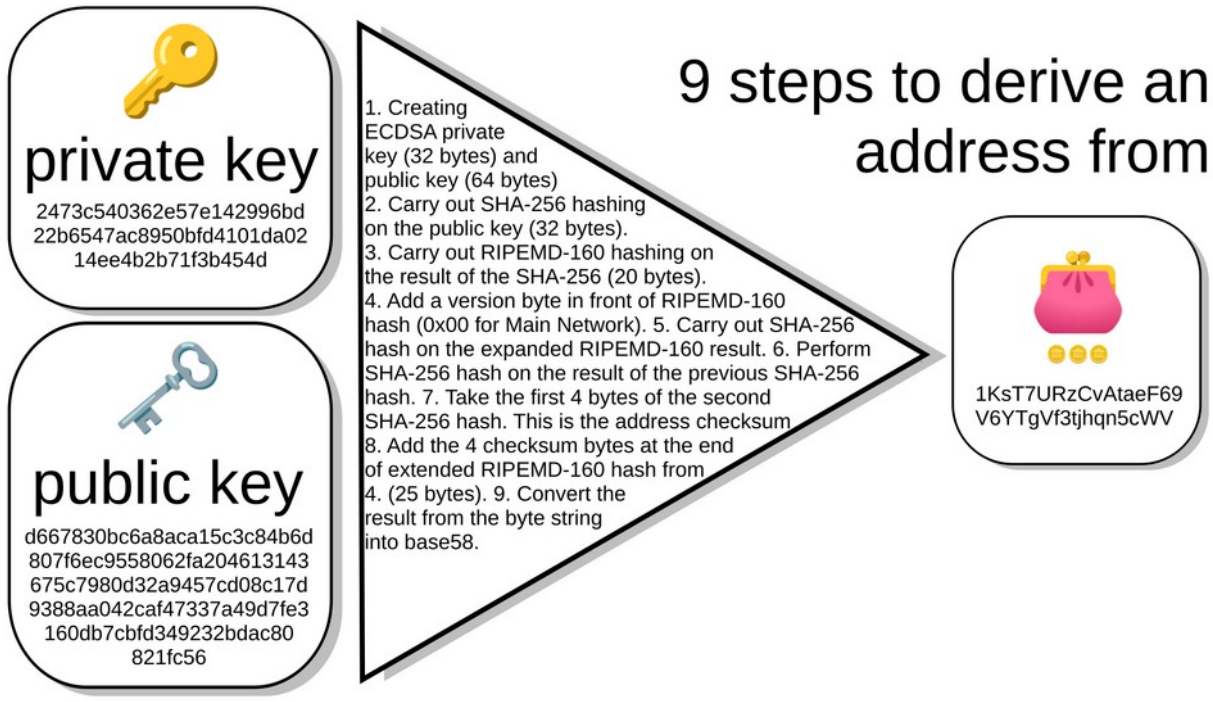
Code Lecture 23

Lecture 24 – How Do You Get From a Private and Public Key to a Blockchain Address? Watch now!

[VS CODE]

You have now successfully created a way to create a wallet. So far, however, this consists of two long hexadecimal numbers.

Firstly, they are difficult to remember, and secondly, they are quite long and unwieldy. But there is a solution, because you can derive an address from them that is significantly shorter and reasonably easy to handle.



This is what this lesson will be about.

There are various options and approaches for generating such an address, which have been developed and refined in recent years. However, a first-generation address will be used here.

The technical background is explained in a wiki. You are welcome to study it further, but there is also a comprehensible step-by-step guide in nine simple steps on how to get from an ECDSA private key to a so-called base58 string, which represents a valid address.

https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses

Explaining all the details and considerations would go far beyond the scope of this course. Simply treat these instructions like a recipe for your favorite dish that you want to cook!

1. Creating ECDSA private key (32 bytes) and public key (64 bytes)
2. Carry out SHA-256 hashing on the public key (32 bytes).
3. Carry out RIPEMD-160 hashing on the result of the SHA-256 (20 bytes).
4. Add a version byte in front of RIPEMD-160 hash (0x00 for Main Network).
5. Carry out SHA-256 hash on the expanded RIPEMD-160 result.
6. Perform SHA-256 hash on the result of the previous SHA-256 hash.
7. Take the first 4 bytes of the second SHA-256 hash. This is the address checksum
8. Add the 4 checksum bytes at the end of extended RIPEMD-160 hash from 4. (25 bytes).
9. Convert the result from the byte string into base58.

https://en.bitcoin.it/wiki/Invoice_address

https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses

Lecture 25 – Implementation of the Creation of a Genuine Blockchain Address!

[TERMINAL IN PROJECT FOLDER]

```
go get github.com/btcsuite/btcutil/base58
go get golang.org/x/crypto/ripemd160
```

[WALLET.GO]

```
type Wallet struct {
	privateKey         *ecdsa.PrivateKey
	publicKey          *ecdsa.PublicKey
	blockchainAddress string
}

func NewWallet() *Wallet {
// 1. Creating ECDSA private key (32 bytes) and public key (64 bytes)
	w := new(Wallet)
	privateKey, _ := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
	w.privateKey = privateKey
	w.publicKey = &w.privateKey.PublicKey
	// 2. Carry out SHA-256 hashing on the public key (32 bytes).
	// 3. Carry out RIPEMD-160 hashing on the result of the SHA-256 (20 bytes).
	// 4. Add a version byte in front of RIPEMD-160 hash (0x00 for Main Network).
	// 5. Carry out SHA-256 hash on the expanded RIPEMD-160 result.
```

```
	// 6. Perform SHA-256 hash on the result of the previous SHA-256 hash.
	// 7. Take the first 4 bytes of the second SHA-256 hash as checksum.
	// 8. Add the 4 checksum bytes at the end of extended RIPEMD-160 hash from 4. (25 bytes).
	// 9. Convert the result from the byte string into base58.
	return w
}


func NewWallet() *Wallet {
	// 1. Creating ECDSA private key (32 bytes) and public key (64 bytes)
	w := new(Wallet)
	privateKey, _ := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
	w.privateKey = privateKey
	w.publicKey = &w.privateKey.PublicKey
	// 2. Carry out SHA-256 hashing on the public key (32 bytes).
	h2 := sha256.New()
	h2.Write(w.publicKey.X.Bytes())
	h2.Write(w.publicKey.Y.Bytes())
	digest2 := h2.Sum(nil)
	// 3. Carry out RIPEMD-160 hashing on the result of the SHA-256 (20 bytes).
	h3 := ripemd160.New()
	h3.Write(digest2)
	digest3 := h3.Sum(nil)
	// 4. Add a version byte in front of RIPEMD-160 hash (0x00 for Main Network).
	vd4 := make([]byte, 21)
	vd4[0] = 0x00
	copy(vd4[1:], digest3[:])
	// 5. Carry out SHA-256 hash on the expanded RIPEMD-160 result.
```

```
	h5 := sha256.New()
	h5.Write(vd4)
	digest5 := h5.Sum(nil)
	// 6. Perform SHA-256 hash on the result of the previous SHA-256 hash.
	h6 := sha256.New()
	h6.Write(digest5)
	digest6 := h6.Sum(nil)
	// 7. Take the first 4 bytes of the second SHA-256 hash as checksum.
	chsum := digest6[:4]
	// 8. Add the 4 checksum bytes at the end of extended RIPEMD-160 hash from 4. (25 bytes).
	dc8 := make([]byte, 25)
	copy(dc8[:21], vd4[:])
	copy(dc8[21:], chsum[:])
	// 9. Convert the result from the byte string into base58.
	address := base58.Encode(dc8)
	w.blockchainAddress = address
	return w
}
```

[END OF FILE]

```
func (w *Wallet) BlockchainAddress() string {
	return w.blockchainAddress
}
```

[MAIN.GO]

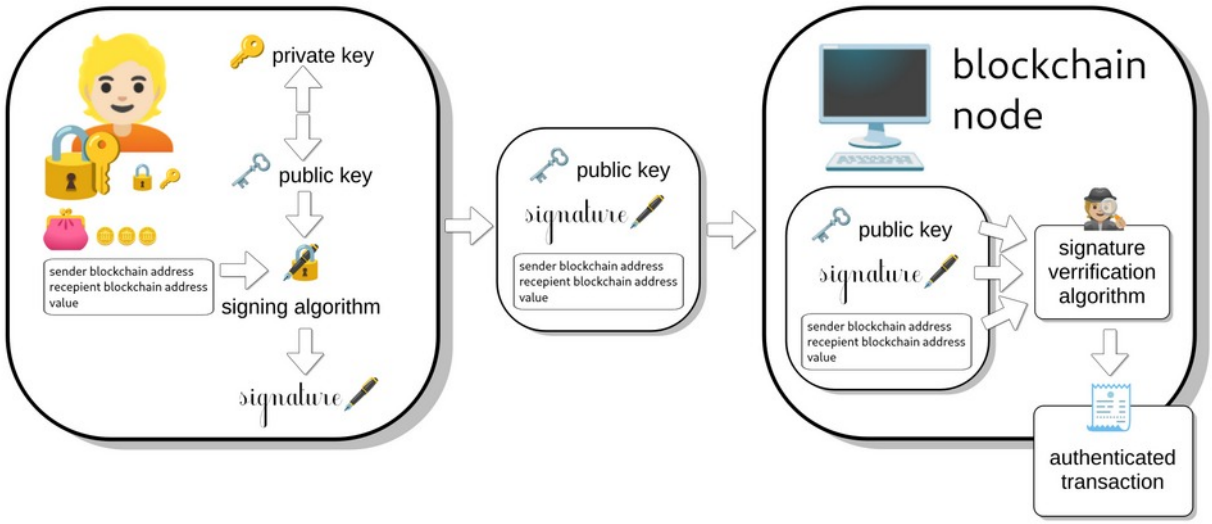
```
func main() {
	w := wallet.NewWallet()
	fmt.Println(w.PrivateKeyStr())
	fmt.Println(w.PublicKeyStr())
	fmt.Println(w.BlockchainAddress())
```

```
}
```

[RUN]

Code Lecture 25

Lecture 26 – Signatures! Sign here, here and here. Every transaction must be duly signed!



[WALLET/TRANSACTION.GO]

```
type Transaction struct {
     senderPrivateKey           *ecdsa.PrivateKey
     senderPublicKey            *ecdsa.PublicKey
     senderBlockchainAddress    string
     recipientBlockchainAddress string
     value                      float32
}

func NewTransaction(privateKey *ecdsa.PrivateKey, publicKey
*ecdsa.PublicKey,
     sender string, recipient string, value float32) *Transaction {
     return &Transaction{privateKey, publicKey, sender, recipient,
value}
}
```

```
func (t *Transaction) GenerateSignature() *utils.Signature {
	m, _ := json.Marshal(t)
	h := sha256.Sum256([]byte(m))
	r, s, _ := ecdsa.Sign(rand.Reader, t.senderPrivateKey, h[:])
	return &utils.Signature{R: r, S: s}
}
```

[UTILS/ECDSA.GO] create and implement:

```
package utils

type Signature struct {
	R *big.Int
	S *big.Int
}
```

[WALLET/TRANSACTION.GO]

You have to remember that you have to provide a method `MarshalJSON()` due to the fact that your `type Transaction struct` works with lower-case members and cannot be marshaled with `json.Marshal()` as long it does not bring its own method as a wrapper with it.

[CTRL + CLICK ON JSON-MARSHAL]

By the way, that works because of the feature of interfaces in GO called interfaces. Here a `type Marshaler` is defined as an `interface` which makes all types which implement a method `MarshalJSON`, no matter of what type they are, also as of type Marshaler. That's the whole magic here.

You can copy the method from `package blockchain`, because this type transaction is in a different package and hence a different type, but for creating the signature we will need marshal the elements needed for transactions only and that are sender, recipient and value only – private key and public key are omitted!

```
func (t *Transaction) MarshalJSON() ([]byte, error) {
	return json.Marshal(struct {
		Sender    string  `json:"sender_blockchain_address"`
		Recipient string  `json:"recipient_blockchain_address"`
		Value     float32 `json:"value"`
```

```
    }{
        Sender:    t.senderBlockchainAddress,
        Recipient: t.recipientBlockchainAddress,
        Value:     t.value,
    })
}
```

[UTILS/ECDSA]

```
func (s *Signature) String() string {
    return fmt.Sprintf("%064x%064x", s.R, s.S)
}
```

[MAIN.GO]

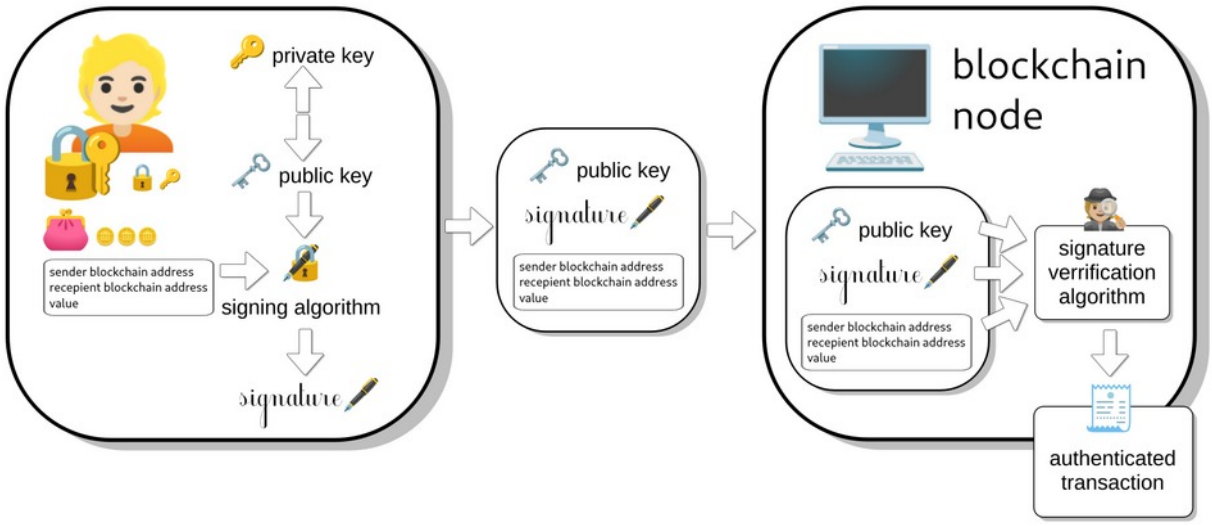
```
func main() {
    w := wallet.NewWallet()
    fmt.Println(w.PrivateKeyStr())
    fmt.Println(w.PublicKeyStr())
    fmt.Println(w.BlockchainAddress())

    t := wallet.NewTransaction(w.PrivateKey(), w.PublicKey(), w.BlockchainAddress(), "Morty", 137.0)
    fmt.Printf("signature %s\n", t.GenerateSignature())
}
```

[RUN]

We can use `%s` as verb in this `Sprintf()`, because our method for `type transaction GenerateSiganture` has a function `String()`. That works seamless together with `Spintf()` providing the right data to output the signature in a human-readable format.

Code Lecture 26

Lecture 27 – Transaction Verification. Get Out Your Loupe, Check With Mathematical Precision!



[FOCUS ON TRANSACTION VERIFICATION]

[BLOCKCHAIN/BLOCKCHAIN.GO BELOW ADDTRANSACTION]

```
func (bc *Blockchain) VerifyTransactionSignature(
     senderPublicKey *ecdsa.PublicKey, s *utils.Signature, t
*Transaction) bool {
     m, _ := json.Marshal(t)
     h := sha256.Sum256([]byte(m))
     return ecdsa.Verify(senderPublicKey, h[:], s.R, s.S)
}
```

[FOCUS ON SIGNATURE VERIFICATION ALGORITHM]

[BLOCKCHAIN/BLOCKCHAIN.GO ADDTRANSACTION]

```
func (bc *Blockchain) AddTransaction(sender string, recipient
string, value float32,
     senderPublicKey *ecdsa.PublicKey, s *utils.Signature) bool {
     t := NewTransaction(sender, recipient, value)

     if sender == MINING_SENDER {
```

```
            bc.transactionPool = append(bc.transactionPool, t)
            return true
        }

        if bc.VerifyTransactionSignature(senderPublicKey, s, t) {
            bc.transactionPool = append(bc.transactionPool, t)
            return true
        } else {
            log.Println("ERROR: Could not verify transaction")
        }
        return false
}
```

[BLOCKCHAIN/BLOCKCHAIN.GO MINING]

```
bc.AddTransaction(MINING_SENDER, bc.blockchainAddress,
MINING_REWARD, nil, nil)
```

[MAIN.GO]

```
func main() {
    walletMiner := wallet.NewWallet()
    walletAlice := wallet.NewWallet()
    walletBob := wallet.NewWallet()

    // wallet transaction request
    t := wallet.NewTransaction(walletAlice.PrivateKey(),
walletAlice.PublicKey(), walletAlice.BlockchainAddress(),
walletBob.BlockchainAddress(), 23.0)

    // blockchain node transaction request handling
    blockchain :=
blockchain.NewBlockchain(walletMiner.BlockchainAddress())
    isAdded :=
blockchain.AddTransaction(walletAlice.BlockchainAddress(),walletBo
```

```
b.BlockchainAddress(), 23.0, walletAlice.PublicKey(),
t.GenerateSignature())
     fmt.Println("Transaction added to transaction pool?", isAdded)
}
```

[FOCUS ON CHECK 5 DIFFERENT VALUES FOR]

[RUN]

[MAIN.GO]

```
func main() {
     walletMiner := wallet.NewWallet()
     walletAlice := wallet.NewWallet()
     walletBob := wallet.NewWallet()

     // wallet transaction request
     t := wallet.NewTransaction(walletAlice.PrivateKey(),
walletAlice.PublicKey(), walletAlice.BlockchainAddress(),
walletBob.BlockchainAddress(), 23.0)

     // blockchain node transaction request handling
     blockchain :=
blockchain.NewBlockchain(walletMiner.BlockchainAddress())
     isAdded :=
blockchain:AddTransaction(walletAlice.BlockchainAddress(),walletBo
b.BlockchainAddress(), 23.0, walletAlice.PublicKey(),
t.GenerateSignature())
     fmt.Println(“Transaction added to transaction pool?”, isAdded)

     blockchain.Mining()
     blockchain.Print()

fmt.Printf(“Miner has %.1f\n”,
blockchain.CalculateTotalAmount(walletMiner.BlockchainAddress()))
```

```
    fmt.Printf("Alice has %.1f\n",
blockchain.CalculateTotalAmount(walletAlice.BlockchainAddress()))
fmt.Printf("Bob has %.1f\n",
blockchain.CalculateTotalAmount(walletBob.BlockchainAddress()))
}
```

[RUN]

[BLOCKCHAIN/BLOCKCHAIN.GO ADDTRANSACTION]

```
func (bc *Blockchain) AddTransaction(sender string, recipient
string, value float32,
    senderPublicKey *ecdsa.PublicKey, s *utils.Signature) bool {
    t := NewTransaction(sender, recipient, value)

    if sender == MINING_SENDER {
        bc.transactionPool = append(bc.transactionPool, t)
        return true
    }

    if bc.VerifyTransactionSignature(senderPublicKey, s, t) {
        if bc.CalculateTotalAmount(sender) < value {
            log.Println("ERROR: Not enough balance in wallet")
            return false
        }
        bc.transactionPool = append(bc.transactionPool, t)
        return true
    } else {
        log.Println("ERROR: Could not verify transaction")
    }
    return false
}
```

[RUN – CHECK LOG]

[BLOCKCHAIN/BLOCKCHAIN.GO ADDTRANSACTION]

```
func (bc *Blockchain) AddTransaction(sender string, recipient
string, value float32,
     senderPublicKey *ecdsa.PublicKey, s *utils.Signature) bool {
     t := NewTransaction(sender, recipient, value)

     if sender == MINING_SENDER {
          bc.transactionPool = append(bc.transactionPool, t)
          return true
     }

     if bc.VerifyTransactionSignature(senderPublicKey, s, t) {
          // if bc.CalculateTotalAmount(sender) < value {
          //   log.Println("ERROR: Not enough balance in wallet")
          //   return false
          // }
          bc.transactionPool = append(bc.transactionPool, t)
          return true
     } else {
          log.Println("ERROR: Could not verify transaction")
     }
     return false
}
```

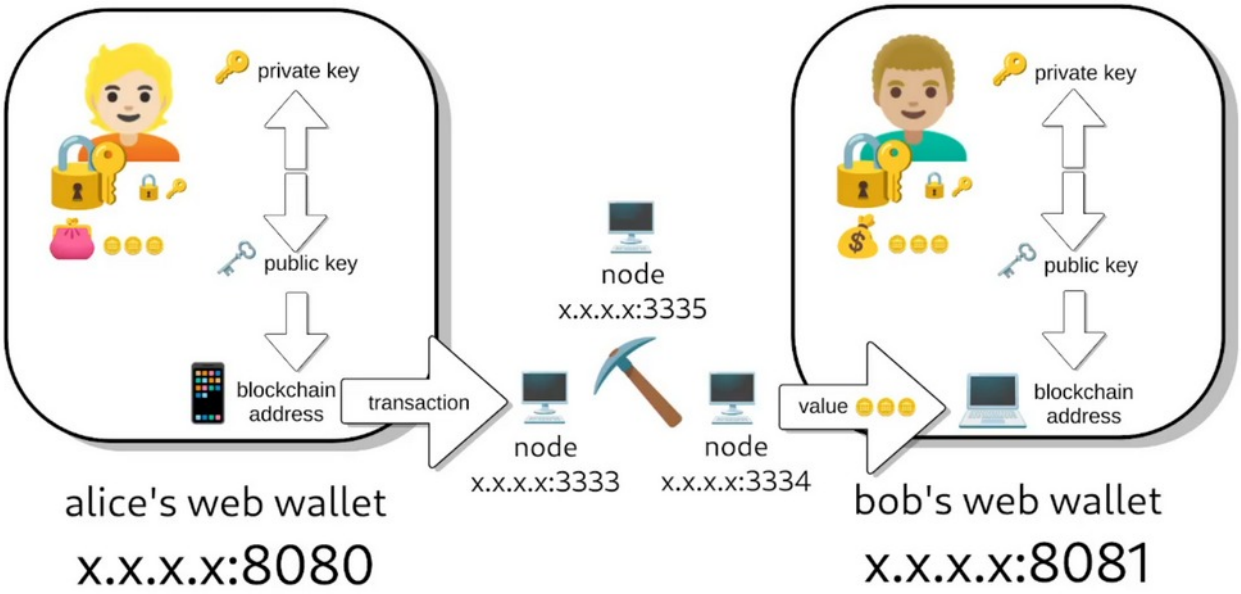
Code Lecture 27

Section 5 – Making Connections: Create an Online Wallet and an API to Your Blockchain Node

> *Everywhere was connected to everywhere else, Pismire had said.*
>
> *Terry Pratchett*

Lecture 28 – Start Your Web Server and See GO Flexing His Muscles!

Throughout this section, you will learn to take full advantage of the strengths that make Google's Go programming language so exceptionally useful. You will be launching servers, deploying services and provide APIs in low-to-no time; things that take much longer in other programming languages. Watch out, as Francine once said in American Dad: “Soon things are getting too spicy for the pepper!”



The blockchain is based on a decentralized network of individual nodes. The nodes exchange data with each other peer-to-peer. In addition, there are wallets that users can also access via the network, and these wallets in turn also send their requests and orders to individual blockchain nodes.

In reality, both the nodes and the wallets are connected via the internet. During development, however, we will simplify the system. You can't expect everyone to have a bunch of computers available in their home network, or that the computing power of their home workstation is sufficient to start several virtual machines.

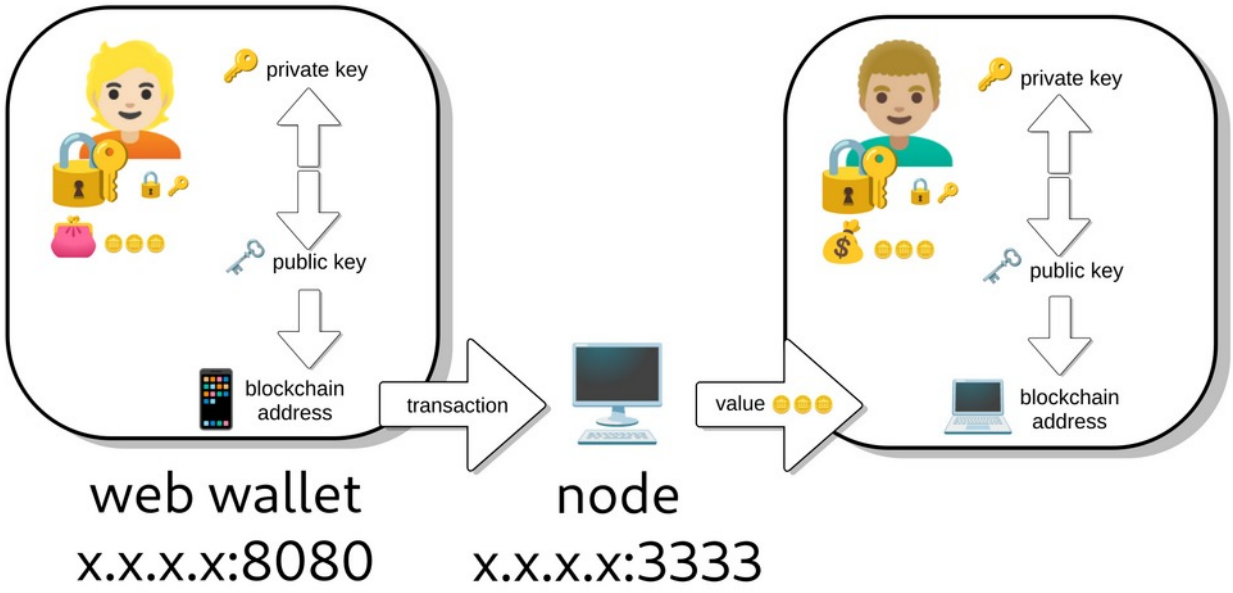
Instead, we will start various instances of the blockchain nodes yet to be implemented on localhost. In Order to identify them these nodes will listen on different ports to provide their services. Let's say between 3333 and the following.

The wallets start web servers to grant users access because ports 80 and 443 are often already occupied for web services, we use ports 8080 and 8081 for our two wallets here.

This is a very simple solution that basically emulates the behavior in the wild in a closed area of the home network on a single computer.

[VS CODE]

We must therefore be able to start two types of web servers. One for the wallets, which can connect to a blockchain and the user can interact with. And another one for each individual node, which finds all the other nodes in your neighborhood and shares and exchanges information with them, but also with a connected wallet on request.



First we take care of the services of the blockchain node, specifically the node which services will be available on port 3333.

[VS CODE]

Thanks to some care in advance, you already have two separate packages that share utilities in a third package: Wallet, Utils and Blockchain.

Create a new folder and name it: `blockchain_node`

Here you can create a file `main.go` of `package main` as well as a file `blockchain_node.go` as of `package main`. You only take a part of the old `main.go` in the parent folder and then delete it. Respectively I move the main.go here and delete everything from it and add a hello-word-like output here.

[MAIN.GO]

```
package main

func init() {
     log.SetPrefix("Blockchain Node: ")
}

func main() {
     fmt.Println("Hello World")
}
```

[TERMINAL]

If you from within this folder execute `go run .` the package main and all necessary packages will be imported, the package will be compiled and the included functions `init()` and `main()` will be executed. Looks fair enough at this point.

[BLOCKCHAIN_NODE.GO]

```
package main

type BlockchainNode struct {
     port uint16
}

func NewBlockchainNode(port uint16) *BlockchainNode {
     return &BlockchainNode{port}
}

func (bcn *BlockchainNode) Port() uint16 {
     return bcn.port
}

func HelloWorld(w http.ResponseWriter, r *http.Request) {
```

```
	io.WriteString(w, "Hello World")
}

func (bcn *BlockchainNode) Run() {
	http.HandleFunc("/", HelloWorld)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}
```

This is the simplest method I know of to start a web server with GO. Further information on creating web servers can be found on countless websites, including: https://medium.com/nerd-for-tech/golang-build-a-simple-web-server-and-interact-with-it-689ee0f4d1de

The concept is always based on the same principles of performing a request/response cycle for get and POST requests.

[UDEMY PROFILE PAGE]

In my courses on Udemy you will also find courses for beginners and faily advanced developers, in which far more complex web servers with routing, middle ware, database connection, JavaScript and templates are covered.

If you have problems to grasp what's going on at all in this lecture I highly recommend that you familiarize yourself with the basics of developing web applications with Go. This article on https://go.dev/doc/articles/wiki/ is of invaluable help, as it basically makes everything very clear, easy to understand and comprehensible in an acceptably short time.

[MAIN.GO]

```
package main

func init() {
	log.SetPrefix("Blockchain Node: ")
}

func main() {
	port := flag.Uint("port", 3333, "TCP Port Number for
Blockchain Node")
	flag.Parse()
	fmt.Println(uint16(*port)
```

```
}
```

[RUN -help]

[RUN]

[CTRL + C]

[RUN with -port=3334]

```
func main() {
     port := flag.Uint("port", 3333, "TCP Port Number for
Blockchain Server")
     flag.Parse()
     fmt.Println(uint16(*port)

     app := NewBlockchainNode(uint16(*port))
     log.Println("Starting blockchain node on port:", *port)
     app.Run()
}
```

[RUN]

[CTRL + C]

Depending on your operating system, it may be necessary at this point to open the port in response to a request from your firewall.

[RUN with -port=3334]

Now open a web browser. I currently prefer Firefox. I'll explain why later. Now enter in the address line:

```
localhost:3333
```

and in another browser tab:

```
localhost:3334
```

You can see that your two blockchain nodes are returning "Hello World" on different ports. This means that you can now start several instances of a blockchain node in parallel, each of which provides a web server on different ports.

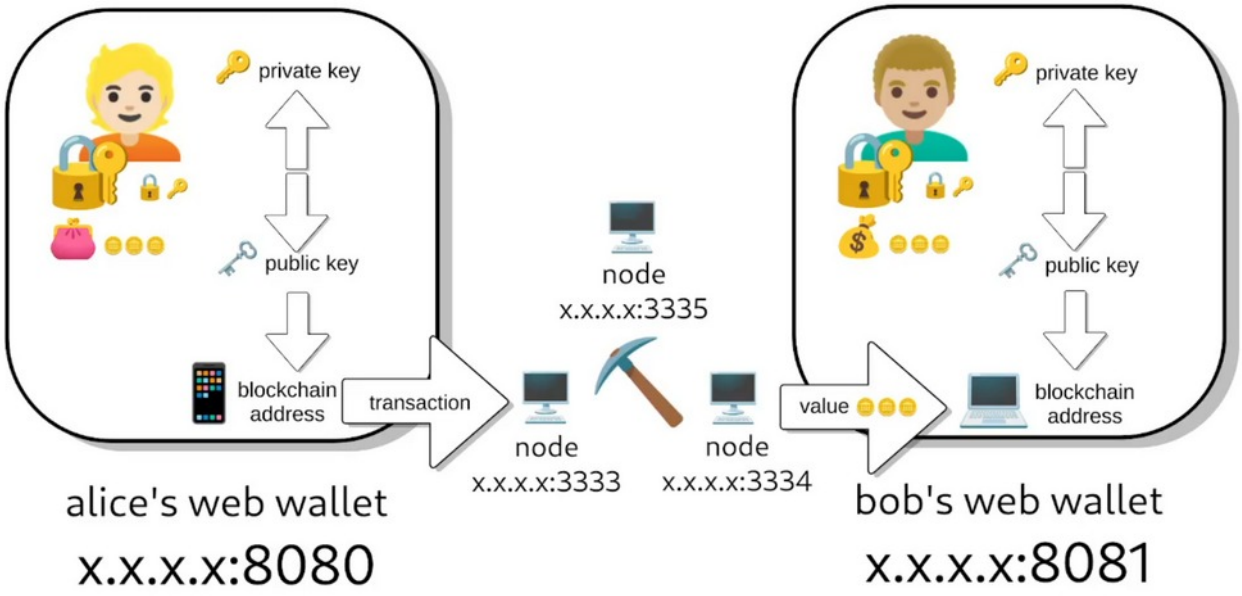
[BLOCKCHAIN_NODE.GO]

I would like to briefly explain what is happening here below. Basically, you are starting a web server for a web application that is currently only responding to a GET request in the root directory. You specify the function to be executed with `HandleFunc` from the `http` package. In your case, this is a HelloWorld function that receives an `http.ResponseWriter` and a pointer to an `http.Request` as arguments and outputs the string "Hello World" to the `http.ResponseWriter`.

Mission accomplished!

Code Lecture 28

Lecture 29 – Blockchain API - Your Goal Is Transaction Execution on the Blockchain



The aim of the first lesson in this section was to create an initial blockchain node that offers an API. There you want be able to start queries, display information or execute transactions using various requests.

[BLOCKCHAIN_NODE.GO TOP]

In this lecture you learn how to provide the blockchain in JSON instead of a “Hello World” string. For that, whenever you access the blockchain, it should already be available. Let's start by caching the blockchain completely.

```
var cache map[string]*blockchain.Blockchain =
make(map[string]*blockchain.Blockchain)
```

This creates a map and makes a complete blockchain available under a string.

[ABOVE HELLOWORLD]

```
func (bcn *BlockchainNode) GetBlockchain() *blockchain.Blockchain
{
     bc, ok := cache["blockchain"]
     if !ok {
          minerWallet := wallet.NewWallet()
          bc =
blockchain.NewBlockchain(minerWallet.BlockchainAddress(),
bcn.Port())
     }
     return bc
}
```

[BLOCKCHAIN.GO]

```
type Blockchain struct {
     transactionPool   []*Transaction
     chain             []*Block
     blockchainAddress string
     port              uint16
}

func NewBlockchain(blockchainAddress string, port uint16)
*Blockchain {
     b := &Block{}
     bc := new(Blockchain)
```

```
	bc.blockchainAddress = blockchainAddress
	bc.CreateBlock(0, b.Hash())
	bc.port = port
	return bc
}
```

This is a simple way to ensure that a port number becomes part of every blockchain.

[BLOCKCHAIN_NODE.GO CONTINUE]

```
func (bcn *BlockchainNode) GetBlockchain() *block.Blockchain {
	bc, ok := cache["blockchain"]
	if !ok {
		minerWallet := wallet.NewWallet()
		bc = block.NewBlockchain(minerWallet.BlockchainAddress(),
bcn.Port())
		cache["blockchain"] = bc
		log.Printf("Public Key %v", minerWallet.PublicKeyStr())
		log.Printf("Private Key %v", minerWallet.PrivateKeyStr())
		log.Printf("Blockchain Address %v",
minerWallet.BlockchainAddress())
	}
	return bc
}
```

A lot is happening here now. First, you check whether a blockchain called blockchain is stored in your cache or not. If not, simply create one. In your demonstration example, you are using a blockchain that you assume your miner identity holds. And you just place it in the cache!

So that you later know who you are dealing with here, you issue the private and public keys together with the blockchain address. This is quite simply a security catastrophe. Remember that this is a learning course and this data is only shown here for illustrative purposes.

```
func (bcn *BlockchainNode) HelloWorld(w http.ResponseWriter, r
*http.Request) {
	io.WriteString(w,"Hello World")
}
```

Now we want to create a method that is called when someone sends a GET request to the root directory of our blockchain node. That's fairly simple in GO.

```
func (bcn *BlockchainNode) GetChain(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodGet:
          bc := bcn.GetBlockchain()
     default:
          log.Printf("ERROR: Invalid HTTP method")
     }
}
```

At this point, however, we want to output data in JSON so that the user, or in this case your browser, can at least do something with the data. In blockchain.go, we need to implement a corresponding MarshalJSON method.

[BLOCKCHAIN.GO BELOW NEWBLOCKCHAIN]

```
func (bc *Blockchain) MarshalJSON() ([]byte, error) {
     return json.Marshal(struct {
          Blocks []*Block `json:"chain"`
     }{
          Blocks: bc.chain,
     })
}
```

At this point, you are already using MarshalJSON as a method to convert your blocks into JSON. But your block has a `previousHash` of type `[32]byte`. This is difficult to read. In [BLOCKCHAIN/BLOCK.GO MARSHALJSON], simply change this to type String.

[BLOCKCHAIN/BLOCK.GO MARSHALJSON]

```
func (b *Block) MarshalJSON() ([]byte, error) {
```

```
    return json.Marshal(struct {
        Timestamp    int64          `json:"timestamp"`
        Nonce        int            `json:"nonce"`
        PreviousHash string         `json:"previous_hash"`
        Transactions []*Transaction `json:"transactions"`
    }{
        Timestamp:    b.timestamp,
        Nonce:        b.nonce,
        PreviousHash: fmt.Sprintf("%x", b.previousHash),
        Transactions: b.transactions,
    })
}
```

Now you can properly marshal your blockchain.

[BLOCKCHAIN_NODE.GO CONTINUE]

```
func (bcn *BlockchainNode) GetChain(w http.ResponseWriter, r
*http.Request) {
    switch r.Method {
    case http.MethodGet:
        w.Header().Add("Content-Type", "application/json")
        bc := bcn.GetBlockchain()
        m, _ := bc.MarshalJSON()
        io.WriteString(w, string(m[:]))
    default:
        log.Printf("ERROR: Invalid HTTP method")
    }
}
```

You will output the whole JSON thing as a response to a GET request. To do this, you need a corresponding header, the data as JSON and you output everything as a simple string. Let the browser see what it makes of it! Last but not least, you can add the request handler to the route of your web server with changing HelloWorld to bcn.GetChain:

[BLOCKCHAIN_NODE.GO CONTINUE]

```
func (bcn *BlockchainNode) Run() {
     http.HandleFunc("/", bcn.GetChain)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}
```

[RUN]

[CHECK TERMINAL LOG]

[OPEN BROWSER FF LOCALHOST:3333]

Please note that Firefox offers an optional view for JSON here. Either use Firefox as well, or install an extension such as JSON Formatter in Chrome, which offers corresponding formatting.

Chrome Extension JSON Formatter

From now on your blockchain node will output your complete blockchain as JSON whenever you request it and at the same time shows private key, public key and blockchainaddress of the running node where the information came from on the terminal.

All that does not really make sense in a real-world scenario, but remember that you're doing this for demonstration purposes and to learn a bit about the inner workings and structure of both the blockchain itself and blockchain applications.

Code Lecture 29

Lecture 30 – UI Server: Create a Server Providing User Interface To Wallets

Now you have at least a rudimentary blockchain node that can deliver its services as JSON. Now concentrate on creating a wallet for the time being.

[CREATE FOLDER WALLET_SERVER AND WITHIN]

[MAIN.GO]

```
package main
```

[WALLET_SERVER.GO]

```
package main
```

[SUBFOLDER TEMPLATES AND WITHIN INDEX.HTML]

```
<!DOCTYPE html>
<html>
<head>
        <title>Styled Page</title>
          <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
     <h1>Hello World</h1>
</body>
</html>
```

[WALLET_SERVER.GO]

```
const pathToTemplateDir = "templates"

type WalletServer struct {
     port    uint16
     gateway string
}

func NewWalletServer(port uint16, gateway string) *WalletServer {
     return &WalletServer{port, gateway}
}
```

```
func (ws *WalletServer) Port() uint16 {
     return ws.port
}

func (ws *WalletServer) Gateway() string {
     return ws.gateway
}

func (ws *WalletServer) Index(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodGet:
          t, _ := template.ParseFiles(path.Join(pathToTemplateDir,
"index.html"))
          t.Execute(w, "")
     default:
          log.Printf("ERROR: Invalid HTTP method")
     }
}

func (ws *WalletServer) Run() {
     http.HandleFunc("/", ws.Index)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(ws.Port(
))), nil))
}
```

[MAIN.GO]

```
package main

func init() {
     log.SetPrefix("Wallet Server: ")
}
```

```
func main() {
     port := flag.Uint("port", 8080, "TCP Port Number for Online
wallet")
     gateway := flag.String("gateway", "http://127.0.0.1:3333",
"Blockchain Gateway")
     flag.Parse()


     app := NewWalletServer(uint16(*port), *gateway)
     log.Println("Starting wallet server on port:", *port, "using
blockchain node", *gateway, "as gateway")
     app.Run()
}
```

[RUN]

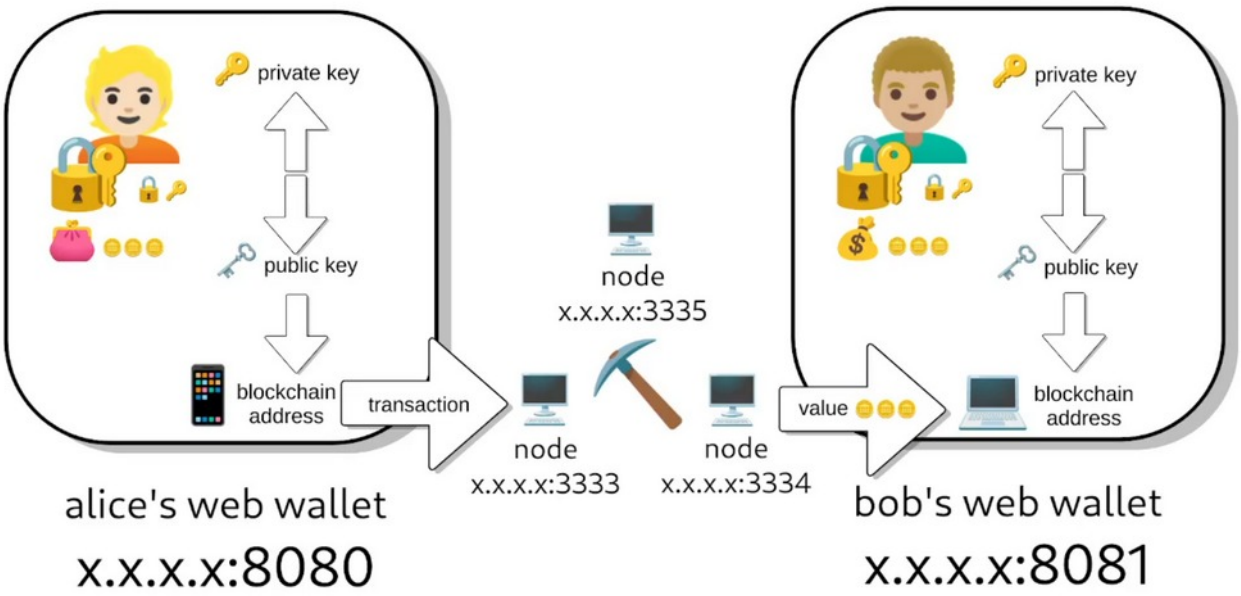
[BROWSER LOCALHOST:8080]

As you see your little online wallet setup seems to work out of the box. In this lesson you learned how to start a small web server in Go and then make it to deliver a simple HTML template.

Code Lecture 30

Lecture 31 – ... And Now Create a Simple Web Frontend for the Wallets

At the end of this course, the user should be able to connect to a wallet, create a transaction, the wallet forwards the transaction to a blockchain node, and returns the result to the wallet, which presents it to the user in the browser.



You have already started a web server for your wallet providing a first user interface, now take care of a simple web frontend that is displayed to the users in their browser. Basically a website that allows the user to enter and send transaction requests.

[TEMPLATES/INDEX.HTML]

```
<!DOCTYPE html>
<html lang="en">
<html>
<head>
        <title>Wallet</title>
          <link rel="stylesheet" type="text/css" href="styles.css">
          <!--
          <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min
.js"></script>
          <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.7.1/jquery.mi
n.js"></script>
          <script
src="https://cdn.jsdelivr.net/npm/jquery@3.7.1/dist/jquery.min.js"
></script>
          <script
src="https://cdn.bootcdn.net/ajax/libs/jquery/3.7.1/jquery.min.js"
></script>
          -->
```

```
        <script
src="https://ajax.microsoft.com/ajax/jquery/jquery-
3.7.1.min.js"></script>
</head>
<body>
    <h1>Hello World</h1>
</body>
</html>
```

I am a big fan of vanilla JavaScript and usually refrain from using external JavaScript libraries in my courses. But here I want you to concentrate fully on creating and understanding the blockchain-related content. I therefore rely on the use of JQuery to avoid having to program the AJAX-related processes myself. That would go beyond the scope of this course and would only distract from the essentials.

[TEMPLATES/INDEX.HTML]

```
<!DOCTYPE html>
<html lang="en">
<html>
<head>
       <title>Wallet</title>
         <!--
         <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min
.js"></script>
         <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.7.1/jquery.mi
n.js"></script>
         <script
src="https://cdn.jsdelivr.net/npm/jquery@3.7.1/dist/jquery.min.js"
></script>
         <script
src="https://cdn.bootcdn.net/ajax/libs/jquery/3.7.1/jquery.min.js"
></script>
         -->
```

```
        <script
src="https://ajax.microsoft.com/ajax/jquery/jquery-
3.7.1.min.js"></script>
</head>
<body style="background-color: darkslategray; color:beige">
    <div>
        <h3>Wallet</h3>
        <span style="font-size:larger;"
id="wallet_amount">0</span>
        <button id="reload_wallet">Reload</button>

        <p>Public Key</p>
        <textarea id="public_key" rows="2" cols="63"></textarea>

        <p>Private Key</p>
        <textarea id="private_key" rows="1" cols="63"></textarea>

        <p>Blockchain Address</p>
        <textarea id="blockchain_address" rows="1"
cols="63"></textarea>
    </div>

    <div>
        <h3>Send Value</h3>
        <div>
            <p>
                Address <input id="recipient_blockchain_address"
size="48" type="text">
            </p>
            <p>
                Amount <input id="send_amount" size="5"
type="text">
                <button id="send_money_button">Send</button>
            </p>
        </div>
    </div>
```

```
</body>
</html>
```

Creating the user interface is basically the same old hard work and perhaps a web developer will do it for you. In this course we will limit it to the bare essentials, but I would like it to be reasonably readable even if we have two wallets on the screen at the same time later in the final demonstration.

Code Lecture 31

Lecture 32 – jQuery and AJAX Come Into Play. Wire Your UI and Wallets With By of JSON

[BROWSER LOCALHOST:8080]

This could basically be the website that opens after you have logged in to your online wallet and clicked on "send" in the main menu. You are required to enter your public key, private key and blockchain address, but with a proper online wallet this should of course be done in the background.

And this is exactly what you will implement in this lesson by obtaining this data with jQuery and

displaying it here. Before you can do that, you need to do another thing. To access the wallet data, you need to make it available on the wallet server using an API. This means you need a method `Wallet()` that outputs the data in JSON and you need a route `/wallet` that calls the that method on request.

[WALLET_SERVER.GO]

```
func (ws *WalletServer) Wallet(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodPost:
          w.Header().Add("Content-Type", "application/json")
          myWallet := wallet.NewWallet()
          m, _ := myWallet.MarshalJSON()
     }
```

```
}
```

[WALLET.GO BOTTOM]

```
func (w *Wallet) MarshalJSON() ([]byte, error) {
     return json.Marshal(struct {
          PrivateKey        string `json:"private_key"`
          PublicKey         string `json:"public_key"`
          BlockchainAddress string `json:"blockchain_address"`
     }{
          PrivateKey:        w.PrivateKeyStr(),
          PublicKey:         w.PublicKeyStr(),
          BlockchainAddress: w.BlockchainAddress(),
     })
}
```

[WALLET_SERVER.GO]

```
func (ws *WalletServer) Wallet(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodPost:
          w.Header().Add("Content-Type", "application/json")
          myWallet := wallet.NewWallet()
          m, _ := myWallet.MarshalJSON()
          io.WriteString(w, string(m[:]))
     default:
          w.WriteHeader(http.StatusBadRequest)
          log.Println("ERROR: Invalid HTTP method")
     }
}

func (ws *WalletServer) Run() {
     http.HandleFunc("/", ws.Index)
```

```
    http.HandleFunc("/wallet", ws.Wallet)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(ws.Port(
))), nil))
}
```

I hope it becomes clear that whenever you call this method, a new wallet is created and the corresponding data, public key, private key and blockchain address are sent as JSON to the calling instance. However, the data is not stored anywhere, which in turn means that it is dropped with every subsequent reload of your frontend.

This also makes no sense in a real-world scenario, but is completely sufficient for a simple demonstration like this. Please also note that the data of the user wallets is not identical to that of the miners. This will become important later.

There are very good options for making all possible data types available from Go in HTML templates. For example, you can put the data into sessions and cookies or deliver it to the template in the form of maps, where you can use special expressions to retrieve it.

The problem with this is that it is quite complex, requires the import of external packages and many sub-functions to convert the required data types and all that takes us too far off track. I have therefore decided to implement this in jQuery.

[https://jquery.com/]

If you have no experience at all with jQuery, I must refer you to the documentation on the jquery.com website. But using jQuery is not difficult, you probably won't have much trouble following the course if you look at the implementation of the code.

[WALLET_SERVER/TEMPLATES/INDEX.HTML]

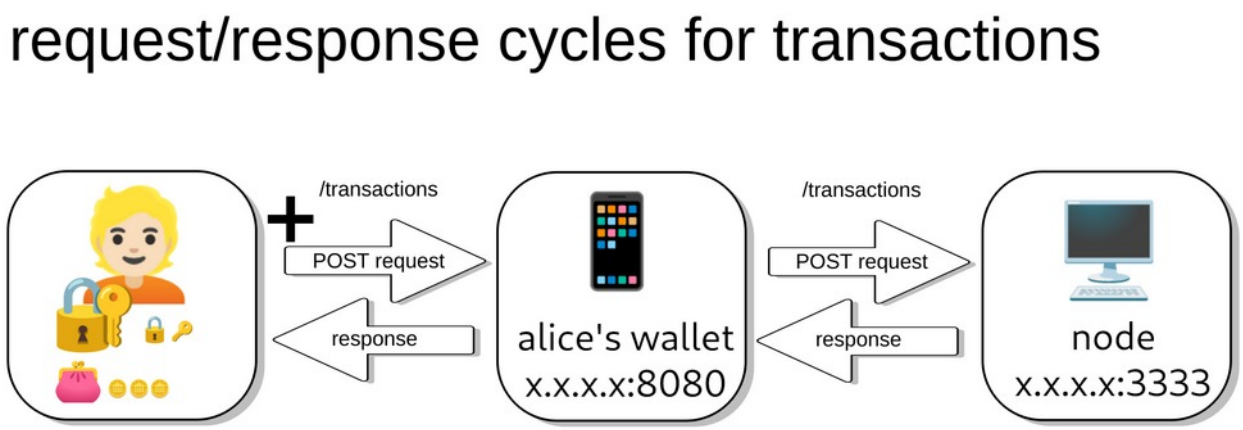
```
<script>
        $(function () {
            $.ajax({
                url: '/wallet',
                type: 'POST',
                success: function (response) {
                    $('#public_key').val(response['public_key']);
                    $
('#private_key').val(response['private_key']);
                    $
('#blockchain_address').val(response['blockchain_address']);
                    console.info(response);
```

```
            },
            error: function(error) {
                console.error(error);
            }
        });
    });
</script>
```

[BROWSER LOCALHOST:8080, F12, CONSOLE]

Code Lecture 32

Lecture 33 – User Interface to Wallet: Incoming Transaction, Prepare for Processing!



At the end of this course, you will be able to send a value from wallet A to wallet B. You will also have learned more about the mechanisms working within blockchains, but in this section we will focus almost exclusively on the wallet.

The request/response cycle from the user to the wallet, from the wallet to the blockchain node, from the blockchain node back to the wallet and from there to the user is quite complex. You will therefore program it in smaller sections. Divide et imperat, as the ancient Romans said, in other words divide and conquer.

[WALLET_SERVER/TEMPLATES/INDEX.HTML]

```
    $('#send_money_button').click(function () {
            let confirm_text = 'Are you sure to send?';
```

```
                let confirm_result = confirm(confirm_text);
                if (confirm_result !== true) {
                    alert('Canceled');
                    return
                }
```

[LOCALHOST:8080 RELOAD, TEST]

[WALLET_SERVER/TEMPLATES/INDEX.HTML CONTINUE]

```
                let transaction_data = {
                    'sender_private_key': $
('#private_key').val(),
                    'sender_blockchain_address': $
('#blockchain_address').val(),
                    'recipient_blockchain_address': $
('#recipient_blockchain_address').val(),
                    'sender_public_key': $('#public_key').val(),
                    'value': $('#send_amount').val(),
                };

                $.ajax({
                    url: '/transaction',
                    type: 'POST',
                    contentType: 'application/json',
                    data: JSON.stringify(transaction_data),
                    success: function (response) {
                        console.info(response);
                        if (response.message == 'fail') {
                            alert('Send fail')
                        } else {
                            alert('Send success');
                        }
                    },
                    error: function (response) {
```

```
                    console.error(response);
                    alert('Send failed');
                }
            })
```

But that was only the first half of our creation of the request that is sent to the wallet. Now the wallet server must also accept this POST request. This means you need another route/transaction and you need a handler. Start with the route, then the handler.

[WALLET_SERVER.GO]

```
func (ws *WalletServer) Run() {
     http.HandleFunc("/", ws.Index)
     http.HandleFunc("/wallet", ws.Wallet)
     http.HandleFunc("/transaction", ws.CreateTransaction )


log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(ws.Port(
))), nil))
}
```

[ABOVE THAT]

```
func (ws *WalletServer) CreateTransaction(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodPost:
          w.WriteHeader(http.StatusOK)
          io.WriteString(w, "Something")
     default:
          w.WriteHeader(http.StatusBadRequest)
          log.Println("ERROR: Invalid HTTP Method")
     }
}
```

[RESTART, LOCALHOST:8080 RELOAD, TEST WITH F12 CONSOLE]

To pass through status messages and output them formatted in JSON, you need a tool that accepts strings, marshals them as JSON and returns the result as a slice of byte. This is not difficult.

[CREATE UTILS/JSON.GO]

```
package utils

func JsonStatus(message string) []byte {
	m, _ := json.Marshal(struct {
		Message string `json:"message"`
	}{
		Message: message,
	})
	return m
}
```

[WALLET_SERVER.GO CONTINUE]

```
func (ws *WalletServer) CreateTransaction(w http.ResponseWriter, r
*http.Request) {
	switch r.Method {
	case http.MethodPost:
		w.WriteHeader(http.StatusOK)
		io.WriteString(w, "Something")
		io.WriteString(w, string(utils.JsonStatus("Wubba Lubba
Dub Dub")))
	default:
		w.WriteHeader(http.StatusBadRequest)
		log.Println("ERROR: Invalid HTTP Method")
	}
}
```

[RESTART, LOCALHOST:8080 RELOAD, TEST WITH F12 CONSOLE]

Code Lecture 33

Lecture 34 – Interpreting JSON: Teach Your Wallet a New Trick

[WALLET_SERVER.GO CONTINUE]

```
func (ws *WalletServer) CreateTransaction(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodPost:
          io.WriteString(w, string(utils.JsonStatus("Wubba Lubba
Dub Dub")))
          decoder := json.NewDecoder(r.Body)
     default:
          w.WriteHeader(http.StatusBadRequest)
          log.Println("ERROR: Invalid HTTP Method")
     }
}
```

[VISIT https://pkg.go.dev/std]

[CHECKOUT ENCODING/JSON/TYPES/DECODER/EXAMPLE]

[WALLTET/TRANSACTION.GO]

```
package wallet

type TransactionRequest struct {
     SenderPrivateKey           *string `json:"sender_private_key"`
     SenderBlockchainAddress    *string
`json:"sender_blockchain_address"`
     RecipientBlockchainAddress *string
`json:"recipient_blockchain_address"`
     SenderPublicKey            *string `json:"sender_public_key"`
     Value                      *string `json:"value"`
}
```

[WALLET_SERVER.GO CONTINUE]

```
func (ws *WalletServer) CreateTransaction(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodPost:
          decoder := json.NewDecoder(r.Body)
          var t wallet.TransactionRequest
          err := decoder.Decode(&t)
          if err != nil {
               log.Printf("ERROR: %v", err)
               io.WriteString(w, string(utils.JsonStatus("fail")))
               return
          }
     default:
          w.WriteHeader(http.StatusBadRequest)
          log.Println("ERROR: Invalid HTTP Method")
     }
}
```

[WALLTET/TRANSACTION.GO CONTINUE]

```
func (tr *TransactionRequest) Validate() bool {
     // This is a basic simplified validation
     // checking for existence of values only
     // not checking for plausibility
     if tr.SenderPrivateKey == nil ||
          tr.SenderBlockchainAddress == nil ||
          tr.RecipientBlockchainAddress == nil ||
          tr.SenderPublicKey == nil ||
          tr.Value == nil {
          return false
     }
```

```
	return true
}
```

[WALLET_SERVER.GO CONTINUE]

```
func (ws *WalletServer) CreateTransaction(w http.ResponseWriter, r
*http.Request) {
	switch r.Method {
	case http.MethodPost:
		decoder := json.NewDecoder(r.Body)
		var t wallet.TransactionRequest
		err := decoder.Decode(&t)
		if err != nil {
			log.Printf("ERROR: %v", err)
			io.WriteString(w, string(utils.JsonStatus("fail")))
			return
		}
		if !t.Validate() {
			log.Println("ERROR: missing field(s)")
			io.WriteString(w, string(utils.JsonStatus("fail")))
			return
		}

		fmt.Println(*t.SenderPublicKey)
		fmt.Println(*t.SenderBlockchainAddress)
		fmt.Println(*t.SenderPrivateKey)
		fmt.Println(*t.RecipientBlockchainAddress)
		fmt.Println(*t.Value)

	default:
		w.WriteHeader(http.StatusBadRequest)
		log.Println("ERROR: Invalid HTTP Method")
	}
}
```

[RESTART, LOCALHOST:8080 RELOAD, TEST WITH F12 CONSOLE AND TERMINAL OUTPUT]

Code Lecture 34

Lecture 35 – Take the ECDSA-Related String Data and Convert It Into a Suitable Data Types

[WALLET_SERVER.GO CONTINUE]

```
            fmt.Println(*t.SenderBlockchainAddress)
            fmt.Println(*t.RecipientBlochchainAddress)
            fmt.Println(*t.Value)

            fmt.Println(len(*t.SenderPublicKey))
            fmt.Println(len(*t.SenderPrivateKey))
```

[RESTART, LOCALHOST:8080 RELOAD, TEST WITH F12 CONSOLE AND TERMINAL OUTPUT]

[WALLET.GO CHECK PRIVATEKEYSTR AND PUBLICKEYSTR]

This is how you created the original strings. However, if you want to process this data, which you receive as a transaction request in the form of strings, you cannot do anything with it. You need a small untility that converts the strings of a private key and a public key back into the data types that you need in a struct for a transaction, signature, etc.!

[UTILS/ECDSA.GO]

```
func String2BigIntTuple(s string) (big.Int, big.Int) {
     bx, _ := hex.DecodeString(s[:64])
     by, _ := hex.DecodeString(s[64:])

     var bix big.Int
     var biy big.Int
```

```
	_ = bix.SetBytes(bx)
	_ = biy.SetBytes(by)

	return bix, biy
}

func String2PublicKey(s string) *ecdsa.PublicKey {
	x, y := String2BigIntTuple(s)
	return &ecdsa.PublicKey{Curve: elliptic.P256(), X: &x, Y: &y}
}

func String2PrivateKey(s string, publicKey *ecdsa.PublicKey)
*ecdsa.PrivateKey {
	b, _ := hex.DecodeString(s[:])
	var bi big.Int
	_ = bi.SetBytes(b)
	return &ecdsa.PrivateKey{PublicKey: *publicKey, D: &bi}
}
```

[UTILS/ECDSA.GO ABOVE STRING2PUBLICKEY]

```
func String2Signature(s string) *Signature {
	x, y := String2BigIntTuple(s)
	return &Signature{&x, &y}
}
```

[EDIT WALLET.GO]

```
func (w *Wallet) PrivateKeyStr() string {
	return fmt.Sprintf("%064x", w.privateKey.D.Bytes())
}

func (w *Wallet) PublicKeyStr() string {
	return fmt.Sprintf("%064x%064x", w.publicKey.X.Bytes(),
w.publicKey.Y.Bytes())
```

[WALLET_SERVER.GO CONTINUE]

```
            fmt.Println(len(*t.SenderPublicKey))
            fmt.Println(len(*t.SenderPrivateKey))

                        publicKey :=
utils.String2PublicKey(*t.SenderPublicKey)
            privateKey :=
utils.String2PrivateKey(*t.SenderPrivateKey, publicKey)
            value, err := strconv.ParseFloat(*t.Value, 32)
            if err != nil {
                  log.Println("ERROR: parse error")
                  io.WriteString(w, string(utils.JsonStatus("fail")))
                  return
            }
            value32 := float32(value)

            fmt.Println(publicKey)
            fmt.Println(privateKey)
            fmt.Printf("%.1f", value32)
```

[RESTART, LOCALHOST:8080 RELOAD, TEST WITH F12 CONSOLE AND CHECK TERMINAL OUTPUT]

Code Lecture 35

Lecture 36 – Next Step: Sending a Transaction Request From the Wallet Server to the Blockchain Node

[LOOK AT WALLET/TRANSACTION.GO]

```
...
                        type TransactionRequest struct {
    SenderPrivateKey           *string `json:"sender_private_key"`
```

```
    SenderBlockchainAddress    *string
`json:"sender_blockchain_address"`
    RecipientBlockchainAddress *string
`json:"recipient_blockchain_address"`
    SenderPublicKey            *string `json:"sender_public_key"`
    Value                      *string `json:"value"`
}

func (tr *TransactionRequest) Validate() bool {
    // This is a basic simplified validation
    // checking for existence of values only
    // not checking for plausibility
    if tr.SenderPrivateKey == nil ||
        tr.SenderBlockchainAddress == nil ||
        tr.RecipientBlockchainAddress == nil ||
        tr.SenderPublicKey == nil ||
        tr.Value == nil {
        return false
    }
    return true
}
```

request/response cycles for transactions

If you take a look in package `wallet` at `transaction.go` you see that you have already a type for transaction requests as well as a `validate()` method. The five pieces of data you collect and strap together here are

- sender private key
- sender blockchain address
- recipient blockchain address

- sender public key
- value

That is the transaction request you need to create a signature internally. But especially the first value here, sender private key, you want to keep a secret and not transfer to any public blockchain node. If you use your bank card at an ATM you also don’t trumpet your pin out into the world, or?

On Blockchain node’s side side you need also 5 pieces of information but you omit the private key and use a signature instead for a specific transaction instead which proofes that you and only you signed this transaction request and you had the right to do so.

Actually you have all the data you need to create such a transaction request on blockchain node’s side but you need to create the corresponding data type and do a brief check for completeness for your blockchain node.

[BLOCKCHAIN/TRANSACTION.GO BOTTOM]

```
type TransactionRequest struct {
     SenderBlockchainAddress    *string
`json:"sender_blockchain_address"`
     RecipientBlockchainAddress *string
`json:"recipient_blockchain_address"`
     SenderPublicKey            *string  `json:"sender_public_key"`
     Value                      *float32 `json:"value"`
     Signature                  *string  `json:"signature"`
}

func (tr *TransactionRequest) Validate() bool {
     // This is a basic simplified validation only
     // checking for existence of values only
     // not checking for plausibility
     if tr.SenderBlockchainAddress == nil ||
          tr.RecipientBlockchainAddress == nil ||
          tr.SenderPublicKey == nil ||
          tr.Value == nil ||
          tr.Signature == nil {
          return false
     }
     return true
```

```
}
```

[WALLET_SERVER.GO CREATETRANSACTION CONTINUE]

```
...
            value32 := float32(value)

            fmt.Println(publicKey)
            fmt.Println(privateKey)
            fmt.Printf("%.1f", value32)

            w.Header().Add("Content-Type", "application/json")

            transaction := wallet.NewTransaction(privateKey,
publicKey,
                *t.SenderBlockchainAddress,
*t.RecipientBlockchainAddress, value32)
            signature := transaction.GenerateSignature()
            signatureStr := signature.String()

            bt := &blockchain.TransactionRequest{
                SenderBlockchainAddress:
t.SenderBlockchainAddress,
                RecipientBlockchainAddress:
t.RecipientBlockchainAddress,
                SenderPublicKey:            t.SenderPublicKey,
                Value:                      &value32, Signature:
&signatureStr,
            }
            m, _ := json.Marshal(bt)
            buf := bytes.NewBuffer(m)

            resp, _ := http.Post(ws.Gateway()+"/transactions",
"application/json", buf)
            if resp.StatusCode == 201 {
```

```
                io.WriteString(w,
string(utils.JsonStatus("success")))
                return
            }
            io.WriteString(w, string(utils.JsonStatus("fail")))
```

[GO RUN WALLET_SERVER.GO]

[ERROR]

The explanation is relatively simple. The problem is that we have no error handling.

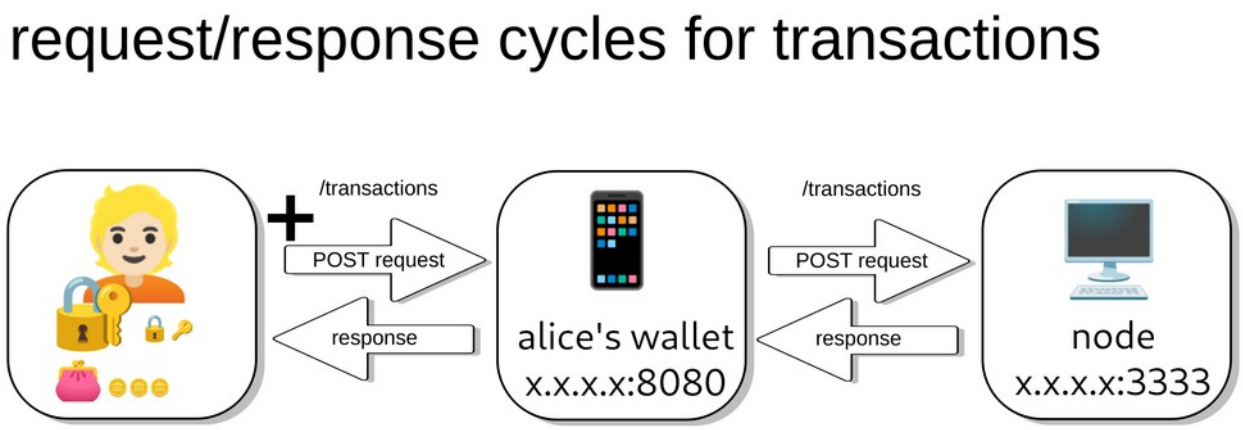
```
http.Post(ws.Gateway()+"/transactions", "application/json", buf)
```

returns an error message, but as is so often the case, we do not evaluate it. Basically, the problem here is the following, we are trying to post something, but there is no server that accepts and receives our POST request. You are welcome to take care of this, but then you not only have to evaluate every single portential error message, but also pass errors that occur in sub-functions such as the continuous marshaling and unmarshaling of JSON to jQuery if necessary. This is too much work for me for this course and that's why I'm neglecting it.

When you finish the next lesson, your blockchain node should also have a route /transactions that accepts and handles POST requests. If you then start the node first and then your wallet, and start a transaction request on the frontend, this error should no longer occur.

Lecture 37 – Implement an API on the Blockchain Node's Side to Receive Transaction Requests

You have successfully established a connection between the user and their online wallet, and from the online wallet to the Node. Now a user can create a transaction request, send the necessary pieces



data

- public key
- signature
- sender blockchain address
- recipient blockchain address
- value

to the blockchain node and they are available in Go in the form of the correct data types.

Let's now move on to the implementation of the second part of a transaction's way into the transaction pool and into a new block eventually: the transaction must not be transmitted from the wallet server to the blockchain node only, there is work to be done on both sides. Time to start on the receiving side, namely the blockchain node, and create a route and handler here.

[BLOCKCHAIN_NODE.GO ABOVE RUN()]

```
func (bcn *BlockchainNode) Transactions(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodGet:
          // TODO
     case http.MethodPost:
          decoder := json.NewDecoder(r.Body)
          var t blockchain.TransactionRequest
          err := decoder.Decode(&t)
```

```
            if err != nil {
                log.Printf("ERROR: %v", err)
                io.WriteString(w, string(utils.JsonStatus("fail")))
                return
            }
            if !t.Validate() {
                log.Println("ERROR: missing field(s)")
                io.WriteString(w, string(utils.JsonStatus("fail")))
                return
            }
            publicKey := utils.String2PublicKey(*t.SenderPublicKey)
            signature := utils.String2Signature(*t.Signature)

        default:
            log.Println("ERROR: Invalid HTTP method")
            w.WriteHeader(http.StatusBadRequest)
        }
}
```

[BLOCKCHAIN.GO ABOVE ADDTRANSACTION COPY&PASTE, EDIT TO CREATETRANSACTION]

```
func (bc *Blockchain) CreateTransaction(sender string, recipient string, value float32,
    senderPublicKey *ecdsa.PublicKey, s *utils.Signature) bool {
    isTransacted := bc.AddTransaction(sender, recipient, value, senderPublicKey, s)
    // TODO
    // sync
    return isTransacted
}
```

[BLOCKCHAIN_NODE.GO CONTINUE]

```
func (bcn *BlockchainNode) Transactions(w http.ResponseWriter, r
*http.Request) {
	switch r.Method {
	case http.MethodGet:
		// TODO
	case http.MethodPost:
		decoder := json.NewDecoder(r.Body)
		var t block.TransactionRequest
		err := decoder.Decode(&t)
		if err != nil {
			log.Printf("ERROR: %v", err)
			io.WriteString(w, string(utils.JsonStatus("fail")))
			return
		}
		if !t.Validate() {
			log.Println("ERROR: missing field(s)")
			io.WriteString(w, string(utils.JsonStatus("fail")))
			return
		}
		publicKey := utils.String2PublicKey(*t.SenderPublicKey)
		signature := utils.String2Signature(*t.Signature)
		bc := bcn.GetBlockchain()
		isCreated :=
bc.CreateTransaction(*t.SenderBlockchainAddress,
			*t.RecipientBlockchainAddress, *t.Value, publicKey,
signature)

		w.Header().Add("Content-Type", "application/json")
		var m []byte
		if !isCreated {
			w.WriteHeader(http.StatusBadRequest)
			m = utils.JsonStatus("fail")
		} else {
			w.WriteHeader(http.StatusCreated)
			m = utils.JsonStatus("success")
```

```
            }
            io.WriteString(w, string(m))
      default:
            log.Println("ERROR: Invalid HTTP Method")
            w.WriteHeader(http.StatusBadRequest)
      }
}
```

[BLOCKCHAIN_NODE.GO CONTINUE METHODGET]

```
func (bcn *BlockchainNode) Transactions(w http.ResponseWriter, r
*http.Request) {
      switch r.Method {
      case http.MethodGet:
            w.Header().Add("Content-Type", "application/json")
            bc := bcn.GetBlockchain()
            transactions := bc.TransactionPool()
      case http.MethodPost:
```

…

[BLOCKCHAIN.GO BELOW NEWBLOCKCHAIN]

```
func (bc *Blockchain) TransactionPool() []*Transaction {
      return bc.transactionPool
}
```

BLOCKCHAIN_NODE.GO CONTINUE]

```
func (bcn *BlockchainNode) Transactions(w http.ResponseWriter, r
*http.Request) {
      switch r.Method {
      case http.MethodGet:
            w.Header().Add("Content-Type", "application/json")
            bc := bcn.GetBlockchain()
            transactions := bc.TransactionPool()
```

```
            m, _ := json.Marshal(struct {
                Transactions []*block.Transaction
`json:"transactions"`
                Length       int                    `json:"length"`
            }{
                Transactions: transactions,
                Length:       len(transactions),
            })
            io.WriteString(w, string(m[:]))
      case http.MethodPost:
…

[AT THE BOTTOM ADD THE ROUTE]

func (bcn *BlockchainNode) Run() {
      http.HandleFunc("/", bcn.GetChain)
      http.HandleFunc("/transactions", bcn.Transactions)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}

[START WALLET_SERVER]
[BROWSER LOCALHOST:8080]
[START BLOCKCHAIN_NODE]
[BROWSER LOCALHOST:3333/transactions]
[PICTURE PAGE REQUEST/RESPONSE]

[TRANSACTION TO "Morty", VALUE 137]
[RELOAD  LOCALHOST:3333/transactions]
[TRANSACTION TO "Hagbard", VALUE 23]
[TRANSACTION TO "Arthur Dent", VALUE 42]
```

In the case that the code does not work like this for you now, I recommend that you go through it again line by line. If necessary, also check the output on the terminal. You may also need to examine

the many possible return values of type `error` that with recurring indifference and use of comma underscore have been discarded by me. It is entirely up to you if it is worth the effort. From the student's point of view, it is always a good idea to find bugs, get to the bottom of the causes and eliminate any problems found. That's basically how learning works. Alternatively, you can download the release for this lesson from github.com and continue working with that.

Code Lecture 37

Lecture 38 – Create Another API, This Time for Transaction Processing. It's About Mining.

At the moment it looks like this: you have a user who creates a transaction request on the front end. The wallet server accepts this and creates its own transaction request with a signature. The transaction data

- public key
- signature
- sender blockchain address
- recipient blockchain address
- value

is sent to the blockchain node, which checks it and places it in the transaction pool.
In this lesson, you will now create another API that processes all transactions in this pool when called and attaches them to the blockchain in a block. You will create a mining API.

[BLOCKCHAIN/BLOCKCHAIN.GO METHOD MINING]

This is the blockchain method that our API should be able to call. A large part of it is already finished, we just have to package it nicely now.

[BLOCKCHAIN_NODE.GO ABOVE RUN()]

```
func (bcn *BlockchainNode) Mine(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodGet:
          bc := bcn.GetBlockchain()
          isMined := bc.Mining()

          var m []byte
```

```
            if !isMined {
                w.WriteHeader(http.StatusBadRequest)
                m = utils.JsonStatus("fail")
            } else {
                m = utils.JsonStatus("success")
            }
            w.Header().Add("Content-Type", "application/json")
            io.WriteString(w, string(m))
        default:
            log.Println("ERROR: Invalid HTTP method")
            w.WriteHeader(http.StatusBadRequest)
        }
}

func (bcn *BlockchainNode) Run() {
        bcn.GetBlockchain().Run()

        http.HandleFunc("/", bcn.GetChain)
        http.HandleFunc("/transactions", bcn.Transactions)
        http.HandleFunc("/mine", bcn.Mine)


log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}
```

[START WALLET_SERVER]

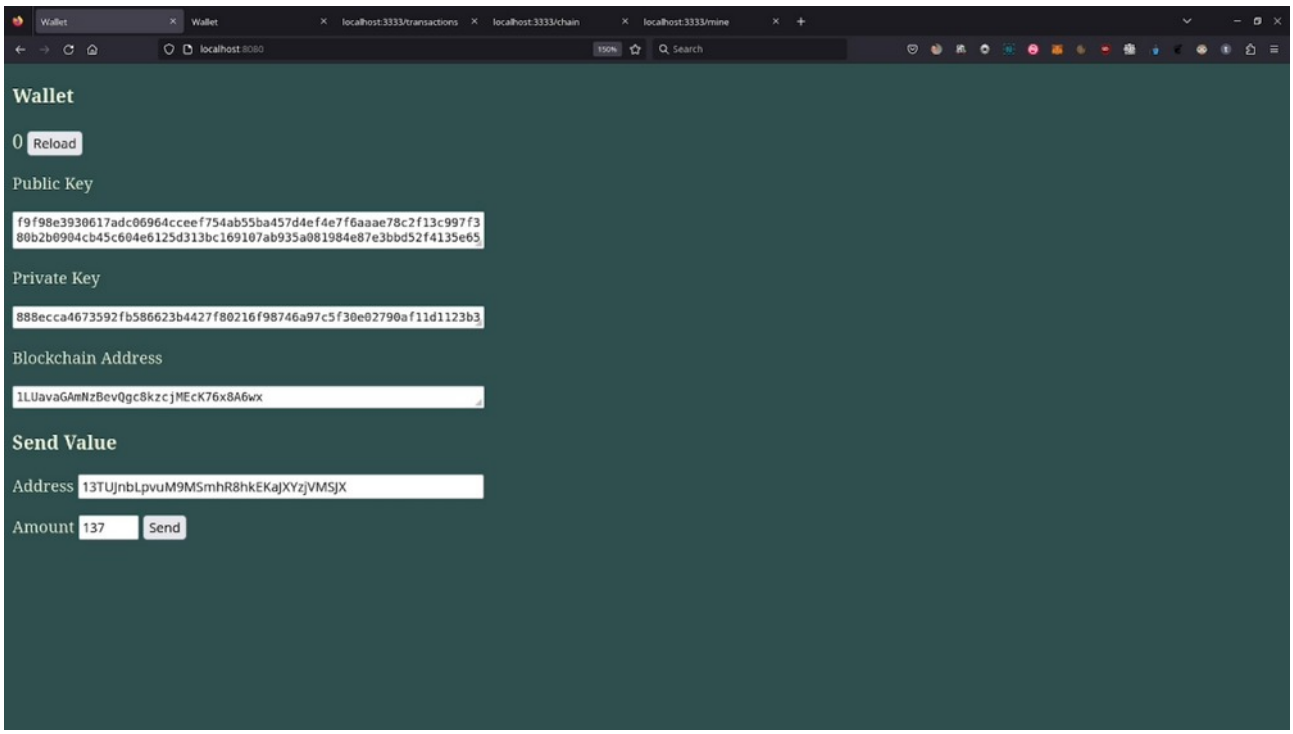
[START BLOCKCHAIN_NODE]

[2X BROWSER LOCALHOST:8080]

[BROWSER LOCALHOST:3333/TRANSACTIONS]

[BROWSER LOCALHOST:3333/CHAIN]

[BROWSER LOCALHOST:3333/MINE]



[TRANSACTION "A" TO "B", VALUE 137]

[RELOAD BROWSER LOCALHOST:3333/TRANSACTIONS]

[RELOAD BROWSER LOCALHOST:3333/CHAIN]

[TRANSACTION "A" TO "B", VALUE 23]

[RELOAD BROWSER LOCALHOST:3333/TRANSACTIONS]

[RELOAD BROWSER LOCALHOST:3333/CHAIN]

As you can see, three transactions were added to the last block in the blockchain. Two that were created by us and one that rewards the miner. Hence you have successfully created an API on the blockchain node's side. I think you should now automate the whole process in the following lesson.

Code Lecture 38

Lecture 39 – Mining Is Transaction Processing. Heigh-Ho, Heigh-Ho, It's off to Work We Go!

At the moment, mining is only carried out exactly once when we trigger it via the API. In reality, however, mining takes place constantly and the average block generation rate of around 1 block per 10 minutes is controlled via the difficulty in relation to the globally available mining power. This is too complex for our demo. We fake this - not the complete mining, but we simulate a minimum time to create a block.

[BLOCKCHAIN.GO]

```
const (
```

```
    MINING_DIFFICULTY = 3
    MINING_SENDER     = "BLOCKCHAIN REWARD SYSTEM (e.g. minting & fees)"
    MINING_REWARD     = 1.0
    MINING_TIMER_SEC  = 20
)
```

The simplest option here is a mutex lock. Mutex stands for mutual exclusion and the name is programmatic. Perhaps you have already worked with it. A lock is exactly what is now becoming part of the `blockchain` data type.

[BLOCKCHAIN.GO]

```
type Blockchain struct {
    transactionPool   []*Transaction
    chain             []*Block
    blockchainAddress string
    port              uint16
    mux               sync.Mutex
}
```

[BLOCKCHAIN.GO MINING()]

```
func (bc *Blockchain) Mining() bool {
    bc.mux.Lock()
    defer bc.mux.Unlock()

    if len(bc.transactionPool) == 0 {
        return false
    }

    bc.AddTransaction(MINING_SENDER, bc.blockchainAddress, MINING_REWARD, nil, nil)
    nonce := bc.ProofOfWork()
    previousHash := bc.LastBlock().Hash()
```

```
        bc.CreateBlock(nonce, previousHash)
        log.Println("action=mining, status=success")
        return true
}
```

You can see that this prevents empty blocks from being mined. With Bitcoin, for example, empty blocks are now a rarity, but they are not generally prohibited - and never have been. The fact that empty blocks may be created, but the effort is basically the same as getting the transaction fees for all transactions in a block, ensures that there is competition and that the miners are keen to include as many transactions with the highest possible transaction fees in a block. In our small demo, however, the empty blocks are now disturbing, but it was important to me to make you understand where you can intervene if you want to determine whether empty blocks should be able to exist or not.

```
func (bc *Blockchain) StartMining() {
        bc.Mining()
        _ = time.AfterFunc(time.Second*MINING_TIMER_SEC, bc.StartMining)
}
```

This now works in such a way that when the `StartMining()` method is called, it is always called again after 20 seconds. The mutex lock ensures that the `Mining()` method cannot be called again ahead of time. Imagine a case worker in the office who basically only needs seconds to process each task. But he locks the door and puts his feet up for a while for each job. He also simulates full employment though.

[BLOCKCHAIN_NODE.GO ABOVE RUN()]

```
func (bcn *BlockchainNode) StartMine(w http.ResponseWriter, r *http.Request) {
        switch r.Method {
        case http.MethodGet:
                bc := bcn.GetBlockchain()
                bc.StartMining()

                m := utils.JsonStatus("success")
                w.Header().Add("Content-Type", "application/json")
```

```
            io.WriteString(w, string(m))
        default:
            log.Println("ERROR: Invalid HTTP method")
            w.WriteHeader(http.StatusBadRequest)
        }
}

func (bcn *BlockchainNode) Run() {
        bcn.GetBlockchain().Run()

        http.HandleFunc("/", bcn.GetChain)
        http.HandleFunc("/transactions", bcn.Transactions)
        http.HandleFunc("/mine", bcn.Mine)
        http.HandleFunc("/mine/start", bcn.StartMine)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}
```

[START WALLET_SERVER]

[START BLOCKCHAIN_NODE]

[BROWSER LOCALHOST:3333/CHAIN]

[BROWSER LOCALHOST:3333/TRANSACTIONS]

2X [BROWSER LOCALHOST:8080]

[BROWSER LOCALHOST:3333/MINE/START]

In the current scenario and the setting that you only want to create blocks if there are transactions, you will now execute a transaction without calling /mine in the API. Mining has been started once and should now run in the background every 20 seconds.

[TRANSACTION "A" TO "B", VALUE 23]

[RELOAD BROWSER LOCALHOST:3333/TRANSACTIONS]

[RELOAD BROWSER LOCALHOST:3333/CHAIN]

I admit that this is not really an elegant solution. Basically, you have a process constantly running against a locked door that only opens every 20 seconds and then only to accept a single instruction. But it seems to work anyway. If this is too inconvenient for you now and you prefer to perform mining on demand, you can of course do that too. You now have the two API calls `/mine` and `/mine/start`.

Please note that once you have started automated mining in a session you cannot use the /mine route anymore to trigger manual mining. That is because the mutex lock block access to the mining function every twenty seconds.

Code Lecture 39

Lecture 40 – Check Your Address' Total Amount: Creating a Blockchain Node API

[START 2X BROWSER LOCALHOST:8080]

You have probably already noticed that the amount of current values for an address should be displayed at this point in the frontend. You must therefore be able to query this value at any time. Your blockchain node will have to offer this value as an API. This is not difficult and since you are here anyway, you can do it straight away.

[BLOCKCHAIN_NODE.GO]

```
func (bcn *BlockchainNode) Amount(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodGet:
          blockchainAddress :=
r.URL.Query().Get("blockchain_address")
          amount :=
bcn.GetBlockchain().CalculateTotalAmount(blockchainAddress)
```

[BLOCKCHAN.GO]

```
type AmountResponse struct {
     Amount float32 `json:"amount"`
}

func (ar *AmountResponse) MarshalJSON() ([]byte, error) {
```

```
	return json.Marshal(struct {
		Amount float32 `json:"amount"`
	}{
		Amount: ar.Amount,
	})
}
```

[BLOCKCHAIN NODE CONTINUE]

```
			ar := &block.AmountResponse{Amount: amount}
			m, _ := ar.MarshalJSON()

			w.Header().Add("Content-Type", "application/json")
			io.WriteString(w, string(m[:]))

		default:
			log.Println("ERROR: Invalid HTTP method")
			w.WriteHeader(http.StatusBadRequest)
		}
}


func (bcn *BlockchainNode) Run() {
	bcn.GetBlockchain().Run()
	http.HandleFunc("/", bcn.GetChain)
	http.HandleFunc("/transactions", bcn.Transactions)
	http.HandleFunc("/mine", bcn.Mine)
	http.HandleFunc("/mine/start", bcn.StartMine)
	http.HandleFunc("/amount", bcn.Amount)


log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}
```

[START WALLET_SERVER]

[START BLOCKCHAIN_NODE]

[2X BROWSER LOCALHOST:8080]

[BROWSER LOCALHOST:3333/AMOUNT?BLOCKCHAIN_ADDRESS=]

At the moment, nothing is displayed because our query does not contain an address. I will now copy the address of user B and execute a transaction. I will mine this transaction manually.

[TRANSACTION "A" TO "B", VALUE 137]

[BROWSER LOCALHOST:3333/MINE]

[BROWSER LOCALHOST:3333/AMOUNT?BLOCKCHAIN_ADDRESS=B]

As you can see, anyone who knows a valid blockchain address and has access to the API of a blockchain node can query the stored value. This is completely intentional. However, in the present case, where you want to display the value stored under an address in the online wallet, we have now put the cart before the horse. Let's tackle this in the next lesson.

Code Lecture 40

Lecture 41 – And Now an API Letting the Wallet Server Return the Total Amount on an Address

In this lesson, you will now ensure that your wallet server also has an API to display the total amount of an address. And for that it also needs a handler which can access the blockchain node’s API.

[WALLET_SERVER.GO]

```
func (ws *WalletServer) WalletAmount(w http.ResponseWriter, r
*http.Request) {
     switch r.Method {
     case http.MethodGet:
          blockchainAddress :=
r.URL.Query().Get("blockchain_address")
          endpoint := fmt.Sprintf("%s/amount", ws.Gateway())
```

```
        client := &http.Client{}
        bcnRequest, _ := http.NewRequest("GET", endpoint, nil)
        q := bcnRequest.URL.Query()
        q.Add("blockchain_address", blockchainAddress)
        bcnRequest.URL.RawQuery = q.Encode()

        bcnResponse, err := client.Do(bcnRequest)
        if err != nil {
             log.Printf("ERROR: %v", err)
             io.WriteString(w, string(utils.JsonStatus("fail")))
             return
        }

        w.Header().Add("Content-Type", "application/json")
        if bcnResponse.StatusCode == 200 {
             decoder := json.NewDecoder(bcnResponse.Body)
             var baresp blockchain.AmountResponse
             err := decoder.Decode(&baresp)
             if err != nil {
                  log.Printf("ERROR: %v", err)
                  io.WriteString(w,
string(utils.JsonStatus("fail")))
                  return
             }

             m, _ := json.Marshal(struct {
                  Message string  `json:"message"`
                  Amount  float32 `json:"amount"`
             }{
                  Message: "success",
                  Amount:  baresp.Amount,
             })
             io.WriteString(w, string(m[:]))
```

```
            } else {
                io.WriteString(w, string(utils.JsonStatus("fail")))
            }
        default:
            log.Printf("ERROR: Invalid HTTP Method")
            w.WriteHeader(http.StatusBadRequest)
        }
}

func (ws *WalletServer) Run() {
        http.HandleFunc("/", ws.Index)
        http.HandleFunc("/wallet", ws.Wallet)
        http.HandleFunc("/wallet/amount", ws.WalletAmount)
        http.HandleFunc("/transaction", ws.CreateTransaction)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(ws.Port(
))), nil))
}
```

[START WALLET_SERVER]

[START BLOCKCHAIN_NODE]

2X [BROWSER LOCALHOST:8080]

[BROWSER LOCALHOST:3333/MINE/START]

[BROWSER LOCALHOST:8080/WALLET/AMOUNT?BLOCKCHAIN_ADDRESS=]

[TRANSACTION "A" TO "B", VALUE 23]

[BROWSER LOCALHOST:8080/WALLET/AMOUNT?BLOCKCHAIN_ADDRESS=B]

I hope it is clear that your wallet server does nothing other than providing this API here. The data displayed comes from a query to the API of the blockchain node. Now you make all this available cleanly and neatly in the user interface, namely your frontend. I will show you how this works in the next lesson.

Code Lecture 41

Lecture 42 – Eventually, Display the Total Amount Stored on an Address in the User Interface

As always, work on the front end takes place in HTML, more precisely in jQuery.

[WALLET_SERVER/TEMPLATES/INDEX.HTML]

[WITHIN SCRIPT TAG]

```
 function reload_amount() {
                let data = {'blockchain_address': $
('#blockchain_address').val()}
                $.ajax({
                    url: '/wallet/amount',
                    type: 'GET',
                    data: data,
                    success: function (response) {
                        let amount = response['amount'];
                        $('#wallet_amount').text(amount);
                        console.info(amount)
                    },
                    error: function(error) {
                        console.error(error)
                    }
                })
            }

            $('#reload_wallet').click(function(){
                reload_amount();
            });
```

[START WALLET_SERVER]

[START BLOCKCHAIN_NODE]

2X [BROWSER LOCALHOST:8080]

[TRANSACTION "A" TO "B", VALUE 23]

[BROWSER LOCALHOST:8080 RELOAD BUTTON]

[OUTPUT -23]

This should be enough for you as a small function test. However, I will now comment this out and simply reload the value every three seconds instead.

```
 /*
$('#reload_wallet').click(function(){
    reload_amount();
});
*/

setInterval(reload_amount, 3000)
```

2X [BROWSER LOCALHOST:8080]

[TRANSACTION "A" TO "B", VALUE 23]

2X [BROWSER LOCALHOST:8080]

Automatic display of the value for a blockchain address updated in the background every three seconds should works. If your code is not running as it should now, you can go troubleshooting or download the release for this lesson from Github.

Code Lecture 42

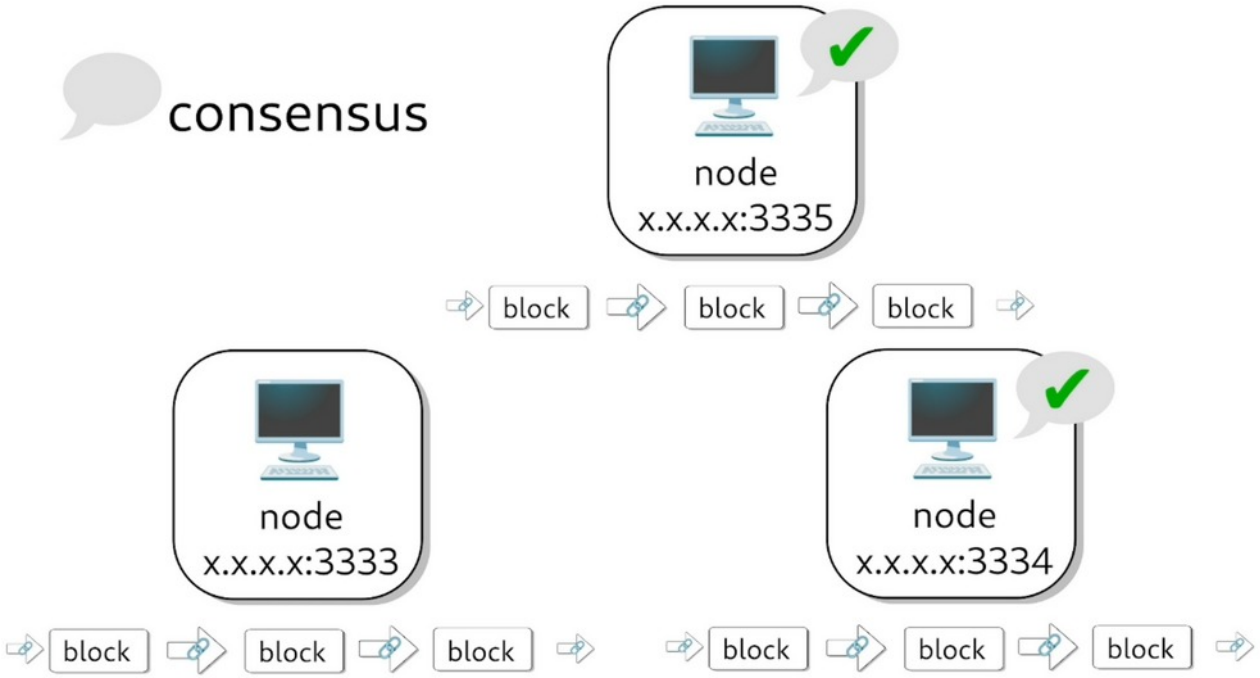
Section 6 – Reaching a Consensus Enables the Synchronization of Nodes Across the Network

> *I have always believed in evolving a consensus before taking any major decision.*
>
> *Narendra Modi*

Lecture 43 – Section Overview: Unraveling Blockchain Mystery - Decentralization Demystified!

In this section, you will learn a lot about the inner workings that allow blockchain technology to function in a decentralized way and without a single entity that has special decision-making power.

Basically, you are programming a kind of self-organizing swarm intelligence that could perhaps be compared to large flocks of birds in flight or schools of fish in the ocean. It is about the functioning of a peer-to-peer network which, as an open system, allows new participants to join or leave at any time without affecting the function of the overall system.



For the sake of demonstration, we will keep our example simple, but the principles used are similar to those that make blockchain applications, especially cryptocurrencies, work across the globe – or across the disk if your are a flat earthener.

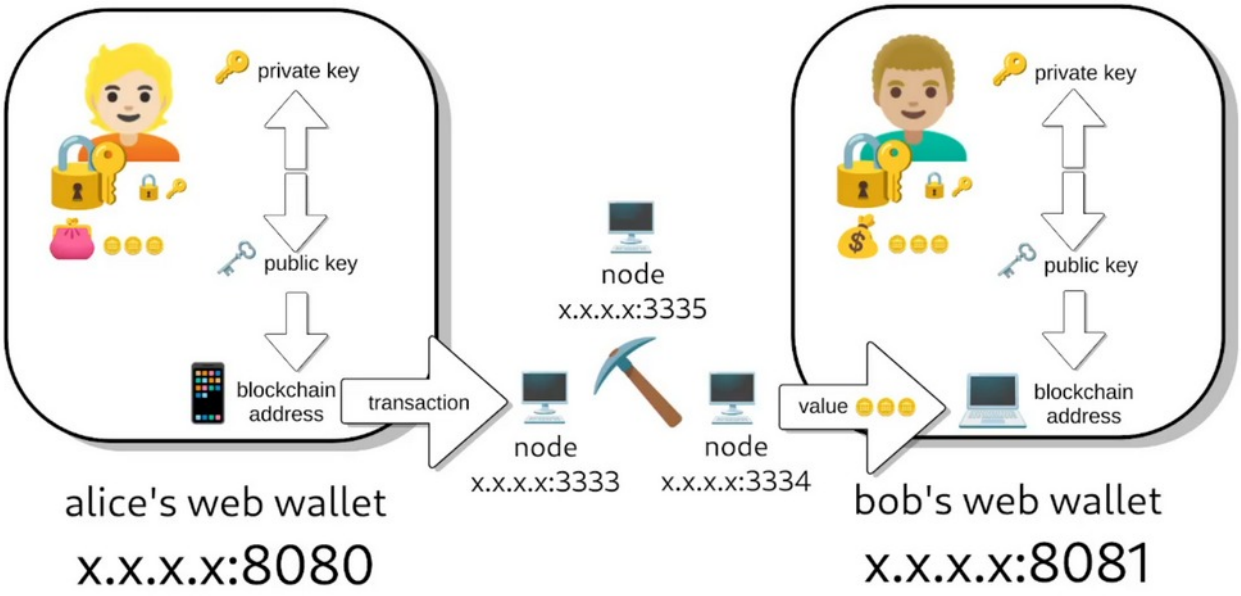
And as with large swarms, the individual instance, as the smallest unit, must be equipped with a simple set of rules that work together to ensure that large structures can be created and tasks can be accomplished that the single entity could not achieve alone.

But before the smallest units can organize themselves, they must have the same information and be able to exchange information with each other without long delay as if they were in one room sharing the same experience at a time.

For this exchange of information to take place, the participants in a peer-to-peer network need to know each other and know how to find more participants and how communicate with each other.

This is your starting point in next lesson of this section. Let’s continue loosely based on Enrico Fermi, who raised the question in a different context: "Where is everybody"?

Lecture 44 – Where Is Everybody? Search for Other Blockchain Nodes on the Net



You are aiming at creating a decentralized system of independently acting nodes. As said before, it is better to keep things simple and manageable. That means you should limit it to 3 nodes. And due to the fact that all your nodes run on you local machine as well as your wallets run on you local machine we differ between both ports and port ranges. 8080 and 8081 for the wallet servers and 3333 to 3335 for the nodes. I will try to provide you some code that should work in the local network and also, at least principally, beyond your internet gateway.

So far you have one blockchain node only and a wallet server which is able to connect to it. They’re connected through APIs exchanging information in JSON. Later you will create transactions requests and the nodes need to share transaction pools and other data with each other. No matter what kind of information your nodes will have to exchange later, first they need to find each other across the network. And that is what you should work on first.

[VS CODE, NEW FOLDER/FILE CMD/MAIN.GO]

```
package main
```

[VS CODE, NEW FILE UTILS/NEIGHBORS.GO]

```
package utils

func IsFoundNode(host string, port uint16) bool {
     target := fmt.Sprintf("%s:%d", host, port)
}
```

[CHECKOUT NET.DIALTIMEOUT CTRL+CLICK]

[VS CODE, NEW FILE UTILS/NEIGHBOR.GO]

```
package utils

func IsFoundNode(host string, port uint16) bool {
     target := fmt.Sprintf("%s:%d", host, port)

     _, err := net.DialTimeout("tcp", target, 1*time.Second)
     if err != nil {
          fmt.Printf("%s %v\n", target, err)
     return false
     }
     return true
}
```

[CMD/MAIN.GO]

```
func main() {
     fmt.Println(utils.isFoundHost("127.0.0.1", 3333))
     fmt.Println(utils.isFoundHost("localhost", 3333))

}
```

[/CMD GO RUN .]

[START NODE]

[/CMD GO RUN .]

```
func main() {
     fmt.Println(utils.iFoundHost("127.0.0.1", 3333))
     fmt.Println(utils.iFoundHost("localhost", 3334))

}
```

[/CMD GO RUN .]

[START NODE -port=3334]

[/CMD GO RUN .]

That means that our function `isFoundHost` to find nodes answering on specific ports seems to work sufficiently. But here comes our limitation in play. Assume your node is running on you local network port 192.180.0.23. I don't want to search now for like all available networks, but limit the search to the actual IP of the node calling utilizing the our new functions plus a few subsequent numbers in the same subnet, and let's say port range 3333 to 3336. That is more than what you will need by now.

[UTILS/NEIGHBORS.GO]

```
func FindNeighbors(myHostIP string, myPort uint16, startIp uint8,
endIp uint8, startPort uint16, endPort uint16) []string {
     address := fmt.Sprintf("%s:%d", myHostIP, myPort)
```

There are severe ways your IP may look like and probably you have a local ip like 192.168.x.x, or from the other private ip ranges mentions on Wikipedia/private Network. But there are rules how an IP in general has to look like and that rules we will use as regular expression to filter out with a pattern how your IP looks like to check if it's a valid IP address.

[WEBSITE REGEX]

This is a regular expression which serves as a patter for exactly that. You can use that pattern to try to find your IP in.

[NEIGHBORS.GO ABOVE FINDNEIGHBORS]

```
var PATTERN = regexp.MustCompile(`((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\.){3})(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)`)
```

[UTILS/NEIGHBORS.GO CONTINUE]

```
     m := PATTERN.FindStringSubmatch(myHostIP)
     if m == nil {
          return nil
     }
     ipPrefix := m[1]
     hostIdent, _ := strconv.Atoi(m[len(m)-1])
     neighbors := make([]string, 0)

     for guessPort := startPort; guessPort <= endPort; guessPort += 1 {
          for variableHostIdent := startIp; variableHostIdent <= endIp; variableHostIdent += 1 {
               guessIP := fmt.Sprintf("%s%d", ipPrefix, hostIdent+int(variableHostIdent))
               guessTarget := fmt.Sprintf("%s:%d", guessIP, guessPort)
               if guessTarget != address && IsFoundNode(guessIP, guessPort) {
                    neighbors = append(neighbors, guessTarget)
               }
          }
     }
     return neighbors
}
```

[/CMD MAIN.GO]

```
func main() {
	fmt.Println(utils.iFoundHost("127.0.0.1", 3333))
	fmt.Println(utils.iFoundHost("localhost", 3334))
	fmt.Println(utils.FindNeighbors("127.0.0.1", 3333, 0, 3, 3333, 3336))
}
```

[STOP ALL RUNNING NODES]

[/CMD GO RUN .]

[START -PORT=3333, START -PORT=3334]

[/CMD GO RUN .]

As you see, this outputs a slice of strings containing all reachable nodes within the same IP subnet plus the following 3 host identifiers after your own IP with the port number range from 3333 to 3336.

Problem here is that 127.0.0.1 on port 3333 is omitted correctly, while 127.0.0.2 to 127.0.0.4 on port 3333 and from 127.0.0.1 to 127.0.0.4 on all other ports apps are listening to are shown as well. That has to do with the setup of my local machine which has a loopback device `lo` which also brings all IP starting with 127.0.0.x to the surface. I'm not even sure if this happens for oyu too if you are working with a different configuration or on macOS or Windows, but I would like to improve this so that we can work with the external IP address. This is something you can work with and improve on in the next lesson.

Code Lecture 44

Lecture 45 – Crossing Borders - Leave the Limitations of Your Local Network Behind!

I had to experiment a little until I found a good function that will safely return a proper extermal IP.

[/UTILS/NEIGHTBORS.GO]

```
func GetHost() string {
     conn, err := net.Dial("udp", "1.1.1.1:80")
     if err != nil {
          log.Println("ERROR:", err)
          os.Exit(1)
     }
     defer conn.Close()

     address := conn.LocalAddr().(*net.UDPAddr)
     ipStr := fmt.Sprintf("%v", address.IP)

     return ipStr
}
```

Basically the following happens: You open a connection to any address, here with protocol udp, to ip 1.1.1.1 and port 80. You are not interested in whether this connection is established or not and of course you close it again shortly before exiting the function call.

If a fundamental error occurs when trying to establish the connection, you log this on the terminal. Then you can assume that you are offline and you don't need to continue. Stop the program if you like.

Your operating system used your assigned address for the connection attempt and this address is now stored in `conn.LocalAddr().(*net.UDPAddr)`. The IP itself can be found in `address.IP` and you convert this value to a string and use it as return value.

I have to admit that you need to know a bit about the TCP/IP network stack and I have only tried this whole procedure for IPv4 and not for IPv6. But actually I owe you a prove that it all works as it should. Please rewrite the main.go in the /cmd folder.

[/CMD/MAIN.GO]

```
func main() {
     // fmt.Println(utils.IsFoundNode("127.0.0.1", 3333))
     // fmt.Println(utils.IsFoundNode("localhost", 3333))

     // fmt.Println(utils.FindNeighbors("127.0.0.1", 3333, 0, 3,
3333, 3336))
```

```
	fmt.Println(utils.GetHost())
}
```

[/CMD GO RUN .]

[/CMD/MAIN.GO]

```
func main() {
	// fmt.Println(utils.IsFoundNode("127.0.0.1", 3333))
	// fmt.Println(utils.IsFoundNode("localhost", 3333))

	// fmt.Println(utils.FindNeighbors("127.0.0.1", 3333, 0, 3, 3333, 3336))

	// fmt.Println(utils.GetHost())

	myAddress := utils.GetHost()
	fmt.Println(utils.FindNeighbors(myAddress, 3333, 0, 3, 3333, 3336))
}
```

[START -PORT=3333, START -PORT=3334, START -PORT=3335]

[/CMD GO RUN .]

I think that's fair enough for now and for such a small demo here you are working on. You have what you need: A slice of strings with the IPs and ports of your blockchain node’s neighborhood without finding your own machine as an element in this slice.

Code Lecture 45

Lecture 46 – Automatic Registration of Blockchain Nodes. I Saw You, You're on My List Now!

With a function to find other nodes in the network, locally or beyond, and create a neighborhood list from them, you can now set the criteria to teach each node creating this neighborhood list for itself. That criteria are kind of constant in this example, so create them as such.

[BLOCKCHAIN/BLOCKCHAIN.GO]

```
const (
	MINING_DIFFICULTY = 3
	MINING_SENDER     = "BLOCKCHAIN REWARD SYSTEM (e.g. minting & fees)"
	MINING_REWARD     = 1.0
	MINING_TIMER_SEC  = 20

	BLOCKCHAIN_PORT_RANGE_START        = 3333
	BLOCKCHAIN_PORT_RANGE_END          = 3336
	NEIGHBOR_IP_RANGE_START            = 0
	NEIGHBOR_IP_RANGE_END              = 3
	BLOCKCHAIN_NEIGHBOR_SYNC_TIME_SEC = 10
)



type Blockchain struct {
	transactionPool   []*Transaction
	chain             []*Block
	blockchainAddress string
	port              uint16
	mux               sync.Mutex

	neighbors    []string
	muxNeighbors sync.Mutex
}
```

[ABOVE TRANSACTIONPOOL()]

```
func (bc *Blockchain) SetNeighbors() {
     bc.neighbors = utils.FindNeighbors(
          utils.GetHost(),
          bc.port,
          NEIGHBOR_IP_RANGE_START,
          NEIGHBOR_IP_RANGE_END,
          BLOCKCHAIN_PORT_RANGE_START,
          BLOCKCHAIN_PORT_RANGE_END)
     if len(bc.neighbors) > 0 {
          log.Printf("This node's neighbors are %v", bc.neighbors)
     } else {
          log.Printf("This node could not find neighbors.")
     }
}
```

Hooray, now every node can create a list of the neighborhood for itself. That's a good thing. But neighbors come and go and we have to keep this list constantly up to date. This would actually be a task for an independently running GO routine. But then I would have to deal with concurrency, open channels and orchestrate the whole thing somehow. That's far too much for this little demo. Instead, we solve this in a very similar way to automated mining. You've already seen the `muxNeighbors` of type `sync.Mutex` in the blockchain struct.

```
func (bc *Blockchain) SyncNeighbors() {
     bc.muxNeighbors.Lock()
     defer bc.muxNeighbors.Unlock()
     bc.SetNeighbors()
}

func (bc *Blockchain) StartSyncNeighbors() {
     bc.SyncNeighbors()
     _ =
time.AfterFunc(time.Second*BLOCKCHAIN_NEIGHBOR_SYNC_TIME_SEC,
bc.StartSyncNeighbors)
}
```

[ABOVE SETNEIGHBORS]

```
func (bc *Blockchain) Run() {
     bc.StartSyncNeighbors()
}
```

[BLOCKCHAIN_NODE.GO]

```
func (bcn *BlockchainNode) Run() {
     bcn.GetBlockchain().Run()

     http.HandleFunc("/", bcn.GetChain)
     http.HandleFunc("/transactions", bcn.Transactions)
     http.HandleFunc("/mine", bcn.DoMine)
     http.HandleFunc("/mine/start", bcn.StartMine)
     http.HandleFunc("/amount", bcn.Amount)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}
```

[START BLOCKCHAIN_NODE.GO -PORT=3333, FOLLOW OUTPUT]

[START BLOCKCHAIN_NODE.GO -PORT=3334, FOLLOW OUTPUT]

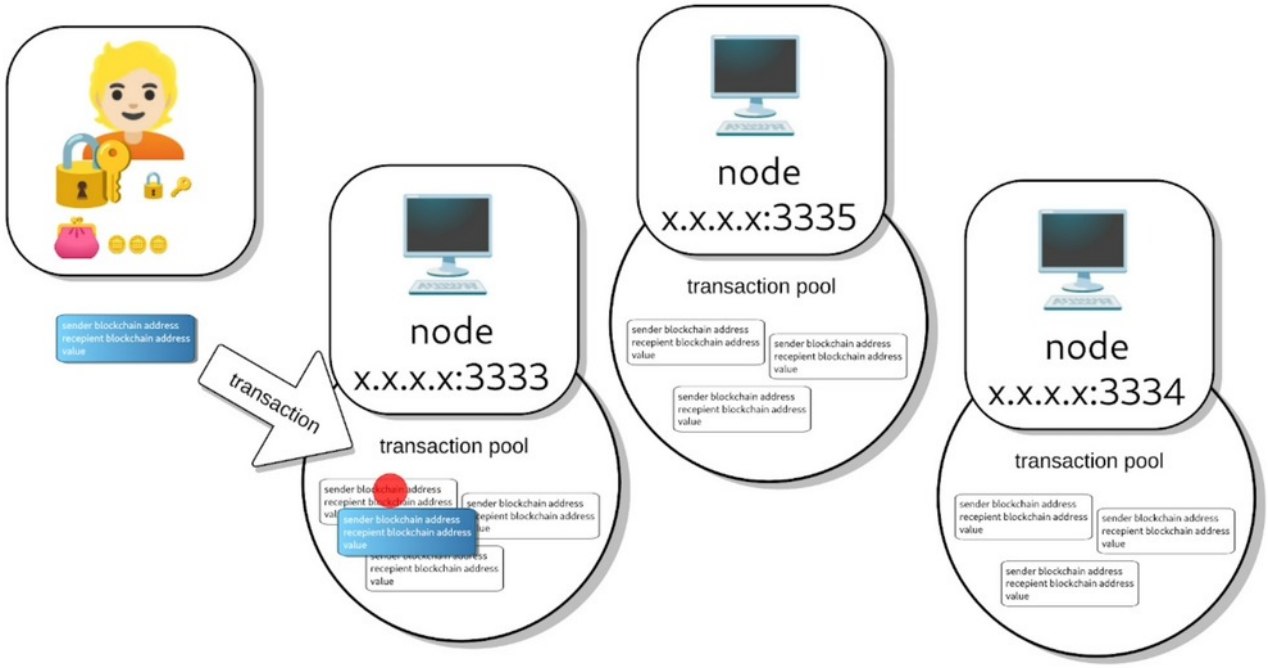
[START BLOCKCHAIN_NODE.GO -PORT=3335, FOLLOW OUTPUT]

[BACK BLOCKCHAIN_NODE.GO -PORT=3333, FOLLOW OUTPUT]

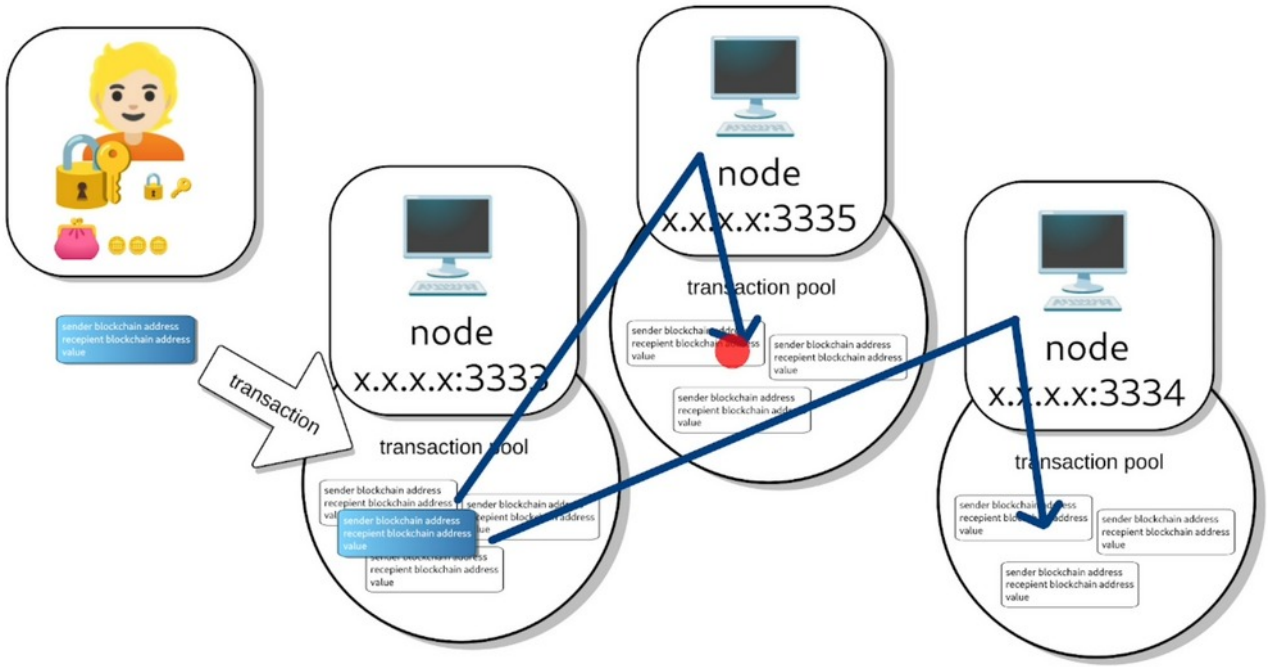
Congratulations to you. You can see that your nodes can not only look around in the neighborhood, they also create each a list and do this automatically on a recurring basis. Unfortunately the nodes are spamming all their efforts to the terminal, but each node has its own list of surrounding nodes now with which it can exchange information soon.

Code Lecture 46

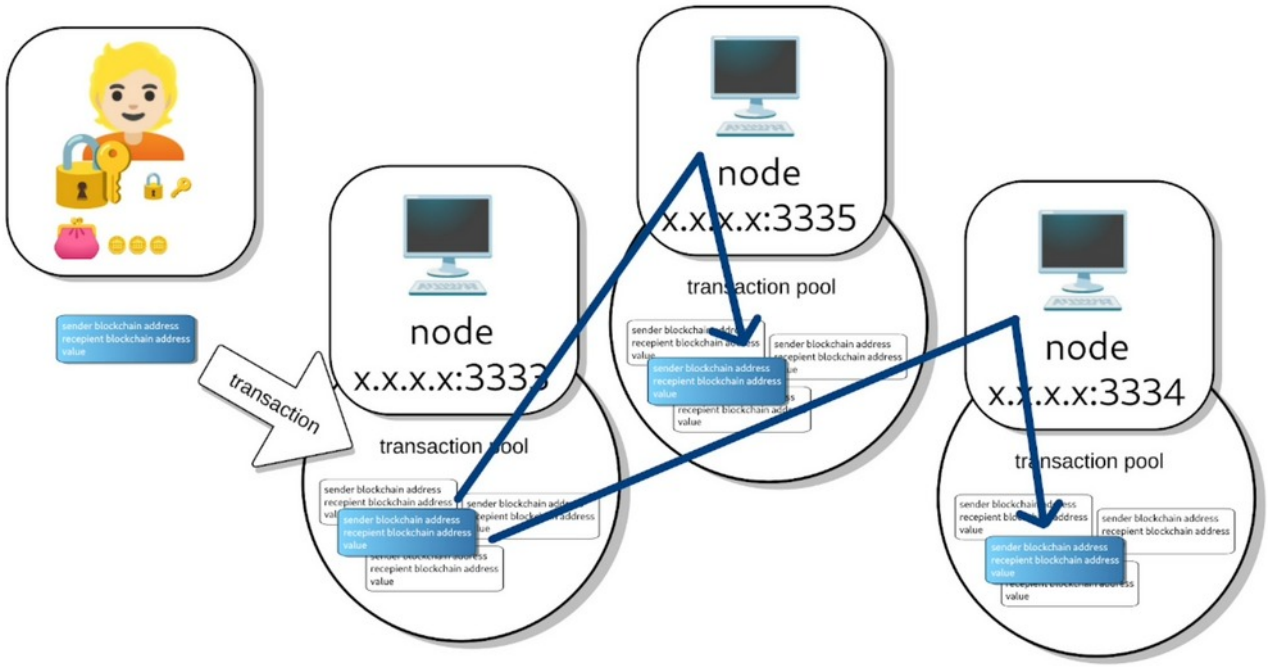
Lecture 47 – Sharing Is Caring - Synchronizing Transactions Across the Known Nodes



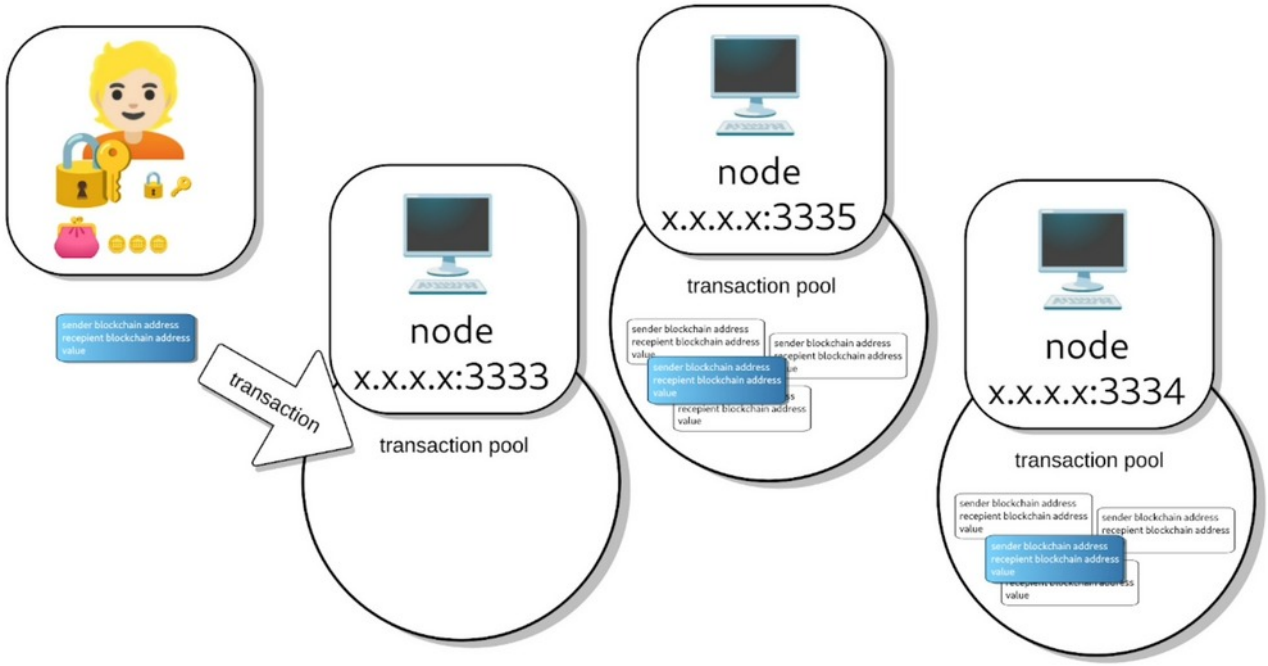
Now that all neighbors know each other, you can start implementing mutual data exchange. As you can see here, you have a wallet server that is connected to the node on port 3333. Let's assume a transaction request that is sent to this node, checked by it, and then transferred to the local transaction pool.



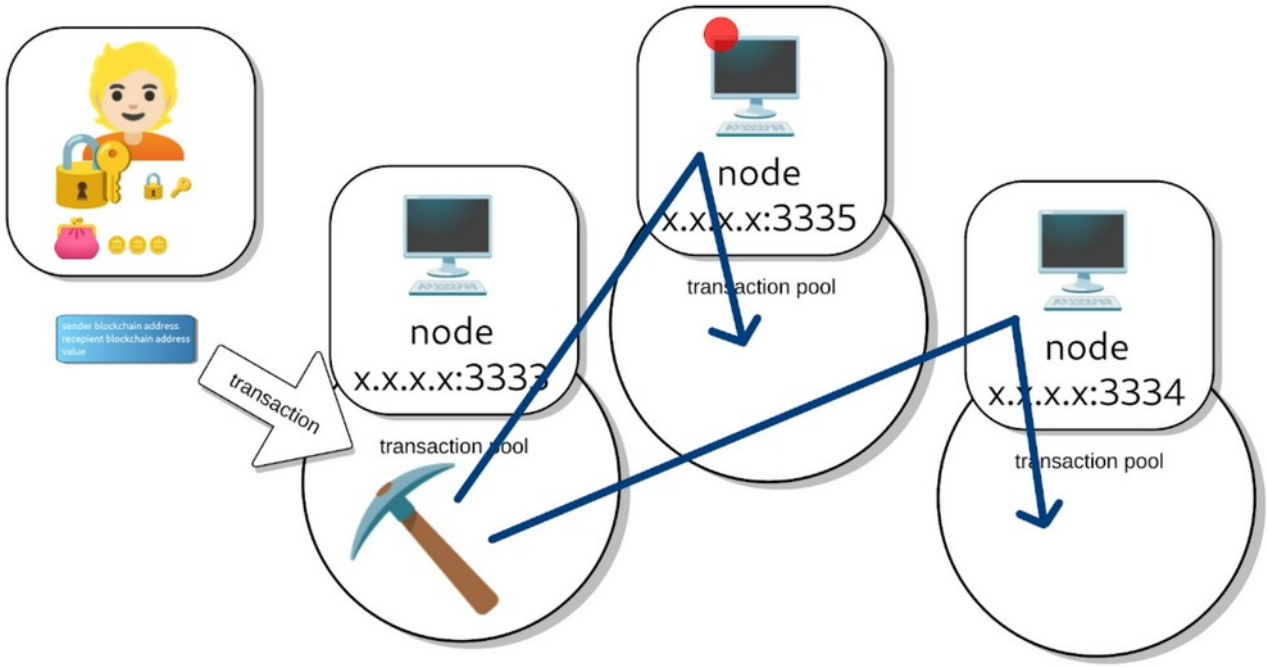
In this lesson, you must first ensure that this transaction from the pool is also transferred to all known nodes so that all nodes have the same chance of mining a block. It makes no sense to keep this transaction secret, because other nodes will need the transaction data at the latest when they are asked to confirm a block - potentially even one from you.



And your incentive to share all the necessary information should be that you want any block that you have found as well as every transaction you have initiated to be verified as quickly as possible before someone else beats you to it with another block, respectively because you want your transaction to be set in tablets of stone as soon as possible.

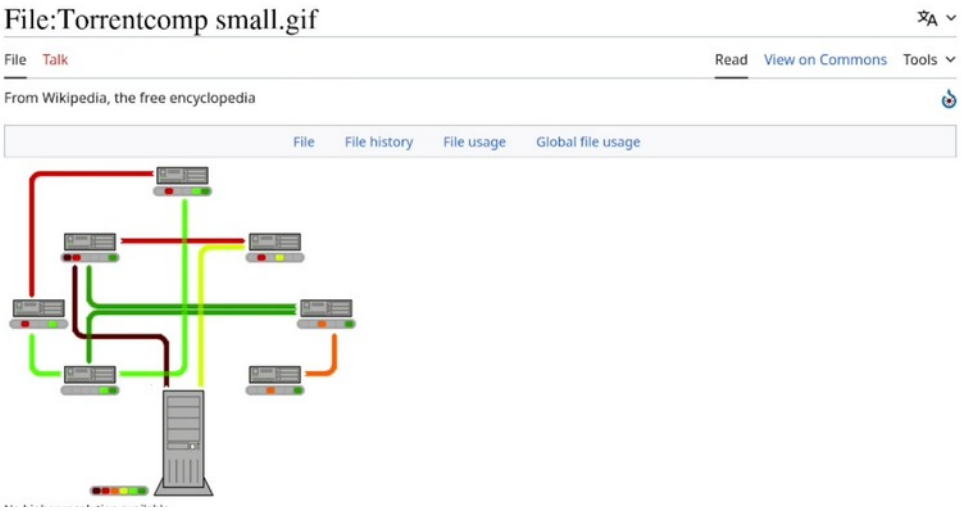


There is also another task with transactions that must be implemented across the swarm. Remember that in our simplified system, all available transactions are simply packed into one block. But there is no real limit to the block size. This makes it possible to empty the transaction pool in one go whenever a new block is confirmed. From here on, however, you must also ensure that all transaction pools are flushed at all known nodes. But one thing at a time.



[Peer-to-peer network exacmples like Bittorrent, napster, Bitcoin, SETI[@]home aso]

[https://en.wikipedia.org/wiki/Peer-to-peer]



In real blockchain applications the data is usually passed around like in large peer-to-peer networks like Bittorrent. Or do you remember Napster? Or SETI@home? Even Microsoft Windows 10 distribution system acts as a proprietary peer-to-peer network system relieving Microsoft's update servers from 30%-50% of the internet bandwidth traffic.

[https://en.wikipedia.org/wiki/REST]

Here, you use REST instead and simply send the data to each node. The approach is basically based on the same principles and is quite sufficient here to make clear what is happening. That means you can start with the receiving side of the blockchain nodes as a kind of remote control receiver.

[VS CODE BLOCKCHAIN_NODE.GO]

[TRANSACTIONS COPY/PASTE METHODPOST]

[EDIT AS FOLLOWS]

```
    case http.MethodPut:
        decoder := json.NewDecoder(r.Body)
        var t block.TransactionRequest
        err := decoder.Decode(&t)
        if err != nil {
            log.Printf("ERROR: %v", err)
            io.WriteString(w, string(utils.JsonStatus("fail")))
            return
        }
        if !t.Validate() {
            log.Println("ERROR: missing field(s)")
            io.WriteString(w, string(utils.JsonStatus("fail")))
            return
        }
        publicKey := utils.String2PublicKey(*t.SenderPublicKey)
        signature := utils.String2Signature(*t.Signature)
        bc := bcn.GetBlockchain()
```

[CHECKOUT WITH CTRL+CLICK CREATETRANSACTION]

You cannot call `CreateTransaction()` at this point because the plan is that when a transaction is actually created, you want to distribute it across all nodes. If all nodes do this by calling `CreateTransaction()`, transactions are kind of multiplied. What you really want to do when is comes to providing an option for updating a transaction coming from other nodes is to write it directly locally to your transaction pool. I hope that makes sense to you. If not, just keep watching, it will become clear shortly what you are going to achieve here. You don't want to check for `isTransacted` but for `isUpdated`. Just do that!

[BLOCKCHAIN_NODE.GO CONTINUE]

```
        isUpdated := bc.AddTransaction(*t.SenderBlockchainAddress,
```

```
                *t.RecipientBlockchainAddress, *t.Value, publicKey,
signature)

          w.Header().Add("Content-Type", "application/json")
          var m []byte
          if !isUpdated {
               w.WriteHeader(http.StatusBadRequest)
               m = utils.JsonStatus("fail")
          } else {
```

`http.StatusCreated` is repsonse code 201 but after successfully updating with this PUT request you will get a status code 200, `http.StatusOK` which is default anyway. You don't need to write that header.

```
               w.WriteHeader(http.StatusCreated)
               m = utils.JsonStatus("success")
          }
          io.WriteString(w, string(m))
```

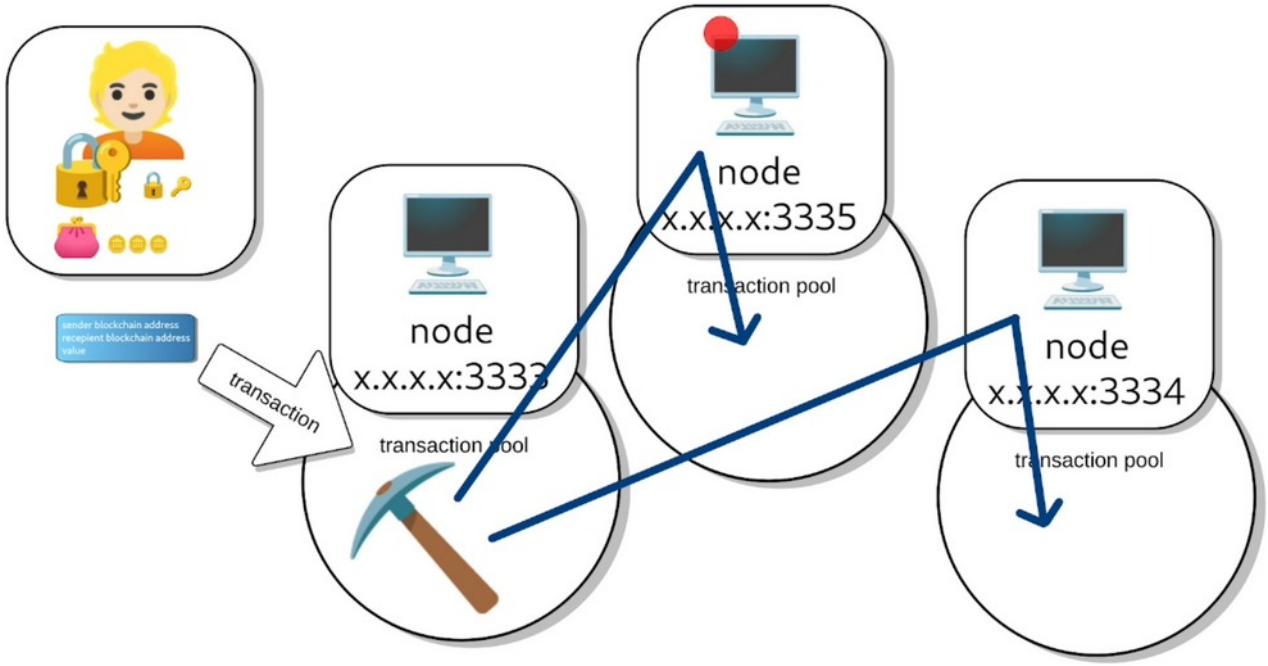
That's all what was to do here. This is your entire handler for `MethodPut`, which allows you to write a transaction directly to the transaction pool. While you are here, why don't you write a request handler for `MethodDelete`?

```
     case http.MethodDelete:
          bc := bcn.GetBlockchain()
          bc.ClearTransactionPool()
          io.WriteString(w, string(utils.JsonStatus("success")))
```

[BLOCKCHAIN/BLOCKCHAIN.GO BELOW TRANSACTIONPOOL()]

```
func (bc *Blockchain) ClearTransactionPool() {
     bc.transactionPool = bc.transactionPool[:0]
}
```

In this way, you now have a simple handler that uses `MethodDelete` to ensure that the local transaction pool is emptied.



A small progress report at this point. In principle, you have now laid the foundation for other nodes to be able to manipulate the transaction pool of each node's current blockchain copy. On the one hand, others can now use method PUT to add single transactions to your pool, and on the other hand, they can also delete transactions. In this example, all transactions are always wiped out immediately.

Up to this point, you have practically implemented the command-receiving side. There has been no sign of any merging logic or self-organizing synchronization of the transaction pools between the nodes. Let's tackle that now.

[BLOCKCHAIN/BLOCKCHAIN.GO CREATETRANSACTION() TODO]

```
     if isTransacted {
          for _, n := range bc.neighbors {
               publicKeyStr := fmt.Sprintf("%064x%064x",
senderPublicKey.X.Bytes(),
                    senderPublicKey.Y.Bytes())
               signatureStr := s.String()
               bt := &TransactionRequest{
                    &sender, &recipient, &publicKeyStr, &value,
&signatureStr}
               m, _ := json.Marshal(bt)
               buf := bytes.NewBuffer(m)
               endpoint := fmt.Sprintf("http://%s/transactions", n)
```

```
                client := &http.Client{}
                req, _ := http.NewRequest("PUT", endpoint, buf)
                resp, err := client.Do(req)
                if err != nil {
                     log.Printf("%v", resp)
                }
           }
      }
```

This is the most simple way I can imagine to spread a new valid transaction across all known nodes and place them using `methodPut` into the other node's transaction pools.

[EXPLAIN LINE BY LINE]

But whenever a node has mined a block, it will empty its transaction pool. The other nodes must be informed of this and their pool must also be emptied. As mentioned before, I realize that this is a simplified model and leaves out a lot of verification and double and triple checking of the legitimacy of such requests. But hopefully the principles become clear.

Let's delete some other people's transaction pools!

Actually you do that for your local transaction pool only in

[BLOCKCHAIN.GO CREATEBLOCK()]

```
func (bc *Blockchain) CreateBlock(nonce int, previousHash
[32]byte) *Block {
     b := NewBlock(nonce, previousHash, bc.transactionPool)
     bc.chain = append(bc.chain, b)
     bc.transactionPool = []*Transaction{}
     for _, n := range bc.neighbors {
          endpoint := fmt.Sprintf("http://%s/transactions", n)
          client := &http.Client{}
          req, _ := http.NewRequest("DELETE", endpoint, nil)
          resp, err := client.Do(req)
          if err != nil {
               log.Printf("%v", resp)
          }
     }
```

```
    return b
}
```

[EXPLAIN LINE BY LINE]

As You see, in GO such procedures are really really simple to implement. It feels a bit like it was made for that, and that is close to the truth. GO was explicitly written for online server/client applications, well among other things. But I will resist trying to give a long praise to the GO programming language and its developers at this point. Instead, let's start a few nodes and see how they deal with new transactions and distribution of the transaction pool among themselves.

[START BLOCKCHAIN NODES -PORT=3333, 3334, and 3335, WAIT NEIGHBORHOOD SYNC]

[BROWSER LOCALHOST:3333/TRANSACTIONS]

[BROWSER LOCALHOST:3334/TRANSACTIONS]

[BROWSER LOCALHOST:3335/TRANSACTIONS]

[BROWSER LOCALHOST:8080]

[BROWSER LOCALHOST:8081]

[TRANSACTION "A" TO "B", VALUE 137]

[CHECKOUT 3333, 3334, 3335]

[TRANSACTION "A" TO "B", VALUE 23]

[TRANSACTION "A" TO "B", VALUE 42]

[CHECKOUT 3333, 3334, 3335]

[BROWSER LOCALHOST:3333/MINE]

[BROWSER LOCALHOST:3333/CHAIN]

[CHECKOUT 3333, 3334, 3335]

The whole thing seems to work so well that you want to rub your hands together like Mr. Burns and exclaim "excellent!".

However, I have now restarted all blockchain nodes and both wallet servers and would like to briefly show you another transaction and a problem that is becoming apparent.

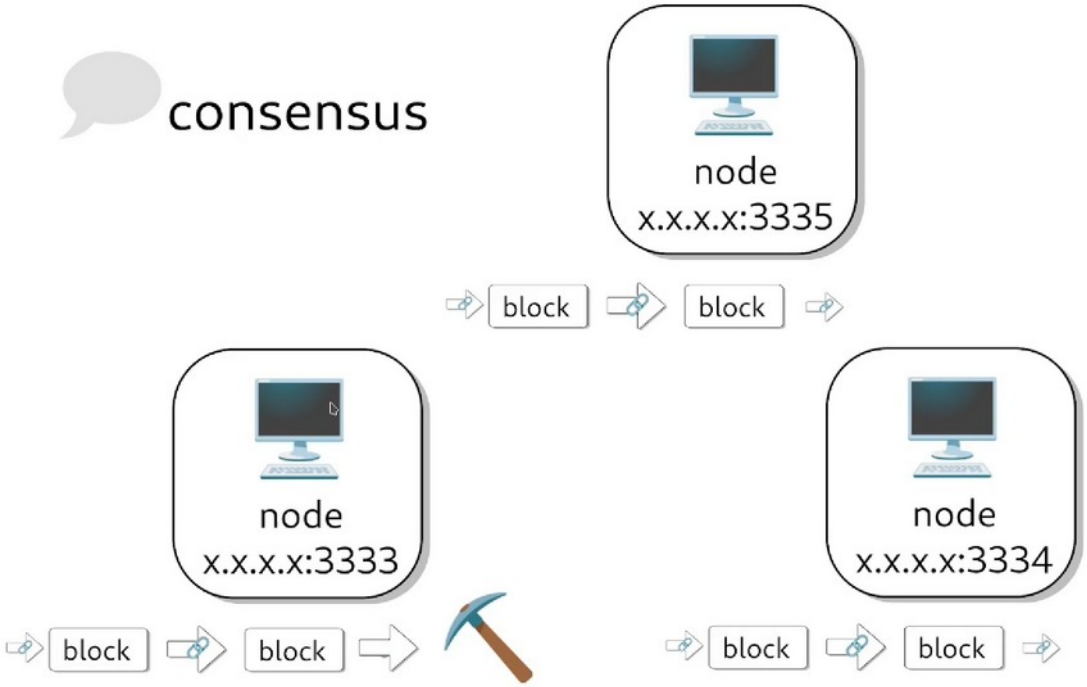
[SAME TEST, CHECK DESTINATION WALLET]

This last example of a transaction and the resulting blockchain at the node on port 3334, which is empty except for a genesis block, show that we not only have to transfer the transactions to the

transaction pools of the other nodes, but that the current status of the blockchain also has to be transferred and synchronized. We will tackle this in the next lessons.

Code Lecture 47

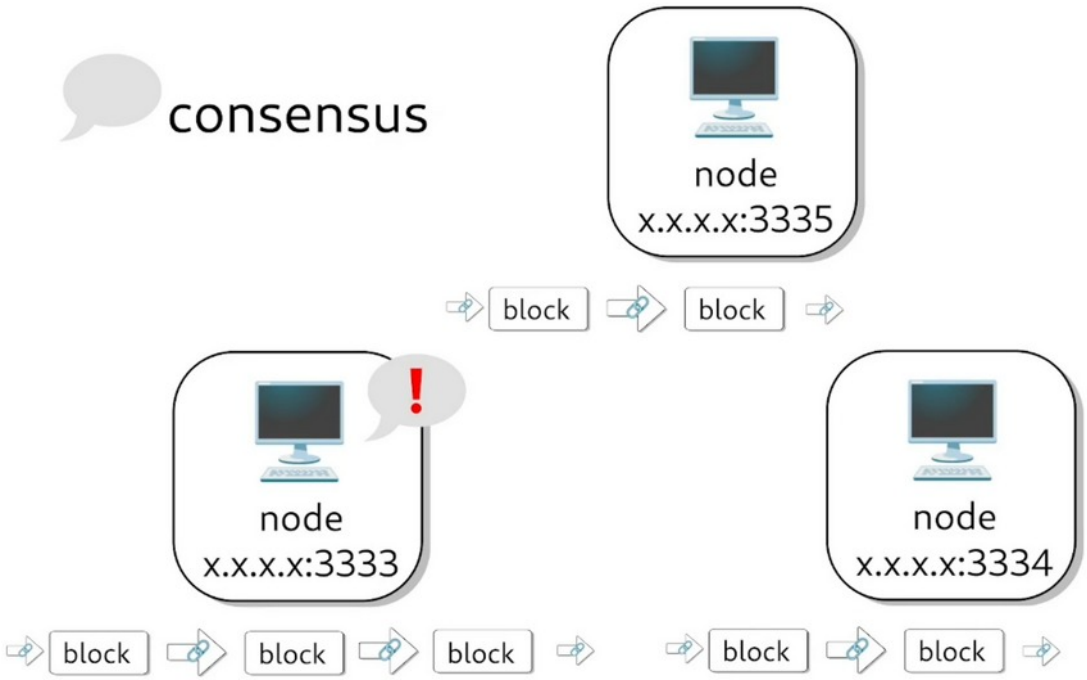
Lecture 48 – What Is This Consensus That Everyone Is Talking About and How Do You Achieve It?



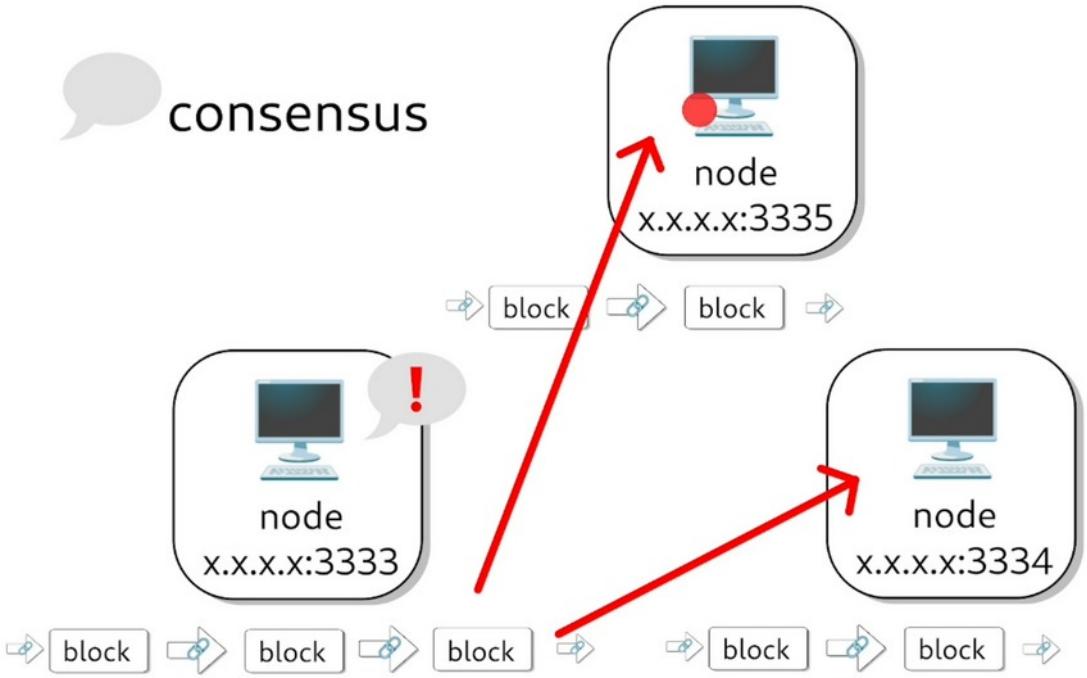
Time to simplify an often misunderstood topic which is in reality more complex than presented here. If you are already more familiar with the topic of how blockchain systems ensure agreement on the current state of the blockchain ledger, please forgive me for not being able to go into all the details here.

This lecture is about consensus.

Alright, let's break down how consensus is achieved among different nodes in a blockchain network, using some easy-to-follow example. Imagine we have three nodes accessible through port numbers 3333, 3334, and 3335.

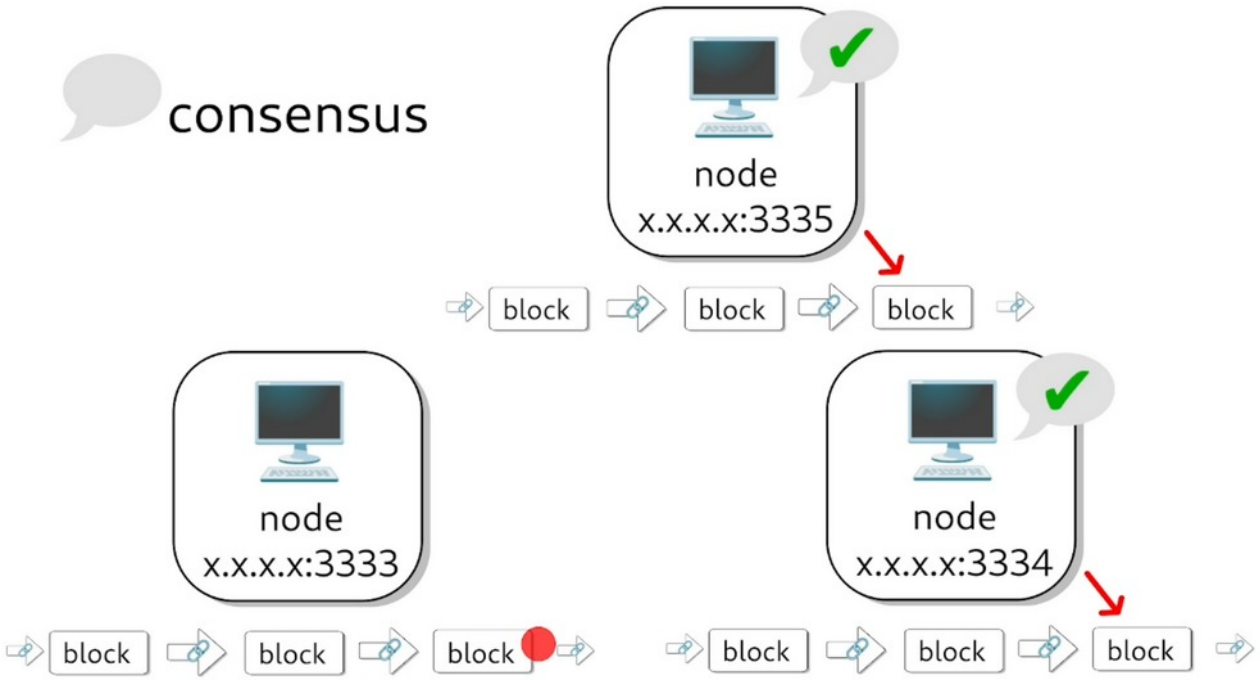


So, let's say the mining node on port 3333 finds a new block and adds that block to its copy of the blockchain. The nodes 3334 and 3335 need to verify it before it's officially part of the blockchain. Until it's verified, the blockchain copies of other nodes remain unchanged. All participating nodes need to reach consensus of what “the blockchain” is and of how many and which blocks are part of the latest version.

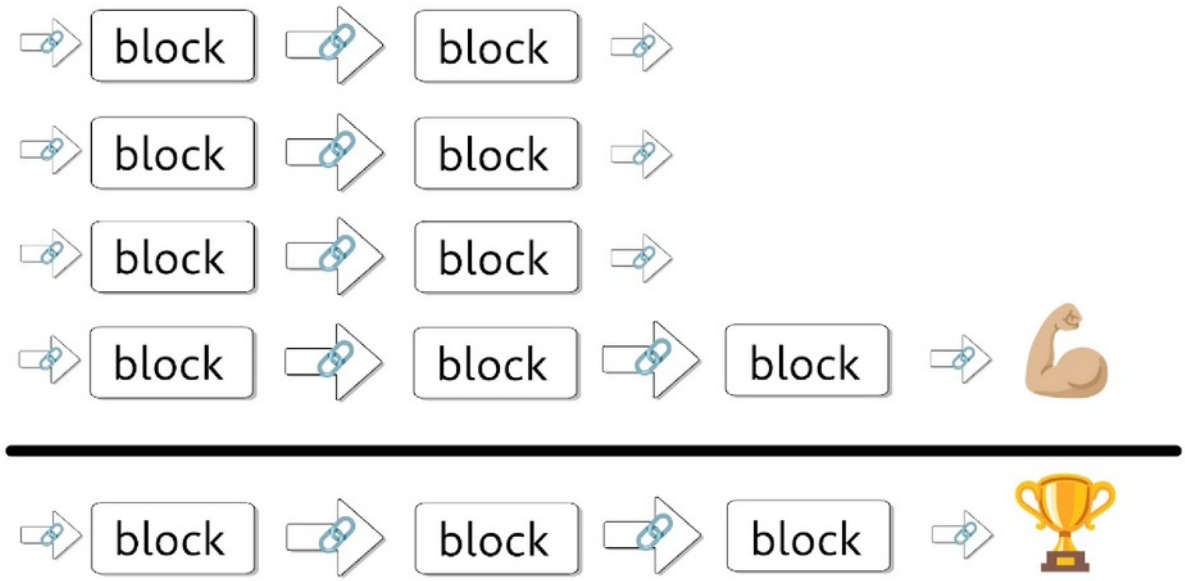


That’s said, the new block needs to be passed to all other nodes for verification. The other nodes should already be equipped with the same two blocks before the new block. That means the last hash value which is the link to the new block is identical.

You know already that each block of the blockchain comes with a set of values which lead to a hash value which becomes part of the next block. But if you randomly add blocks without proper verification, we end up with a messed-up chain that won't be accepted by the network.



When you communicate with other nodes to synchronize their chains, you need to ensure that validf blocks have been received before adding them to your own chain. It's like checking the authenticity of information before accepting it. There may come up some situation where one of the nodes has to catch up some more than just one block. How to decide who to trust?



Now, there is a simple rule in blockchain networks: the longest chain wins. This rule ensures consistency and prioritizes the most robust chain. A longer blockchain not only demonstrates more computational power but also builds trust among nodes. Remember, as more nodes already verified the chain over time, its more reliability and hence trustworthiness increases significantly.

Now, imagine if someone tries to introduce a shorter chain into the network. Even if it's technically correct, it can't compete with the longer, more established chain. Trust and consensus go hand in hand in blockchain networks.

Next, let's dive into some code to see how this all works in practice.

Lecture 49 – Don’t Trust, Verify! Let Your Node Verify a Blockchain First, Then Accept It

Blockchain applications claim to be trustless systems which does not mean that trust is a bad thing as a whole. It means that you don’t have any risk if you just eliminate the need of trust within a relation of any kind from the beginning. It, too, does not mean you trade off faith against distrust. In blockchain systems you can exchange it to verification; what both sides get is truth. And that is what the slogan “Don’t trust, verify!” means.

In this lesson you teach your nodes how to verify any blockchain which is introducing a new block before accepting it. And during this verification they will even strengthen the reliability of the blockchain ledger itself.

[BLOCKCHAIN/BLOCK.GO]

```
type Block struct {
	timestamp    int64
	nonce        int
	previousHash [32]byte
	transactions []*Transaction
}
```

Whenever a new block comes in, besides the timestamp, these are the elements you have to check validity for.

[BELOW NEWBLOCK(), ABOVE PRINT()]

```
func (b *Block) PreviousHash() [32]byte {
	return b.previousHash
}

func (b *Block) Nonce() int {
	return b.nonce
}

func (b *Block) Transactions() []*Transaction {
	return b.transactions
}
```

[BLOCKCHAIN/BLOCKCHAIN.GO BELOW CALCULATETOTALAMOUNT]

```
func (bc *Blockchain) ValidChain(chain []*Block) bool {
     previousBlock := chain[0]
     currentIndex := 1
     for currentIndex < len(chain) {
          b := chain[currentIndex]
          if b.previousHash != previousBlock.Hash() {
               return false
          }

          if !bc.ValidProof(b.Nonce(), b.PreviousHash(),
b.Transactions(), MINING_DIFFICULTY) {
               return false
          }

          previousBlock = b
          currentIndex += 1
     }
     return true
}
```

Basically, this is a while-do loop that loops along the blockchain block by block and uses the existing `ValidProof()` method to ensure that the data existing in the blocks is a valid chain.

You will see that this is a quite fast procedure if you refresh your memory on how `ValidProof()` works.

[CTRL+CLICK VALIDPROOF()]

```
func (bc *Blockchain) ValidProof(nonce int, previousHash [32]byte,
transactions []*Transaction, difficulty int) bool {
     zeros := strings.Repeat("0", difficulty)
     guessBlock := Block{0, nonce, previousHash, transactions}
     guessHashStr := fmt.Sprintf("%x", guessBlock.Hash())
```

```
    return guessHashStr[:difficulty] == zeros
}
```

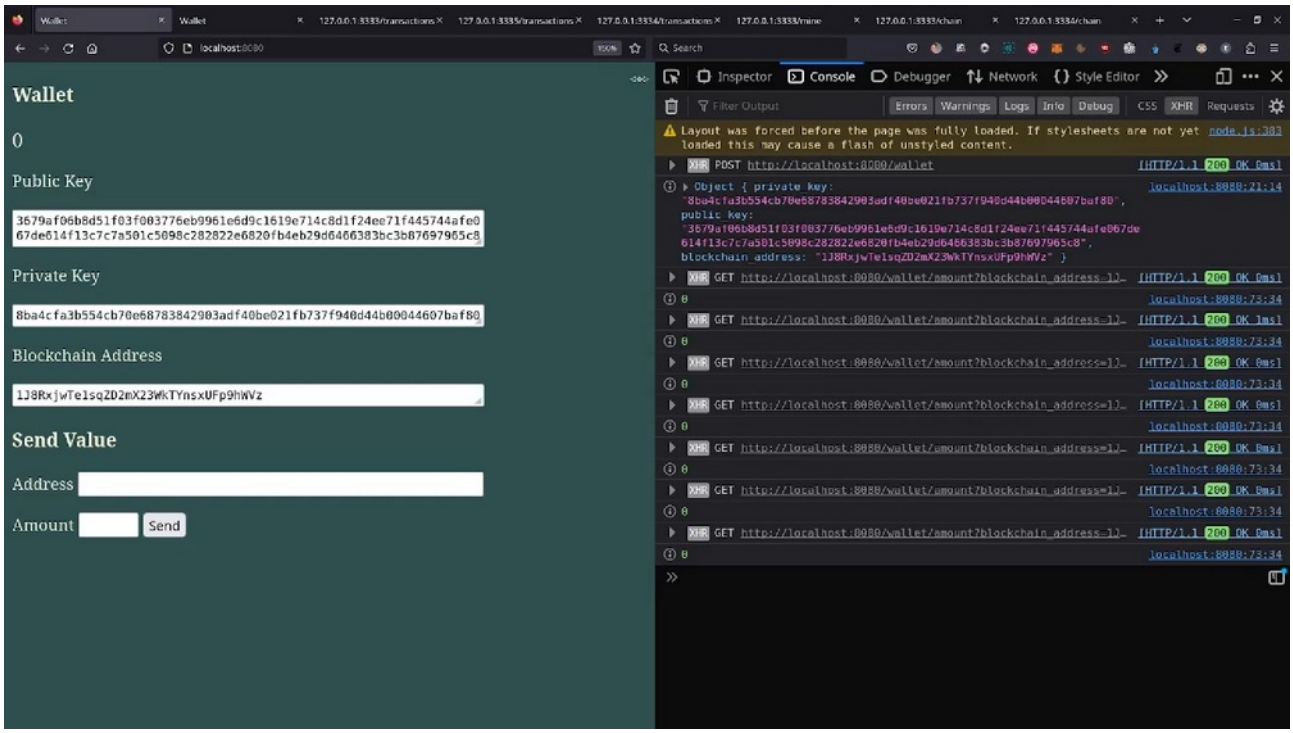
You used `ValidProof()` already when your node was during the `ProofOfWork()` algorithm desperately searching for a `nonce`. But this time you handle over a `nonce` you assume to be good already and only need to ensure for one nonce if it matches to make a block fulfilling the requirements to be valid or not. That's by far less work than finding such a value.

[[BLOCKCHAIN/BLOCKCHAIN.GO VALIDCHAIN()]

With `ValidChain()` you have a method at hand now which makes it a reasonable fast kind of validator of complete chains. In reality this is by far more complex, and caching verification status, local storage in databases, milestone keeping, and other programming concepts come into play. That means you don't want for every new found or proposed block, validate the complete blockchain, right? But in principle, what you achieve with this blockchain method needs to take place at least once you are synchronizing your blockchain with that of other nodes.

Code Lecture 49

Lecture 50 – Resolving Conflicts - Length Does Matter: The Longest-Chain Rule



You are doing really well and I can already see the finishing line, so to say. Do you remember the example demo from before in which the amount sent to the address of wallet B was not updated, because wallet B was connected to a different blockchain node as a gateway and that node had not updated its blockchain copy? That is an inconsistency you can address now. One node showing different values for an address than another node is a conflict like an inner conflict people have from time to time. Contrary to such issues this conflict can be resolved easily and that is what you will do in this lecture: resolve conflicts!

[BLOCKCHAIN/BLOCKCHAIN.GO BELOW METHOD MARSHALJSON()]

For dealing with blockchains and to pass their data around you are already able to marshal all the data in JSON. But now you need also to make chain data which is available from other nodes to be interpreted by your node. So you need also to teach it to unmarshal JSON.

```
func (bc *Blockchain) UnmarshalJSON(data []byte) error {
     v := &struct {
          Blocks *[]*Block `json:"chain"`
     }{
          Blocks: &bc.chain,
     }
     if err := json.Unmarshal(data, &v); err != nil {
          return err
     }
     return nil
}
```

Feels like you did that thousand times already, or?

Please be aware that all the other elements from the type `Blockchain` you don't need to care about when unmarshalling JSON of the blockchain: Transaction pool, addresses, port, neighbors, mutexes – all that you can ignore. It will anyway not be included in the JSON you will have to process.

```
type Blockchain struct {
     transactionPool   []*Transaction
     chain             []*Block
     blockchainAddress string
     port              uint16
```

```
	mux               sync.Mutex

	neighbors    []string
	muxNeighbors sync.Mutex
}
```

Interesting is the structure of the blockchain only at this point. And that structure is in the type pointer to a slice of pointers to blocks: `*[]*Block`

The rest of the information you need to concern about is encoded in JSON in the blocks itself. That means also that information you need to be able to unmarshal when you receive respectively request information from blocks in JSON.

[BLOCKCHAIN/BLOCKS.GO BELOW MASHALJOSN]

```
func (b *Block) UnmarshalJSON(data []byte) error {
	var previousHash string
	v := &struct {
		Timestamp    *int64          `json:"timestamp"`
		Nonce        *int            `json:"nonce"`
		PreviousHash *string         `json:"previous_hash"`
		Transactions *[]*Transaction `json:"transactions"`
	}{
		Timestamp:    &b.timestamp,
		Nonce:        &b.nonce,
		PreviousHash: &previousHash,
		Transactions: &b.transactions,
	}
	if err := json.Unmarshal(data, &v); err != nil {
		return err
	}
	ph, _ := hex.DecodeString(*v.PreviousHash)
	copy(b.previousHash[:], ph[:32])
	return nil
}
```

Seems unmashalling data from the blockchain becomes a rat tail of unmashalling different included data types. After you can now unmarshal the blocks you probably realized that you also need to be able to unmarshal transactions as well. You switch to

[BLOCKCHAIN/TRANSACTIONS.GO BELOW MASHALLJSON]

```
func (t *Transaction) UnmarshalJSON(data []byte) error {
     v := &struct {
          Sender    *string  `json:"sender_blockchain_address"`
          Recipient *string  `json:"recipient_blockchain_address"`
          Value     *float32 `json:"value"`
     }{
          Sender:    &t.senderBlockchainAddress,
          Recipient: &t.recipientBlockchainAddress,
          Value:     &t.value,
     }
     if err := json.Unmarshal(data, &v); err != nil {
          return err
     }
     return nil
}
```

Well, up to now you umashalled a lot of data types but didn’t resolve any kind of conflict. Let’s work on that:

[BLOCKCHAIN/BLOCKCHAIN.GO BELOW METHOD VALIDCHAIN()]

```
func (bc *Blockchain) ResolveConflicts() bool {
     var longestChain []*Block = nil
     maxLength := len(bc.chain)

     for _, n := range bc.neighbors {
          endpoint := fmt.Sprintf("http://%s/chain", n)
          resp, _ := http.Get(endpoint)
          if resp.StatusCode == 200 {
```

```
                var bcResponse Blockchain
                decoder := json.NewDecoder(resp.Body)
                _ = decoder.Decode(&bcResponse)

                chain := bcResponse.Chain()

                if len(chain) > maxLength && bc.ValidChain(chain) {
                     maxLength = len(chain)
                     longestChain = chain
                }
           }
     }
```

[AT TOP UNDER NEWBLOCKCHAIN()]

```
func (bc *Blockchain) Chain() []*Block {
     return bc.chain
}
```

[RESOLVECONFLICTS() CONTINUE]

```
     if longestChain != nil {
          bc.chain = longestChain
          log.Printf("Resolved conflicts: blockchain was replaced")
          return true
     }
     log.Printf("Resolved conflicts: blockchain was not replaced")
     return false
}
```

I think what is going on in `ResolveConflicts()` is pretty straight forward. It is a method you can call for your blockchain. It ranges over all known nodes in the neighborhood and requests their copies of the blockchain. Within that loop a few things happen:

1. the received JSON Data is unmarshalled

2. the JSON is checked to be valid and decodeable
3. IF the length of the blockchain is bigger than the length of your node's blockchain copy AND the received blockchain is a valid and verified blockchain THEN

well, in that case you take over the new blockchain as the better or lets say more reliable one dismissing your own. The rest if logging informational stuff to the full node operator.

I know, the method `ResolveConflicts()` cannot ensure world peace but it will do its job very well. A consensus API is required so that it can start working at all. To create one is your task in the next lecture.

Code Lecture 50

Lecture 51 – Create Consensus API: Open a Door for New Blocks Propagated Through Other Nodes

To have an API to call consensus among nodes we create a route first. That's easy because it is very similar to the `/mine` route.

[BLOCKCHAIN_NODE.GO BOTTOM]

```
func (bcn *BlockchainNode) Run() {
	bcn.GetBlockchain().Run()

	http.HandleFunc("/", bcn.GetChain)
	http.HandleFunc("/transactions", bcn.Transactions)
	http.HandleFunc("/mine", bcn.Mine)
	http.HandleFunc("/mine/start", bcn.StartMine)
	http.HandleFunc("/amount", bcn.Amount)
	http.HandleFunc("/consensus", bcn.Consensus)

log.Fatal(http.ListenAndServe("0.0.0.0:"+strconv.Itoa(int(bcn.Port
())), nil))
}
```

[RIGHT ABOVE THAT]

```
func (bcn *BlockchainNode) Consensus(w http.ResponseWriter, r
*http.Request) {
	switch r.Method {
```

```
     case http.MethodPut:
          bc := bcn.GetBlockchain()
          replaced := bc.ResolveConflicts()

          w.Header().Add("Content-Type", "application/json")
          if replaced {
               io.WriteString(w,
string(utils.JsonStatus("success")))
          } else {
               io.WriteString(w, string(utils.JsonStatus("fail")))
          }
     default:
          log.Printf("ERROR: Invalid HTTP method")
          w.WriteHeader(http.StatusBadRequest)
     }
}
```

That is now your route `/consensus` with the corresponding handler to handle any PUT request addressed to a your node. Now you need to make sure, that whenever one of the nodes found a valid block it will also call for consensus. When is that the case? Answer is easy, whenever `Mining()` took place and a block was created, your node should not only append that block to it's own blockchain, but also call all known nodes' APIs for consensus. The other nodes will then use their brand new methods to validate the incoming blockchain and exchange their blockchain with the the longest chain.

[BLOCKCHAIN.GO METHOD MINING()]

…

```
log.Println("action=mining, status=success")

     for _, n := range bc.neighbors {
          endpoint := fmt.Sprintf("http://%s/consensus", n)
          client := &http.Client{}
          req, _ := http.NewRequest("PUT", endpoint, nil)
          resp, err := client.Do(req)
          if err != nil {
```

```
                log.Printf("%v", resp)
            }
        }

        return true
```

…

With that code implemented right after each time mining no matter what if mined manually or automatically this request for consensus will be sent. That means the other nodes will resolve conflicts, reach consensus, and update their blockchains eventually.

But there is one situation more to consider. Remember this method `Run()` for type `Blockchain` you created to start the blockchain? We need to change something there that in the beginning with every start of the blockchain application

1. all neighbors are found
2. conflicts are resolved, and in case you want to run mining automatically
3. start mining

[BLOCKCHAIN.GO METHOD RUN()]

```
func (bc *Blockchain) Run() {
    bc.StartSyncNeighbors()
    bc.ResolveConflicts()
    // bc.StartMining()
}
```

[UTILS/NEIGHBORS.GO]

```
func IsFoundNode(host string, port uint16) bool {
    target := fmt.Sprintf("%s:%d", host, port)

    _, err := net.DialTimeout("tcp", target, 1*time.Second)
    if err != nil {
        // fmt.Printf("%s %v\n", target, err)
        return false
    }
```

```
    return true
}
```

[LAST CHECK/ADMIT TESTED OFFSCREEN]

COMMENT: UNMARSHALJSON EXPLODE INTO YOUR FACE NO ERROR HANDLING!!!

[START BLOCKCHAIN NODES -PORT=3333, 3334, and 3335, WAIT NEIGHBORHOOD SYNC]

[BROWSER LOCALHOST:8080]

[BROWSER LOCALHOST:8081]

[BROWSER LOCALHOST:3333/TRANSACTIONS]

[BROWSER LOCALHOST:3334/TRANSACTIONS]

[BROWSER LOCALHOST:3335/TRANSACTIONS]

[BROWSER LOCALHOST:3333/CHAIN]

[BROWSER LOCALHOST:3334/CHAIN]

[BROWSER LOCALHOST:3335/CHAIN]

[TRANSACTION "A" TO "B", VALUE 137]

[BROWSER LOCALHOST:3333/MINE]

[CHECKOUT 3333, 3334, 3335 BOTH TRANSACTIONS AND CHAIN]

[TRANSACTION "A" TO "B", VALUE 23]

[TRANSACTION "A" TO "B", VALUE 42]

[BROWSER LOCALHOST:3333/MINE]

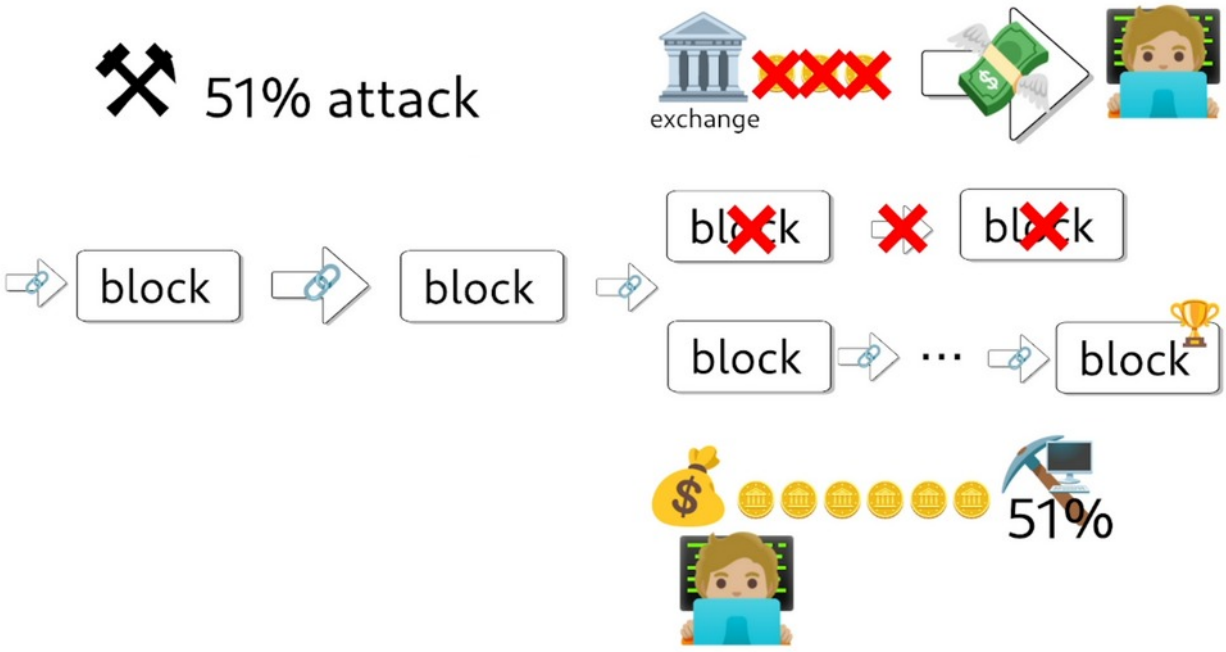
[CHECKOUT 3333, 3334, 3335 BOTH TRANSACTIONS AND CHAIN]

That seems to work like clockwork. If you belong to generation x, you can now sit back, light a cigar and as a tribute to Hannibal Smith from the 80s TV series A-Team say "I love it when a plan comes together."

Before I will perform a final demonstration of your blockchain application I want to draw your attention to some nowadays more academical problem of blockchains called the 51% attack.

Code Lecture 51

Lecture 52 – Hostile Takeover With a 51% Attack: Wild West Style in the Blockchain Realm!



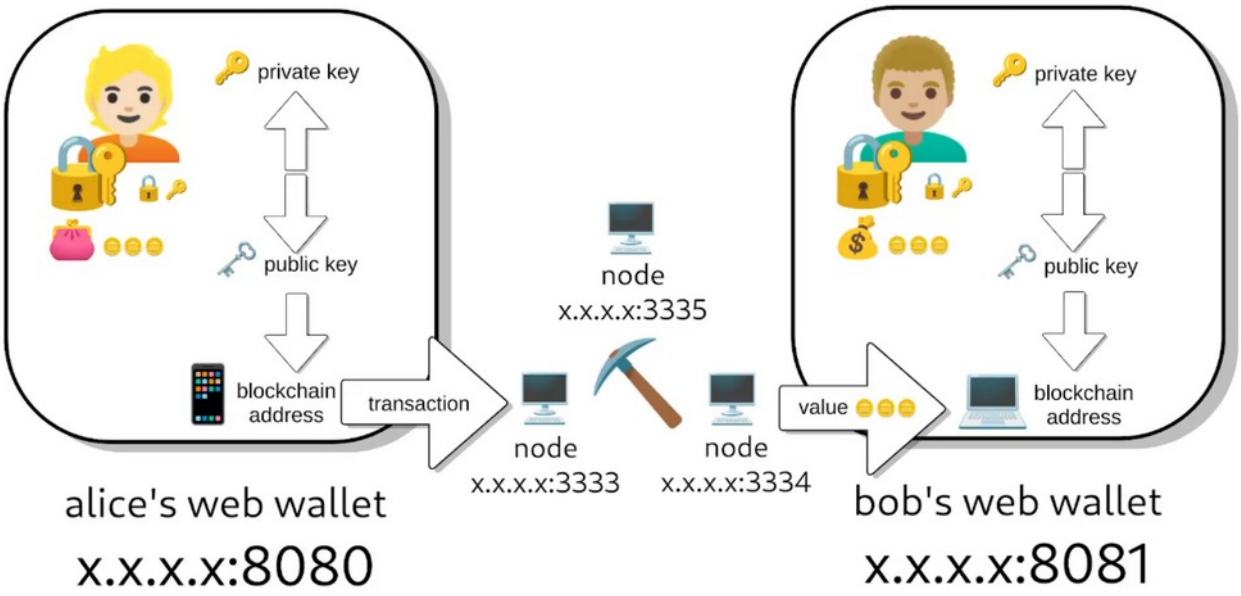
The very possibility of a 51% attack is the reason that many cryptocurrency exchanges require that a transaction first combine a certain number of confirmations of the block containing it on them before a value is condsidered as assigned to an address. Until then, they show the transaction, whether deposit or withdrawal, as pending.

Basically, this measure is no longer necessary today and persists merely for historical reasons. In purely mathematical and statistical terms, far more than 51% of the network would have to be under control of one entity and the actual hashing power is just to high that anybody had a reasonable chance to perform a successful 51% attack. That would be more like a 67% attack. Not only would the effort involved be considerable, it is also not really feasible due to the decentralized nature of mining.

Section 6 – Final Demo, Notes and Some Words to Say Goodbye

I don't know where I am going, but I am on my way.
Voltaire

Lecture 53 – Mission Accomplished or Ta-Daa: A Transaction as Final Demonstration



How to setup the automated start of competetive mining efforts.

[BLOCKCHAIN/BLOCKCHAIN.GO RUN()]

```
func (bc *Blockchain) Run() {
	bc.StartSyncNeighbors()
	bc.ResolveConflicts()
	// bc.StartMining()
}
```

Actually if you don't have transactions you don't let mining happen. That's nice in development stage but here is where you want some at least a tiny little bit more realitic conditions occurs.

[BLOCKCHAIN/BLOCKCHAIN.GO MINING()]

```
func (bc *Blockchain) Mining() bool {
	bc.mux.Lock()
	defer bc.mux.Unlock()

	// if len(bc.TransactionPool()) == 0 {
	//   return false
	// }
...
```

If one does not actually have any value stored to their address they cannot spend it. To solve the so called double spend problem was the reason to invent the blockchain in the first time. So let us also take that into account in

[BLOCKCHAIN/BLOCKCHAIN.GO ADDTRANSACTION()]

```
	if bc.VerifyTransactionSignature(senderPublicKey, s, t) {

		// if bc.CalculateTotalAmount(sender) < value {
		//   log.Println("Error: not enough balance in wallet")
		//   return false
		// }
...
```

That's by far more realistic but also means that you cannot have negative values in your wallets any more. Only after mining or receivinig cryptocurrency, respectively a value stored on an address you have permission at your command by being the proud owner of the matching private key to create a signature if you need one for a transaction. We will solve that problem on the fly during the following demonstration.

[WALLET_SERVER/TEMPLATES/INDEX.HTML]

Here you can take care that the amount of your address shown in your online wallet is updated every few seconds. I did this already by some kind of autoreloading this value. If you want to do that manually you need to comment out the `setInterval()` and remove the comments here above and for the reload button as well. I had good experiences with this setting and keep it as it is.

[START BLOCKCHAIN NODES -PORT=3333, 3334, and 3335, WAIT NEIGHBORHOOD SYNC]

[BROWSER LOCALHOST:8080]

[BROWSER LOCALHOST:8081]

[BROWSER LOCALHOST:3333/CHAIN]

[BROWSER LOCALHOST:3334/CHAIN]

[BROWSER LOCALHOST:3335/CHAIN]

[TRANSACTION "A" TO "B", VALUE 137]

[CHECKOUT 3333, 3334, 3335 CHAIN]

[CEHCKOUT AN ADDRESS FROM MINING NODE; USE PRIV KEY, PUB KEY AND ADDRESS AS WALLET A]

[TRANSACTION "A" TO "B", VALUE XX]

[CHECKOUT 3333, 3334, 3335 BOTH TRANSACTIONS AND CHAIN]

[CHECK WALLET "B", VALUE XX]

Code Lecture 53

Lecture 54 – Disclaimer: This is not the basis for a production system or a cryptocurrency!

If you are a programmer or developer, you have probably already recognized the weaknesses of this small application. It is only intended as a rough skeleton to apply the principles of Beckchain technology. The application is far from a production system. Error handling, database connectivity, tests both unit and table tests, advanced routing and handlers, a jQuery independent frontend, there is a lot to do.

And if you know some programming but were interested in this course in the role of a project manager or interdisciplinary from a whole other field, hopefully you could gain interesting insights into the working methods of blockchain applications.

But don't forget that this little application has more holes than a Swiss cheese–be it lacks of security, missing validations, logical errors, or even programming approaches. But this is not such a bad thing, because an application that is not complete, but covers the essentials of its use case, offers a lot of potential for improvement and, of course, for improving your skills and abilities.

The goal of this course was not only to follow principles of a blockchain application by creating a demo, but also to understand the inner workings and, in the best case, to gain a truly sublime experience by creating a piece of blockchain technology from from scratch by yourself. I believe, one or the other way you have achieved one or more of this goals.

Lecture 55 – Words of Farewell: Keep Coding, Keep Decentralizing, Keep Shaping Your Tomorrow!

Congratulations! You've made it to the end of this course. I hope you learned as much as I did when I developed this course – here and there some about myself at least.

But you should have achieved a few skills and gathered the tools and knowledge to program your own working blockchain application. Remember, it's just a start – image it like a springboard. The world of blockchain is full of possibilities and innovation. Cryptocurriencies are only one possible use case. There a so much more. Image a world wide patent achive office, supply chains and complience audits, industries like publishing, music or film industry who deal with interlectual properties law issues, a free notary who keeps immutable records of real estate property ownerships, or just the next election where afterwards you and only you can check whether your vote has counted or not, but where everyone can publicly verify, down to every single vote far beyond the shadow of a doubt, which candidate or bill has received the most votes.

Thank you for your willingness to learn and your dedication. Take a look together at what you have achieved during this course. From the basics of blockchain, over nodes and wallet, to implementing a functional application, we have covered a lot. But this can be just the beginning of your journey.

I encourage you to keep learning, experimenting and deepening your skills in the world of blockchain. Who knows, you might soon be developing the next ground-breaking application and might be the one who makes web3 being much more than an empty buzzword!

See you next time and happy coding! 👩‍💻