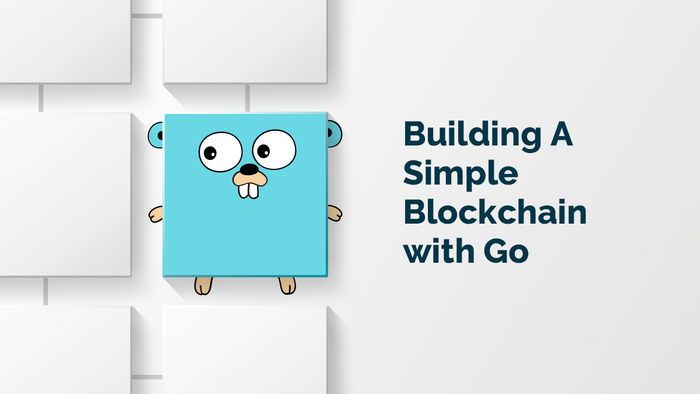
**Building A Simple Blockchain with Go**

Published Jun 12, 2018Last updated Jul 07, 2018

Building A Simple Blockchain with Go

In this tutorial, I'll attempt to demystify the broad concept of the blockchain by helping you write a simple blockchain in Go.

From this tutorial you should be able to:

* Understand blockchain terminology
* Create your own simple blockchain
* Understand what blocks are and how to create one
* Understand how to maintain the integrity of the blockchain

You can find the source code for this tutorial in this [GitHub Repo](https://github.com/codehakase/blockchain-golang-tutorial" \t "_blank).

Blockchain: a digital ledger of records arranged in smaller sets called blocks. These blocks are linked with one another through a cryptographic hashing. Each block contains a hash pointing to a previous block.  
The blockchain is useful for cryptocurrencies because of its decentralized nature, meaning the data stored isn't in a single location, but is accessible by everyone and at same time, immutable by anyone.

**Building a Simple Blockchain**

In this tutorial, we're going to create a sample blockchain for a library system. Our blockchain will store blocks containing data for a book's checkout activities. The flow for this implementation is as follows:

1. Add a new Book
2. Create a Genesis Block for a Book
3. Add Checkout data to the blockchain

This is a single node, non-complex blockchain, with everything stored in memory on runtime.

**Blocks**

In a blockchain, a block stores valuable information. This information can be a transaction or some other information needed by the system implementing the blockchain —for instance, transaction timestamps, or a hash from a previous block.  
We'll proceed to defining the data model for each of the blocks, and the checkout info making up the blockchain:

package main

*// Block contains data that will be written to the blockchain.*

**type** Block **struct** {

Pos **int**

Data BookCheckout

Timestamp **string**

Hash **string**

PrevHash **string**

}

*// BookCheckout contains data for a checked out book*

**type** BookCheckout **struct** {

BookID **string** `json:"book\_id"`

User **string** `json:"user"`

CheckoutDate **string** `json:"checkout\_date"`

IsGenesis **bool** `json:"is\_genesis"`

}

*// Book contains data for a sample book*

**type** Book **struct** {

ID **string** `json:"id"`

Title **string** `json:"title"`

Author **string** `json:"author"`

PublishDate **string** `json:"publish\_date"`

ISBN **string** `json:"isbn:`

}

From the Block struct, Pos holds the position of the data in the chain. Data is the valuable information contained in the block (in this case the checkout item). Timestamp holds the current timestamp of the block creation. Hash is the generated hash of the block. PrevHash stores the hash of the previous block.

With the Block struct defined, we need to think about hashing the blocks. The hashes are used to identify and keep the blocks in the right order. Calculating hashes is a very important feature of the blockchain. Calculating hashes is a difficult operation (compute wise). The difficulty in creating hashes was a well-thought-out architectural design decision, as it makes adding new blocks difficult, preventing mutable operations after they've been added.

**Hashing and Generating Blocks**

We'll start off with a simple hashing method and write a function calculateHash to concatenate the block fields and create a SHA-256 hash:

**func** (b \*Block) **generateHash**() {

*// get string val of the Data*

bytes, \_ := json.Marshal(b.Data)

*// concatenate the dataset*

data := **string**(b.Pos) + b.Timestamp + **string**(bytes) + b.PrevHash

hash := sha256.New()

hash.Write([]**byte**(data))

b.Hash = hex.EncodeToString(hash.Sum(nil))

}

Next, let's write another function CreateBlock to create a new block.

**func** **CreateBlock**(prevBlock \*Block, checkoutItem BookCheckout) \***Block** {

block := &Block{}

block.Pos = prevBlock.Pos + 1

block.Timestamp = time.Now().String()

block.Data = checkoutItem

block.PrevHash = prevBlock.Hash

block.generateHash()

**return** block

}

The CreateBlock function does exactly as it states — it creates a new Block. It requires two parameters for this to happen — the previous block and the checkout item to add. If you’ve noticed, we aren't doing any form of checks on the parameters, just to keep things simple.

**Creating the Blockchain**

We've created the struct for the Block and a function to create one. We'll implement a blockchain to hold a list of those blocks, and a function to add a block to the blockchain.

*// Blockchain is an ordered list of blocks*

**type** Blockchain **struct** {

blocks []\*Block

}

*// BlockChain is a global variable that'll return the mutated Blockchain struct*

**var** BlockChain \*Blockchain

*// AddBlock adds a Block to a Blockchain*

**func** (bc \*Blockchain) **AddBlock** (data BookCheckout) {

*// get previous block*

prevBlock := bc.blocks[len(bc.blocks)-1]

*// create new block*

block := CreateBlock(prevBlock, data)

bc.blocks = append(bc.blocks, block)

}

**The Genesis Block**

In a blockchain, the Genesis Block is the first item in the chain. To add a new block, we must first check for an existing block. If there isn’t one, a Genesis Block is created. Let's write a function to create a new Genesis Block.

**func** **GenesisBlock**() \***Block** {

**return** CreateBlock(&Block{}, BookCheckout{IsGenesis: true})

}

We also need to write a function to create a new blockchain:

**func** **NewBlockchain**() \***Blockchain** {

**return** &Blockchain{[]\*Block{GenesisBlock()}}

}

The NewBlockchain function returns the Blockchain struct with a Genesis Block. Since we aren't taking into measures data persistence for the blockchain, as that'll be out of the scope for this tutorial, we always start at a fresh group by generating a Genesis Block whenever the program is run.

**Validation**

Before we run our blockchain app, we need to somehow implement a validation so the blocks aren't saved while already mutated. We'll create a helper function validBlock and make use of it in the AddBlock method attached to the Blockchain struct:

**func** **validBlock**(block, prevBlock \*Block) **bool** {

*// Confirm the hashes*

**if** prevBlock.Hash != block.PrevHash {

**return** false

}

*// confirm the block's hash is valid*

**if** !block.validateHash(block.Hash) {

**return** false

}

*// Check the position to confirm its been incremented*

**if** prevBlock.Pos+1 != block.Pos {

**return** false

}

**return** true

}

**func** (b \*Block) **validateHash**(hash **string**) **bool** {

b.generateHash()

**if** b.Hash != hash {

**return** false

}

**return** true

}

Our AddBlock method should look like this:

**func** (bc \*Blockchain) **AddBlock** (data BookCheckout) {

*// get previous block*

prevBlock := bc.blocks[len(bc.blocks)-1]

*// create new block*

block := CreateBlock(prevBlock, data)

*// validate integrity of blocks*

**if** validBlock(block, prevBlock) {

bc.blocks = append(bc.blocks, block)

}

}

We've written the main parts of our blockchain so far! Let's create a web server so we can communicate with our blockchain and test it out.

In our main function, we'll write the code needed to create a web server and register routes to communicate with blockchain methods. We'll be using [Gorilla Mux](https://github.com/gorilla/mux) for routing and creating the server mux:

**func** **main**() {

*// register router*

r := mux.NewRouter()

r.HandleFunc("/", getBlockchain).Methods("GET")

r.HandleFunc("/", writeBlock).Methods("POST")

r.HandleFunc("/new", newBook).Methods("POST")

log.Println("Listening on port 3000")

log.Fatal(http.ListenAndServe(":3000", r))

}

In our main function, we have a router and three(3) routes and handlers defined. We'll create those handlers now.

The getBlockchain handler will simply write back the blockchain as a JSON string in the browser:

**func** **getBlockchain**(w http.ResponseWriter, r \*http.Request) {

jbytes, err := json.MarshalIndent(BlockChain.blocks, "", " ")

**if** err != nil {

w.WriteHeader(http.StatusInternalServerError)

json.NewEncoder(w).Encode(err)

**return**

}

*// write JSON string*

io.WriteString(w, **string**(jbytes))

}

The writeBlock handler adds a new block with the data sent.

**func** **writeBlock**(w http.ResponseWriter, r \*http.Request) {

**var** checkoutItem BookCheckout

**if** err := json.NewDecoder(r.Body).Decode(&checkoutItem); err != nil {

w.WriteHeader(http.StatusInternalServerError)

log.Printf("could not write Block: %v", err)

w.Write([]**byte**("could not write block"))

**return**

}

*// create block*

BlockChain.AddBlock(checkoutItem)

resp, err := json.MarshalIndent(checkoutItem, "", " ")

**if** err != nil {

w.WriteHeader(http.StatusInternalServerError)

log.Printf("could not marshal payload: %v", err)

w.Write([]**byte**("could not write block"))

**return**

}

w.WriteHeader(http.StatusOK)

w.Write(resp)

}

We'll write the last handler newBook, which creates new Book data so we'll use the ID generated to add as blocks. Remember the flow:

1. Add a new Book
2. Create a Genesis Block for a Book
3. Add Checkout data to the blockchain

**func** **newBook**(w http.ResponseWriter, r \*http.Request) {

**var** book Book

**if** err := json.NewDecoder(r.Body).Decode(&book); err != nil {

w.WriteHeader(http.StatusInternalServerError)

log.Printf("could not create: %v", err)

w.Write([]**byte**("could not create new Book"))

**return**

}

*// We'll create an ID, concatenating the ISDBand publish date*

*// This isn't an efficient way but it serves for this tutorial*

h := md5.New()

io.WriteString(h, book.ISBN+book.PublishDate)

book.ID = fmt.Sprintf("%x", h.Sum(nil))

*// send back payload*

resp, err := json.MarshalIndent(book, "", " ")

**if** err != nil {

w.WriteHeader(http.StatusInternalServerError)

log.Printf("could not marshal payload: %v", err)

w.Write([]**byte**("could not save book data"))

**return**

}

w.WriteHeader(http.StatusOK)

w.Write(resp)

}

With all three handlers written, let's clean up our main function. Our main function should look like so:

**func** **main**() {

*// initialize the blockchain and store in var*

BlockChain = NewBlockchain()

*// register router*

r := mux.NewRouter()

r.HandleFunc("/", getBlockchain).Methods("GET")

r.HandlerFunc("/", writeBlock).Methods("POST")

r.HandlerFunc("/new", newBook).Methods("POST")

*// dump the state of the Blockchain to the console*

**go** **func**() {

**for** \_, block := **range** BlockChain.blocks {

fmt.Printf("Prev. hash: %x\n", block.PrevHash)

bytes, \_ := json.MarshalIndent(block.Data, "", " ")

fmt.Printf("Data: %v\n", **string**(bytes))

fmt.Printf("Hash: %x\n", block.Hash)

fmt.Println()

}

}()

log.Println("Listening on port 3000")

log.Fatal(http.ListenAndServe(":3000", r))

}

**Almost done!**

With our updated code, let's fire up our application: go run main.go



Head over to [http://localhost:3000](http://localhost:3000%5D/). You'll see the Genesis Block displayed:



Let's add a new Book so we'll have an ID to use in add blocks. I'm going to use cURL from a terminal. Postman is also a good tool for this too:

$ curl -X POST http:*//localhost:3000/new \*

-H "Content-Type: application/json" \

-d '{"title": "Sample Book", "author":"John Doe",

"isbn":"909090","publish\_date":"2018-05-26"}'

After creating a new book, we get a payload back with the generated ID. Take note of that, as it’s always needed to create the blocks.  
To add a checkout record, which will be stored in the blockchain, we send a POST request to the root endpoint, [http://localhost:3000](http://localhost:3000/), with a payload for the checkout item:

$ curl -X POST http:*//localhost:3000 \*

-H "Content-Type: application/json" \

-d '{"book\_id": "generated\_id", "user": "Mary Doe",

"checkout\_date":"2018-05-28"}'

Refreshing the browser, we'll see the new item added with its own hash created:  


**We made it!!**

Congrats, pal! You've come a long way. You just wrote your first Blockchain prototype!!! It’s good to note that the actual blockchain is much more complex compared to the implementation we did above. The implementation in this tutorial made adding new blocks very easy, which isn't the case for the real one. Adding new blocks requires some heavy computations (Proof of Work).

With the concepts explained, you should have a greater understanding of the blockchain. There are other topics that are prerequisites to understanding the basis of the blockchain, like Proof of Stake, Proof of Work, Smart Contracts, DApps, etc.

You can get the source code for this tutorial on this [GitHub Repo](https://github.com/codehakase/blockchain-golang-tutorial" \t "_blank).

If you have questions, you can let me know in the comments or send me a tweet [@codehakase](https://www.codementor.io/codehakase/@codehakase) 

Creating block-chain via javascript

In this tutorial, we'll write a very tiny blockchain in JavaScript. It won't be too fancy, but it’ll be enough to give you an understanding of how a blockchain works.

We'll call this blockchain SavjeeCoin. Let's get started!

# Set up the Blockchain

We'll start by creating a new JavaScript file to store all of our code. We'll call it main.js and let's start by defining what a blockchain and blocks should look like.

Let's begin with creating a Block class and giving it a constructor. When you create a new block, you need to pass it a timestamp, some data and the hash of the block that went before it:

**class** **Block**{

**constructor**(timestamp, data, previousHash = '') {

**this**.previousHash = previousHash;

**this**.timestamp = timestamp;

**this**.data = data;

}

}

Here’s what each property means:

* The timestamp tells us when the block was created. You can use any format you like (in this example we'll use a UNIX timestamp)
* The data parameter can include any type of data that you want to associate with this block. If you want to build a cryptocurrency you can store transaction details in here like sender/receiver and the amount of money that was transferred.
* The previousHash is a string that contains the hash of the previous block. This is what will create the chain of blocks and will be very important to ensure the integrity of our blockchain later.

# Hashes

Each block points towards the previous block (that's why we have the previousHashattribute). That means that each block needs a hash.

A hash is basically like a fingerprint. It's unique for each block. You can calculate a hash of a block by taking all of its contents and running it through a hash function.

So let's start by writing a method that can calculate the hash of the current block. So inside the Block class, we create a calculateHash function:

calculateHash() {

**return** SHA256(**this**.previousHash + **this**.timestamp + JSON.stringify(**this**.data)).toString();

}

Notice that I choose to use the SHA256 hash function. However, the SHA256 hash is not available in JavaScript and requires you to use an external library. I found that [crypto-js](https://www.npmjs.com/package/crypto-js) is a great library and contains secure implementations for different hash functions. Install it with npm like this:

npm **install** crypto-**js**

Then afterwards we can import it in our main.js file like so:

**const** SHA256 = require("crypto-js/sha256");

Now that we have our calculateHash() method, let's use it in the constructor of our Block:

**class** **Block**{

**constructor**(timestamp, data, previousHash = '') {

**this**.previousHash = previousHash;

**this**.timestamp = timestamp;

**this**.data = data;

*// When creating a new Block, automatically calculate its hash.*

**this**.hash = **this**.calculateHash();

}

}

# Blockchain class

After defining what a Block looks like, we can define what a Blockchain should look like. So let's create a new class for that:

**class** **Blockchain**{

**constructor**() {

**this**.chain = [];

}

}

In this case, the blockchain is a very simple object that contains a property chain. This is an array containing all the blocks on the chain.

Before we can add new blocks, we have to add what is called a "genesis block". This is the first block on the chain and it's a bit special because it cannot point to a previous block (it's the first one!)

So to create the genesis block, I'll add a method to our class called createGenesisBlock():

createGenesisBlock()

**return** **new** Block("01/01/2017", "Genesis block", "0");

}

Back in the constructor of our Blockchain class, we can now add the genesis block whenever we create a new Blockchain instance:

**class** **Blockchain**{

**constructor**() {

**this**.chain = [**this**.createGenesisBlock()];

}

# More Blockchain methods

Now, let’s add some methods to our Blockchain class to allow us to do basic things like adding new blocks and getting the latest block.

The getLatestBlock method is the most simple one. It just returns the last element on the chain array:

getLatestBlock(){

**return** **this**.chain[**this**.chain.length - 1];

}

The addBlock method is a bit more complicated. Before we can add a new block to our chain, we have to correctly set the previousHash property of that block. It has to be set to the hash of the latest block on our chain. And we also have to calculate the hash of the new block:

addBlock(newBlock){

*// The new block needs to point to the hash of the latest block on the chain.*

newBlock.previousHash = **this**.getLatestBlock().hash;

*// Calculate the hash of the new block*

newBlock.hash = newBlock.calculateHash();

*// Now the block is ready and can be added to chain!*

**this**.chain.push(newBlock);

}

Just a side note: in reality, adding blocks to a blockchain requires you to "mine" it. That's something we'll add in part 2 of this post (so keep reading!)

# Testing

Let's test it. To do that, create an instance of the blockchain:

**let** savjeeCoin = **new** Blockchain();

And let's add a few blocks:

savjeeCoin.addBlock(**new** Block("20/07/2017", { amount: 4 }));

savjeeCoin.addBlock(**new** Block("22/07/2017", { amount: 10 }));

There — we have created two new blocks. Let's see what our blockchain now looks like. We'll stringify SavjeeCoin and use four spaces to format it:

console.log(JSON.stringify(savjeeCoin, null, 4));

The output is what our blockchain looks like right now. Our blockchain's object contains a property chain (an array), which contains all of our blocks.

# Verify the integrity of the blockchain

Great! Now that we know it works, let's try to do something different. Blockchains are great because once a block is added, it cannot be changed without invalidating the rest of the chain.

In this implementation, however, there is no way for me to verify the integrity of our blockchain. Let's add a new method to our blockchain, called isChainValid. This will return true if the chain is valid or false if something is wrong:

isChainValid(){

}

In order to verify the integrity, we have to loop over the entire chain (not looking at the genesis block). Inside this loop we're going to check two things:

* Is the block's hash still valid? To check that we will recalculate the hash of each block. If something changed inside a block, it will change the hash of that block.
* Does each block point to the correct previous block? To check this, we will see if the previousHash of the block, equals the hash property of the block that went before it.
* Is the genesis block still intact or has it been tampered with?

Here is the full method:

isChainValid(){

**for** (**let** i = 1; i < **this**.chain.length; i++){

**const** currentBlock = **this**.chain[i];

**const** previousBlock = **this**.chain[i - 1];}

*// Recalculate the hash of the block and see if it matches up.*

*// This allows us to detect changes to a single block*

**if** (currentBlock.hash !== currentBlock.calculateHash()) {

**return** false;

}

*// Check if this block actually points to the previous block (hash)*

**if** (currentBlock.previousHash !== previousBlock.hash) {

**return** false;

}

}

*// Check the genesis block*

**if**(**this**.chain[0] !== **this**.createGenesisBlock()){

**return** false;

}

*// If we managed to get here, the chain is valid!*

**return** true;

}

# Testing integrity

Now we can test the integrity of our blockchain by running:

console.log('Blockchain valid? ' + savjeeCoin.isChainValid());

If we run it now, it will say that yes, our chain is indeed valid (we didn't tamper with it).

Now, let's try and tamper with our blockchain. Let's change block 2 and overwriting its data (let's say we want to pretend that we transferred 100 coins instead of 4):

*// Tamper with the chain!*

savjeeCoin.chain[1].data = { amount: 100 };

*// Check if it's valid again*

console.log('Blockchain valid? ' + savjeeCoin.isChainValid()); *// will return false!*

When we run this, notice that the program has detected our attempt to tamper with the chain!

However, you might think that there's another way I can tamper with this. I've changed the block’s data, but I didn't recalculate the hash. So you might try to be clever, take that same block, and recalculate its hash.

*// Tamper with the chain!*

savjeeCoin.chain[1].data = { amount: 100 };

*// Recalculate its hash, to make everything appear to be in order!*

savjeeCoin.chain[1].hash = savjeeCoin.chain[1].calculateHash():

However, when we run this again, it till detects that we tampered with the chain. That's because block 2 now has a previousHash attribute that doesn't match the hash of block 1.

# Summary

So that was it for our simple blockchain implementation! It allows us to add new blocks and detect when someone is tampering with the data inside the chain.

However, it's far from complete!

# Next up

In the next part of this series we will take a look at how we can implement the proof-of-work algorithm so we can prevent people from adding a lot of spam blocks to our chain and to make tampering even harder:

Continue reading: "[Implementing proof-of-work (Blockchain in JavaScript, part 2)](https://www.codementor.io/savjee/implementing-proof-of-work-blockchain-in-javascript-part-2-k9ozymkqw)"

# Source code

The [source code is available on GitHub](https://github.com/SavjeeTutorials/SavjeeCoin/blob/master/src/main.js) and there’s a [video tutorial](https://www.youtube.com/watch?v=zVqczFZr124&list=PLzvRQMJ9HDiTqZmbtFisdXFxul5k0F-Q4) if you want to follow along step-by-step.