

Implement Simple Proof-of-Work Mining in JS (Extended)

Exercise:

Build a Simple Blockchain with
Proof-of-Work Miner in JS

You will create a simple blockchain network – **from scratch** – in JavaScript and add **proof-of-work mining** inside its code. The basic concept of blockchain is quite simple: a distributed ledger that maintains a continuously growing list of ordered records (blocks).

Most of the code is based on this project: <https://github.com/lhartikk/naivechain>. Thanks to the original authors.

For this exercise, you will need **Linux** or Linux-like command-line environment to follow. We used the **Ubuntu** distribution. However, if this is not available, it's okay, other operating systems will also work fine in most cases.

Project Dependencies

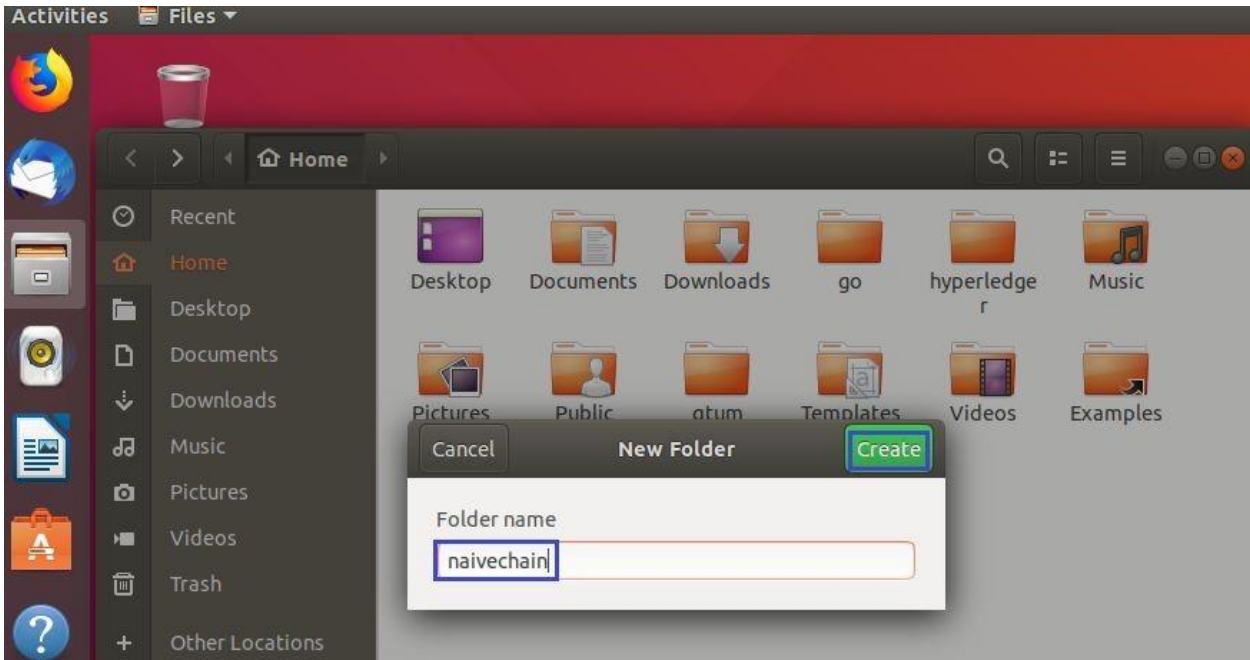
- NodeJS **v12.18** <https://nodejs.org/en/download/>
- NPM **v6.14**
- GIT **v2.28** <https://git-scm.com/downloads>

You may choose only one

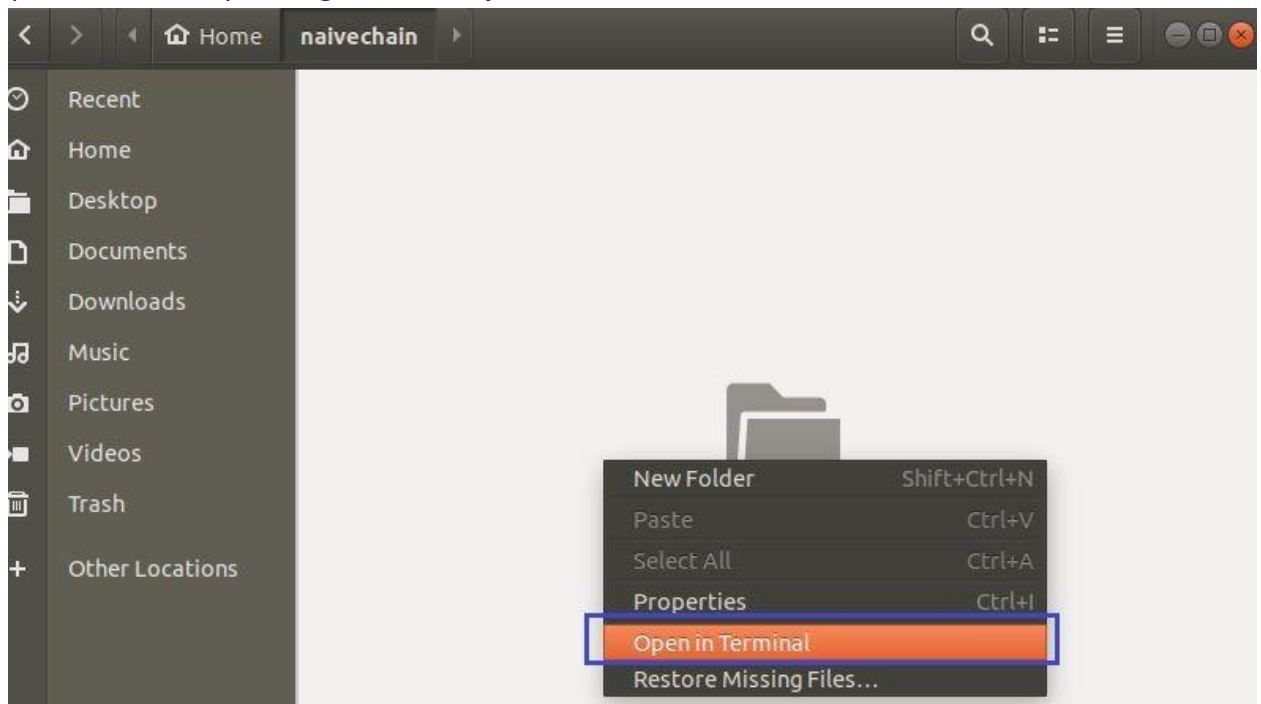
- Postman **v7.31** <https://www.postman.com/downloads/>
- cURL **v7.72** <https://curl.haxx.se/download.html>

1. Setup the project

1. **Create a directory** for the project. Name it “naivechain”.

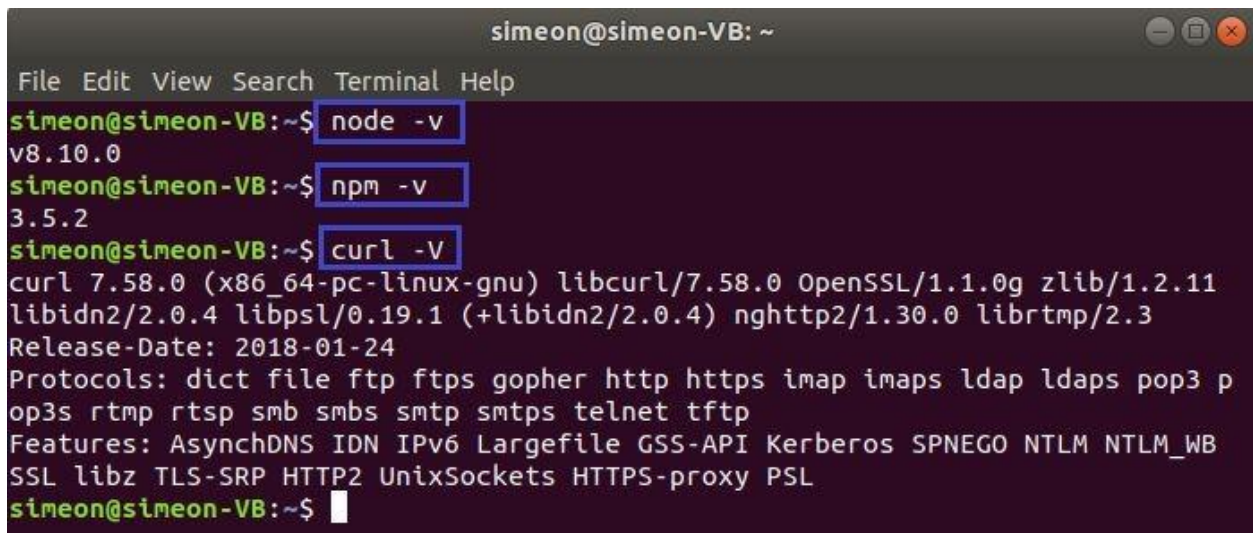


2. Open the directory and right click to **open** it in the **terminal**.



3. First **check the versions** of "**node.js**", "**npm**" and "**cURL**". In terminal type: "**node -v**", "**npm -v**" and "**curl -V**" (for cURL type capital "V" for version). At this point, if you haven't installed these dependencies yet, please do.

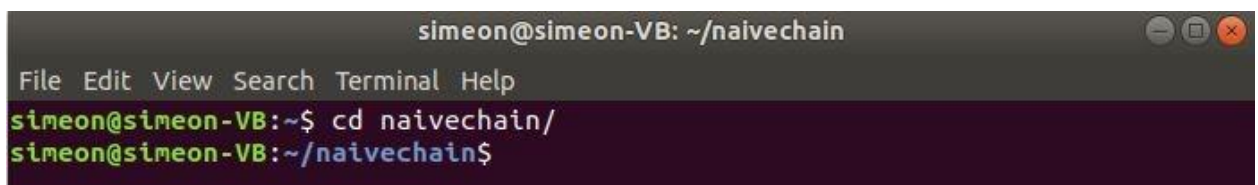
```
node -v
npm -v
curl -V
```

A terminal window titled 'simeon@simeon-VB: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The user enters 'node -v' and receives 'v8.10.0'. Then they enter 'npm -v' and receive '3.5.2'. Finally, they enter 'curl -V' and receive a detailed output including 'curl 7.58.0 (x86_64-pc-linux-gnu) libcurl/7.58.0 OpenSSL/1.1.0g zlib/1.2.11 libidn2/2.0.4 libpsl/0.19.1 (+libidn2/2.0.4) nghttp2/1.30.0 librtmp/2.3 Release-Date: 2018-01-24' followed by a list of protocols and features.

```
simeon@simeon-VB: ~$ node -v
v8.10.0
simeon@simeon-VB: ~$ npm -v
3.5.2
simeon@simeon-VB: ~$ curl -V
curl 7.58.0 (x86_64-pc-linux-gnu) libcurl/7.58.0 OpenSSL/1.1.0g zlib/1.2.11
libidn2/2.0.4 libpsl/0.19.1 (+libidn2/2.0.4) nghttp2/1.30.0 librtmp/2.3
Release-Date: 2018-01-24
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 p
op3s rtmp rtsp smb smbs smtp smtps telnet tftp
Features: AsynchDNS IDN IPv6 Largefile GSS-API Kerberos SPNEGO NTLM NTLM_WB
SSL libz TLS-SRP HTTP2 UnixSockets HTTPS-proxy PSL
simeon@simeon-VB: ~$
```

4. Then go to "**naivechain**" directory.

```
cd naivechain
```

A terminal window titled 'simeon@simeon-VB: ~/naivechain' with a menu bar (File, Edit, View, Search, Terminal, Help). The user enters 'cd naivechain/' and the prompt changes to 'simeon@simeon-VB: ~/naivechain\$'.

```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB: ~$ cd naivechain/
simeon@simeon-VB: ~/naivechain$
```

5. Now we will need to initialize the directory as a **npm** project. To do that, enter this command in the terminal.

```
npm init -y
```



6. Install project packages by issuing this command:

```
npm install body-parser@1.15.2 crypto-js@3.1.6 express@4.11.1 ws@1.1.0
```

2. Implement a simple blockchain

1. **Create** the file “**main.js**” with your preferred text editor or IDE. Import the libraries assigned to relevant variables.

```
'use strict';  
var CryptoJS = require("crypto-js");  
var express = require("express");  
var bodyParser = require('body-parser');  
var WebSocket = require("ws");
```

2. Now, let's pay attention to the **block structure**. To keep things as simple as possible our blockchain contains only the most necessary elements: **index, timestamp, data, hash and previous hash**. Following the blockchain concept, the **hash** of the previous block must be **found** in the block to **preserve** the chain integrity.



3. Now, let's create **class Block** in the code with constructor and **five fields**.

```
class Block {  
  constructor(index, previousHash, timestamp, data, hash) {  
    this.index = index;  
    this.previousHash = previousHash.toString();  
    this.timestamp = timestamp;  
    this.data = data;  
    this.hash = hash.toString();  
  }  
}
```

4. An in-memory JavaScript **array** is used to **store the blockchain**. The **first block** of the blockchain is conventionally called “**genesis-block**” and is hardcoded.

We take it with the function which returns a new Block with the usual attributes.

- The Block index is 0,
- The “**previousHash**” don’t exist in reality and we give him a string value “0”.
- The current Block **hash** is hardcoded
- The **data field** contains the string “**my genesis block**”

```
var getGenesisBlock = () => {  
    return new Block(0, "0", 1465154705, "my genesis block!!",  
        "816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7");  
};  
  
var blockchain = [getGenesisBlock()];
```

We have the code snippet prepared here for you so that it would be easy to copy the exact hashes:

```
var getGenesisBlock = () => {  
    return new Block(0, "0", 1465154705, "my genesis block!!",  
        "816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7");  
};
```

5. Now we have a blockchain with its first block. And it's time to start testing our program. Let's create **testApp()** function and put there a test functions. Create **showBlockchain()** function which prints all blocks of given blockchain on the console.

```
function testApp() {  
  function showBlockchain(inputBlockchain){  
    for (let i = 0; i < inputBlockchain.length; i++) {  
      console.log(inputBlockchain[i]);  
    }  
  
    console.log();  
  }  
  
  showBlockchain(blockchain);  
}  
  
testApp();
```

The result should be our first and only genesis block:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
/usr/bin/node --inspect-brk=33265 main.js  
Debugger listening on ws://127.0.0.1:33265/63eafadb-cb0b-4d8b-9fc2-0038c1bc2ecc  
Debugger attached.  
› Block {index: 0, previousHash: "0", timestamp: 1465154705, data: "my genesis block!!", hash: "816534932c2b7154836da6afc367695e6337db8a921823784c..."}
```

6. Now implement the **getLatestBlock** function. It will give us the last element in blockchain array.

```
var getLatestBlock = () => blockchain[blockchain.length - 1];
```

7. Now create the **"calculateHash"** function in the code. This function is the **moment of mining** when the miner calculates the hash for the next block. For now, we are only calculating the has given the input data. The actual mining process will be implemented later in this exercise.

```
var calculateHash = (index, previousHash, timestamp, data) => {  
  return CryptoJS.SHA256(index + previousHash + timestamp + data).toString();  
};
```


8. When we want to calculate the hash for the block, we will use the **calculateHashForBlock** function. It will execute the **calculateHash** function for a given block and return SHA256 hash of a string which is the result of concatenating **block.index**, **block.previousHash**, **block.timestamp** and **block.data**.

```
var calculateHashForBlock = (block) => {  
  return calculateHash(block.index, block.previousHash, block.timestamp, block.data);  
};
```

9. Now let's ensure that **calculateHashForBlock** function work properly. Calculate the genesis block hash and check the function result on the console. Write a control line:

```
function testApp() {  
  /* function showBlockchain(inputBlockchain){  
    for (let i = 0; i < inputBlockchain.length; i++) {  
      console.log(inputBlockchain[i]);  
    }  
  
    console.log();  
  }  
  
  showBlockchain(blockchain);  
  */  
  console.log(calculateHashForBlock(getGenesisBlock()));  
}  
  
testApp();
```

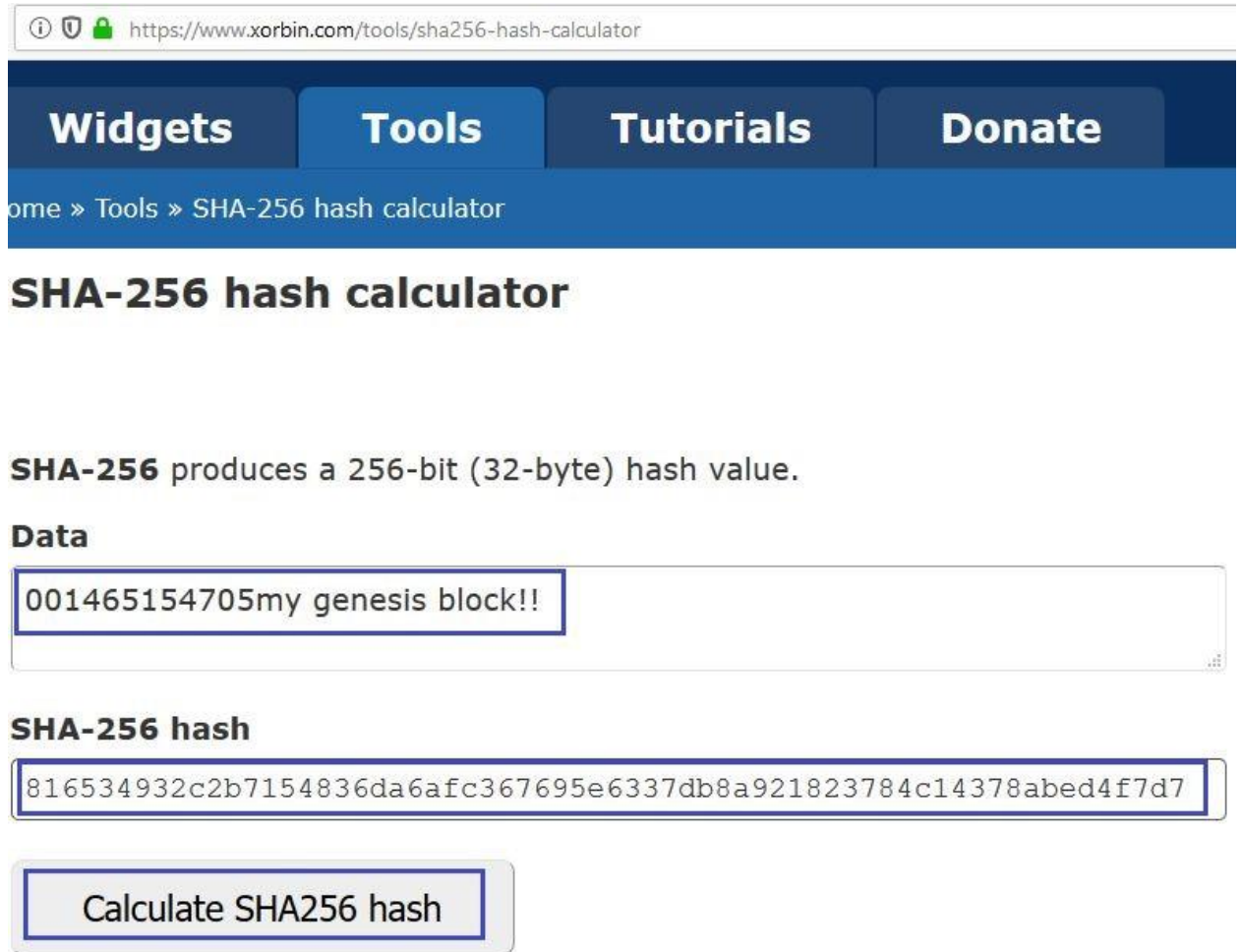
And execute the program.

The result should exactly be:

816534932c2b7154836da6afc367695e6337db8a921823784c14378abe d4f7d7

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
  
/usr/bin/node --inspect-brk=3872 main.js  
Debugger listening on ws://127.0.0.1:3872/5da6cffc-2432-4c91-a7f0-a372ce98a73d  
Debugger attached.  
816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7
```

10. Now, go to an external **SHA-256 hash calculator** tool like <https://www.xorbin.com/tools/sha256-hash-calculator> and put the **genesis block info** there and verify the result.



The screenshot shows a web browser window with the URL <https://www.xorbin.com/tools/sha256-hash-calculator>. The page has a dark blue header with navigation links: **Widgets**, **Tools** (selected), **Tutorials**, and **Donate**. Below the header, a breadcrumb trail reads "ome » Tools » SHA-256 hash calculator". The main heading is **SHA-256 hash calculator**. A descriptive text states: "SHA-256 produces a 256-bit (32-byte) hash value." Under the label **Data**, a text input field contains the string "001465154705my genesis block!!". Below this, under the label **SHA-256 hash**, a text input field displays the resulting hash: "816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7". At the bottom, there is a button labeled "Calculate SHA256 hash".

If the SHA-256 hash is the same, our function is working properly.

11. When we want to **generate** a block, we must first know the **hash of the previous block** because this is the link between chains of the blockchain. Next, we must **create** the rest of the **required content** (index, hash, data and timestamp). The **Block data** is something that is provided by the end-user which you will see later on.

```
var generateNextBlock = (blockData) => {
  var previousBlock = getLatestBlock();
  var nextIndex = previousBlock.index + 1;
  var nextTimestamp = new Date().getTime() / 1000;
  var nextHash = calculateHash(nextIndex, previousBlock.hash, nextTimestamp, blockData);
  return new Block(nextIndex, previousBlock.hash, nextTimestamp, blockData, nextHash);
};
```

12. At any given time, we must be able to **validate** if a block or a chain of blocks are valid in terms of **integrity**. This is true especially when we receive **new blocks** from other **nodes** and must decide whether to **accept** them or not.

```
var isValidNewBlock = (newBlock, previousBlock) => {
  if (previousBlock.index + 1 !== newBlock.index) {
    console.log('invalid index');
    return false;
  } else if (previousBlock.hash !== newBlock.previousHash) {
    console.log('invalid previoushash');
    return false;
  } else if (calculateHashForBlock(newBlock) !== newBlock.hash) {
    console.log(typeof (newBlock.hash) + ' ' + typeof calculateHashForBlock(newBlock));
    console.log('invalid hash: ' + calculateHashForBlock(newBlock) + ' ' + newBlock.hash);
    return false;
  }
  return true;
};
```

13. We must create a function **addBlock** for adding new blocks. It will use the **isValidNewBlock** function.

```
var addBlock = (newBlock) => {
  if (isValidNewBlock(newBlock, getLatestBlock())) {
    blockchain.push(newBlock);
  }
};
```

14. Now, let's test the **addBlock** function. Go to the testApp function and print our blockchain, then add a new block with data: "test block data" and print blockchain again.

```
function testApp() {
  function showBlockchain(inputBlockchain){
    for (let i = 0; i < inputBlockchain.length; i++) {
      console.log(inputBlockchain[i]);
    }

    console.log();
  }

  // showBlockchain(blockchain);

  // console.log(calculateHashForBlock(getGenesisBlock()));

  // addBlock Test
  console.log("blockchain before addBlock() execution:");
  showBlockchain(blockchain);
  addBlock(generateNextBlock("test block data"));
  console.log("\n");
  console.log("blockchain after addBlock() execution:");
  showBlockchain(blockchain);
}

testApp();
```

The result will be:

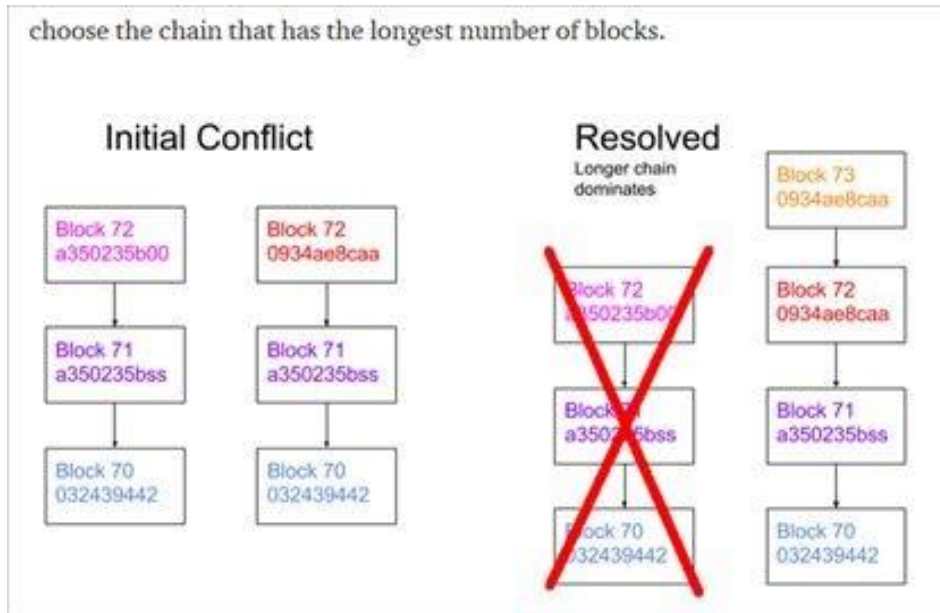
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
/usr/bin/node --inspect-brk=32558 main.js
Debugger listening on ws://127.0.0.1:32558/b24d1f73-6188-4f69-a389-d085f4c9a256
Debugger attached.
blockchain before addBlock() execution:
> Block {index: 0, previousHash: "0", timestamp: 1405154705, data: "my genesis block!!", hash: "816534932c2b7154836da6afc367695e6337db8a921823784c.."}

blockchain after addBlock() execution:
> Block {index: 0, previousHash: "0", timestamp: 1405154705, data: "my genesis block!!", hash: "816534932c2b7154836da6afc367695e6337db8a921823784c.."}
> Block {index: 1, previousHash: "816534932c2b7154836da6afc367695e6337db8a921823784c..", timestamp: 1548616869.677, data: "test block data", hash: "b377eb0a1fea96efffe4b6992c0d1409d9bb7d319cf3ce053c8.."}

```

After testing you can comment this part of the code.

15. There should always be only **one** explicit **set of blocks** in the chain at a given time. In case of **conflicts** (e.g. two nodes both generate block number 72), we choose the chain that has the **longest number of blocks**.



16. To be sure that our block is working according to our expectations, let's create an **isValidChain** function which will perform some elementary checks for network validity.

```
var isValidChain = (blockchainToValidate) => {  
  if (JSON.stringify(blockchainToValidate[0]) !== JSON.stringify(getGenesisBlock())) {  
    return false;  
  }  
  var tempBlocks = [blockchainToValidate[0]];  
  for (var i = 1; i < blockchainToValidate.length; i++) {  
    if (isValidNewBlock(blockchainToValidate[i], tempBlocks[i - 1])) {  
      tempBlocks.push(blockchainToValidate[i]);  
    } else {  
      return false;  
    }  
  }  
  return true;  
};
```

17. In our model of the blockchain, we compare the length of the **new blocks sequence** with the **length of the existing blockchain**. The **longest chain** is accepted like **valid** blockchain.


```

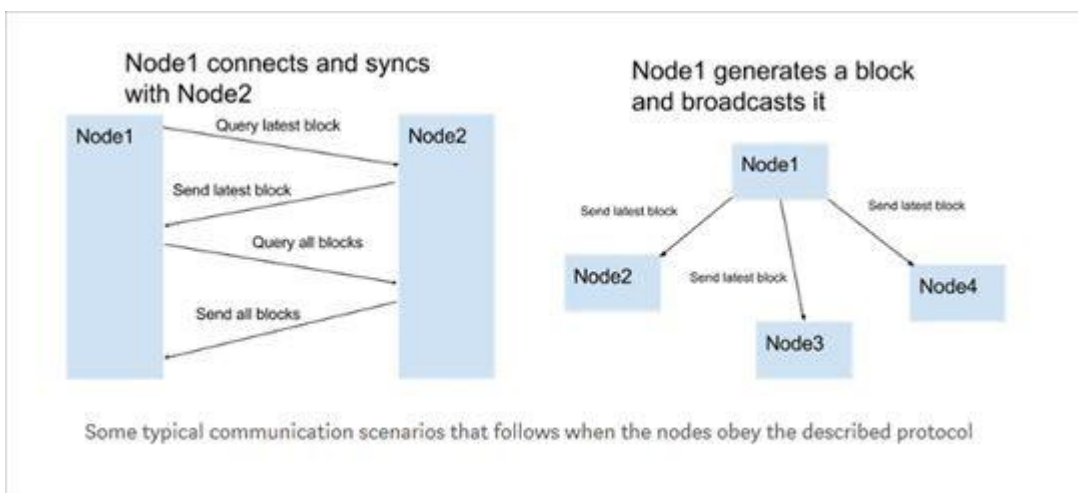
var replaceChain = (newBlocks) => {
  if (isValidChain(newBlocks) && newBlocks.length > blockchain.length) {
    console.log
      ('Received blockchain is valid. Replacing current blockchain with received blockchain');
    blockchain = newBlocks;
  } else {
    console.log('Received blockchain invalid');
  }
};

```

18. For the purpose of modeling the blockchain and mining blocks, the program has a system of **communication** between nodes. An essential part of the node is to **share** and sync the blockchain with **other nodes**. The following **rules** are used to **keep the network in sync**.

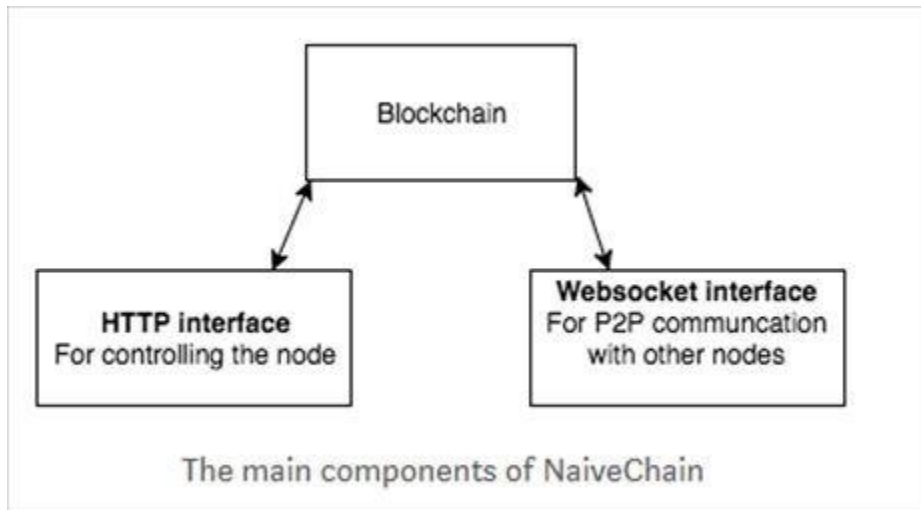
- When a node **generates a new block**, it broadcasts it to the **network**
- When a node **connects to a new peer** it queries for the latest block
- When a node encounters a block that has an **index larger than the current known block**, it either adds the block to its current chain or queries for the full blockchain.

No automatic peer discovery is used. The location (URLs) of peers must be **manually** added.



3. Create the HTTP server

1. The node in our program actually exposes **two web servers**: One for the **user to control the node (HTTP server)** and one for the **peer-to-peer communication** between the nodes. (Websocket server).



2. First, we need a variable for `http_ports` for our HTTP server. For this purpose, we will use ports from **3xxx**.

```
var http_port = process.env.HTTP_PORT || 3001;
```

Using the environment variables is the way to specify communication ports. The `process.env` global variable is injected by the Node at runtime for your application to use and it represents the state of the system environment your application is in when it starts.

We will use:

- HTTP interface to control the node
- Websockets to communicate with other nodes (P2P)

First, let's write HTTP server and test it. After that, we will write the P2P communications.

3. Next, create an **array** variable for **sockets**. We will use it to get peers.

```
var sockets = [];
```

4. The user must be able to **control the node** in some way. This is done by implementing a **HTTP REST API server**:

```
var initHttpServer = () => {
  var app = express();
  app.use(bodyParser.json());

  app.get('/blocks', (req, res) => res.send(JSON.stringify(blockchain)));
  app.post('/mineBlock', (req, res) => {
    var newBlock = generateNextBlock(req.body.data);
    addBlock(newBlock);
    console.log('block added: ' + JSON.stringify(newBlock));
    res.send();
  });
  app.get('/peers', (req, res) => {
    res.send(sockets.map(s => s._socket.remoteAddress + ':' + s._socket.remotePort));
  });
  app.post('/addPeer', (req, res) => {
    res.send();
  });
  app.listen(http_port, () => console.log('Listening http on port: ' + http_port));
};
```

With the implemented API the user is able to **interact** with the node in the following ways:

- GET requests to **/blocks** will make a **get request** and **list all blocks** in the existing blockchain
- POST requests to **/mineBlock** will send a **post request** which will **create a new block** with a content given by the user, calculate the new block's hash and **add** this new block to the blockchain. Then later it will broadcast a message to other peers.
- GET requests to **/peers** will make a **get request** and take an **array with existing peers**. If there are no peers we will receive an empty array [].
- POST requests to **/addPeer** will send a **post request** and connect **peer** to the given one

The most straightforward way to control the node is by using cURL.

For example, to **get all blocks** from the node, issue this command:

```
curl IP_OR_LOCAL_ADDRESS/blocks
```

For a host PC with address **localhost:3001** it will be:

```
curl http://localhost:3001/blocks
```

You may also use Postman if you are more comfortable working with a GUI.

5. Let's **test** the **HTTP server**. Call the **initHTTPServer** function.

```
initHttpServer();
```

Then, **run** the code (for example press F5 if you use Visual Studio Code). In the console you should see:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
C:\Program Files\nodejs\node.exe --inspect-brk=26379 main.js
Debugger listening on ws://127.0.0.1:26379/a4e5c89d-c462-4d76-880a-07a6a02edc6a
For help, see: https://nodejs.org/en/docs/inspector
Listening http on port: 3001
```

6. Our HTTP server is ready. Open the address in your browser and try to **get the blocks**. You will see the **genesis block** in our blockchain, kept by peer one.

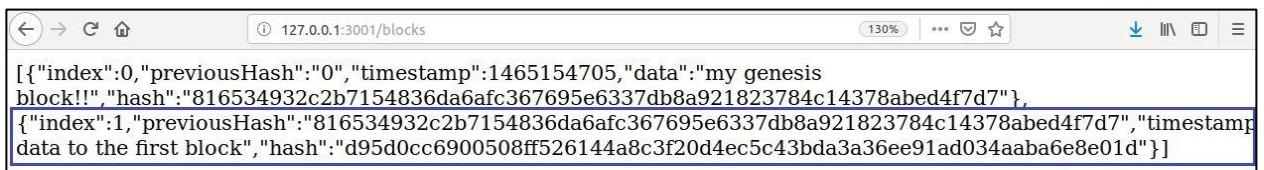
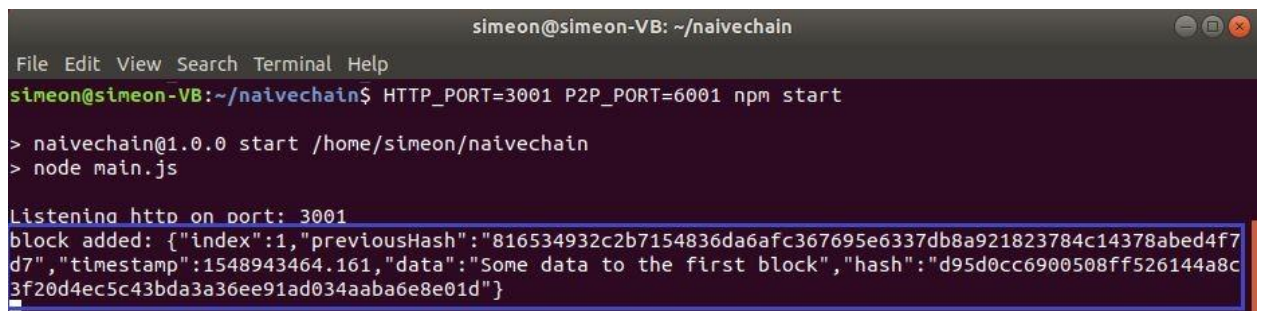
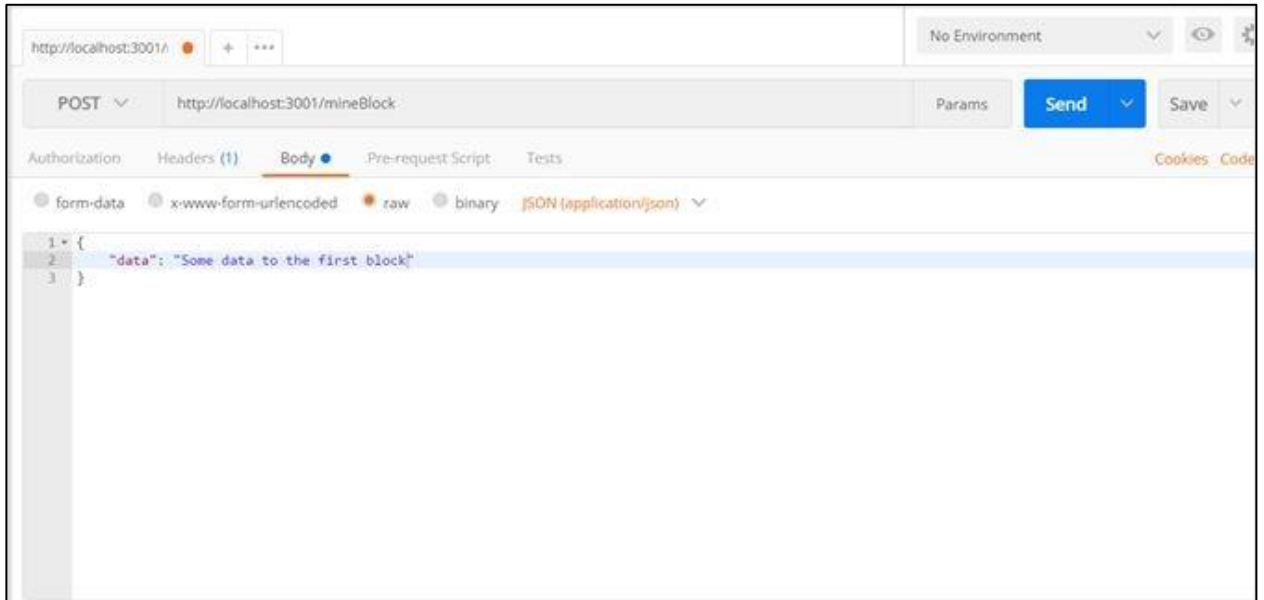
```
[{"index":0,"previousHash":"0","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}]
```

7. Now, try to command the first peer to **mine** a new block by issuing a POST request.

cURL:

```
curl -H "Content-type:application/json" --data '{"data": "Some data to the first block"}' http://localhost:3001/mineBlock
```

Postman:



The new block is here.

8. Next, try to **retrieve the peers**. Since you don't have peers yet, you will receive an **empty array**.



9. HTTP server works properly, so comment the line:

```
// initHttpServer();
```

We will initialize the HTTP server later in the code.

4. Create the P2P Websocket HTTP server

1. First, we need to declare these variables: **p2p_port** and one to **initialPeers**. Web sockets are used for real-time peer-to-peer communication between the nodes.

```
var p2p_port = process.env.P2P_PORT || 6001;
var initialPeers = process.env.PEERS ? process.env.PEERS.split(',') : [];
```

2. Then, we will create a variable for **message types**. We will use the **MessageTypes** to handle messages. Our sockets will expect the following messages:

- **QUERY_LATEST** - Query for the last block
- **QUERY_ALL** - Query for all the blocks
- **RESPONSE_BLOCKCHAIN** - Response message with the requested data (last block or all the blocks)

```
var MessageType = {
  QUERY_LATEST: 0,
  QUERY_ALL: 1,
  RESPONSE_BLOCKCHAIN: 2
};
```

3. Now, let's create the part of the code responsible for **messaging**.

```
var queryChainLengthMsg = () => ({'type': MessageType.QUERY_LATEST});
var queryAllMsg = () => ({'type': MessageType.QUERY_ALL});
var responseChainMsg = () => ({
  'type': MessageType.RESPONSE_BLOCKCHAIN, 'data': JSON.stringify(blockchain)
});
var responseLatestMsg = () => ({
  'type': MessageType.RESPONSE_BLOCKCHAIN,
  'data': JSON.stringify([getLatestBlock()])
});
```

4. Next, we will create the function **handleBlockchainResponse**. It will receive the message and show us the appropriate answer depending on our and received blockchain.

```
var handleBlockchainResponse = (message) => {
  var receivedBlocks = JSON.parse(message.data).sort((b1, b2) => (b1.index - b2.index));
  var latestBlockReceived = receivedBlocks[receivedBlocks.length - 1];
  var latestBlockHeld = getLatestBlock();
  if (latestBlockReceived.index > latestBlockHeld.index) {
    console.log('blockchain possibly behind. We got: ' + latestBlockHeld.index +
      ' Peer got: ' + latestBlockReceived.index);
    if (latestBlockHeld.hash === latestBlockReceived.previousHash) {
      console.log("We can append the received block to our chain");
      addBlock(latestBlockReceived);
      broadcast(responseLatestMsg());
    } else if (receivedBlocks.length === 1) {
      console.log("We have to query the chain from our peer");
      broadcast(queryAllMsg());
    } else {
      console.log("Received blockchain is longer than current blockchain");
      replaceChain(receivedBlocks);
    }
  } else {
    console.log('received blockchain is not longer than current blockchain. Do nothing');
  }
};
```

5. Next, we will implement an **initMessageHandler** function. The function will **parse** the messages to JSON format and send them to **console**. It then writes the **message** according to its type.

```
var initMessageHandler = (ws) => {
  ws.on('message', (data) => {
    var message = JSON.parse(data);
    console.log('Received message' + JSON.stringify(message));
    switch (message.type) {
      case MessageType.QUERY_LATEST:
        write(ws, responseLatestMsg());
        break;
      case MessageType.QUERY_ALL:
        write(ws, responseChainMsg());
        break;
      case MessageType.RESPONSE_BLOCKCHAIN:
        handleBlockchainResponse(message);
        break;
    }
  });
};
```

6. The next function is **initErrorHandler**. When something has gone wrong or an unexpected behavior has occurred, it will attempt to close the socket connection and show a message on the console.

```
var initErrorHandler = (ws) => {
  var closeConnection = (ws) => {
    console.log('connection failed to peer: ' + ws.url);
    sockets.splice(sockets.indexOf(ws), 1);
  };
  ws.on('close', () => closeConnection(ws));
  ws.on('error', () => closeConnection(ws));
};
```


7. Implement the write function. It will take the web socket object and a JSON object as parameters. This function **sends** messages back to the client in a stringified JSON format.

```
var write = (ws, message) => ws.send(JSON.stringify(message));
```

8. Now, we are ready to implement the **initConnection** function. The function receives web socket and adds it to sockets array. Then call **initMessageHandler**, **initErrorHandler** and **write** functions.

```
var initConnection = (ws) => {  
  sockets.push(ws);  
  initMessageHandler(ws);  
  initErrorHandler(ws);  
  write(ws, queryChainLengthMsg());  
};
```

9. Now, let's create one more function for **broadcasting** messages to all sockets/peers that we are connected to.

```
var broadcast = (message) => sockets.forEach(socket => write(socket, message));
```

10. With the original intent to connect to a peer, we will implement the function **connectToPeers**. It will try to initialize connections with other peers.

```
var connectToPeers = (newPeers) => {  
  newPeers.forEach((peer) => {  
    var ws = new WebSocket(peer);  
    ws.on('open', () => initConnection(ws));  
    ws.on('error', () => {  
      console.log('connection failed')  
    });  
  });  
};
```

11. Now, let's go back to **initHttpServer** function. We can add the **connectToPeers** functionality so that other connected peers will know when a user adds a new peer.

```

var initHttpServer = () => {
  var app = express();
  app.use(bodyParser.json());

  app.get('/blocks', (req, res) => res.send(JSON.stringify(blockchain)));
  app.post('/mineBlock', (req, res) => {
    var newBlock = generateNextBlock(req.body.data);
    addBlock(newBlock);
    console.log('block added: ' + JSON.stringify(newBlock));
    res.send();
  });
  app.get('/peers', (req, res) => {
    res.send(sockets.map(s => s._socket.remoteAddress + ':' + s._socket.remotePort));
  });
  app.post('/addPeer', (req, res) => {
    connectToPeers([req.body.peer]);
    res.send();
  });
  app.listen(http_port, () => console.log('Listening http on port: ' + http_port));
};

```

12. Now let's add **broadcasting** functionality to our **replaceChain** method and **initHttpServer**.

```

var replaceChain = (newBlocks) => {
  if (isValidChain(newBlocks) && newBlocks.length > blockchain.length) {
    console.log
      ('Received blockchain is valid. Replacing current blockchain with received blockchain');
    blockchain = newBlocks;
    broadcast(responseLatestMsg());
  } else {
    console.log('Received blockchain invalid');
  }
};

```



```

var initHttpServer = () => {
  var app = express();
  app.use(bodyParser.json());

  app.get('/blocks', (req, res) => res.send(JSON.stringify(blockchain)));
  app.post('/mineBlock', (req, res) => {
    var newBlock = generateNextBlock(req.body.data);
    addBlock(newBlock);
    broadcast(responseLatestMsg());
    console.log('block added: ' + JSON.stringify(newBlock));
    res.send();
  });
  app.get('/peers', (req, res) => {
    res.send(sockets.map(s => s._socket.remoteAddress + ':' + s._socket.remotePort));
  });
  app.post('/addPeer', (req, res) => {
    connectToPeers([req.body.peer]);
    res.send();
  });
  app.listen(http_port, () => console.log('Listening http on port: ' + http_port));
};

```

13. Then we must create a function for P2P server **initP2PServer**. It will initialize the server with given **p2p_port**. For communication between the peers, we will use ports 6xxx.

```

var initP2PServer = () => {
  var server = new WebSocket.Server({port: p2p_port});
  server.on('connection', ws => initConnection(ws));
  console.log('listening websocket p2p port on: ' + p2p_port);
};

```

14. Finally, let's connect to peers and initialize the servers.

```

connectToPeers(initialPeers);
initHttpServer();
initP2PServer();

```

5. Setup Connected Nodes and Mine a Block

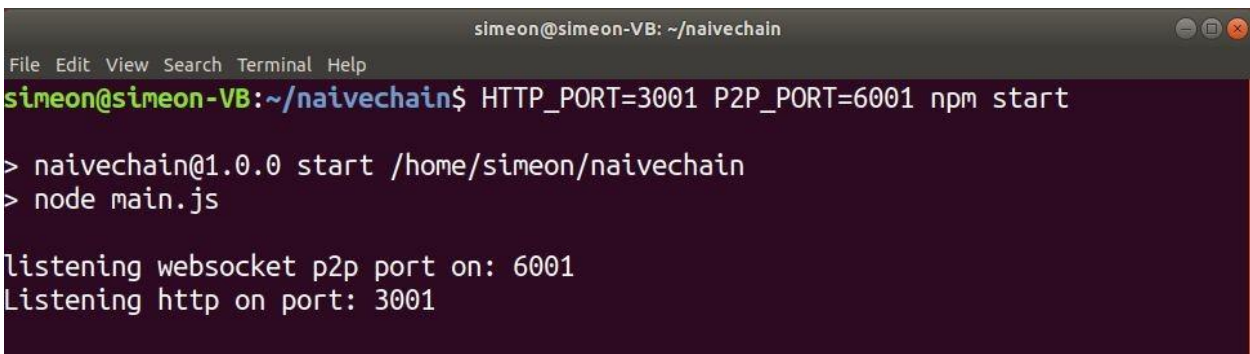
1. First, let's establish the **first node**. The node will listen for **signals from other nodes** by WebSocket interface on port 6001 and will listen for commands via HTTP interface on port 3001. We can see the node's info on address: **localhost:3001**.

Linux/MacOS:

```
HTTP_PORT=3001 P2P_PORT=6001 npm start
```

Windows:

```
set HTTP_PORT=3001 && set P2P_PORT=6001 && npm start
```

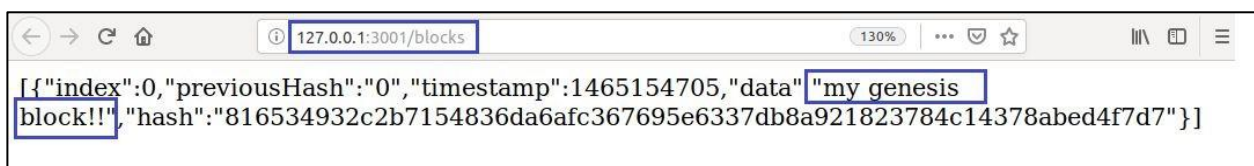


```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB:~/naivechain$ HTTP_PORT=3001 P2P_PORT=6001 npm start
> naivechain@1.0.0 start /home/simeon/naivechain
> node main.js

listening websocket p2p port on: 6001
Listening http on port: 3001
```

2. Open your preferred web browser. Go to the node's address and attach command "**blocks**" in an attempt to receive the **list of all blocks**. On browser write "**localhost:3001/blocks**". Here is the **first block** in blockchain that was **hardcoded**.

```
http://localhost:3001/blocks
```



```
[{"index":0,"previousHash":"0","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}]
```

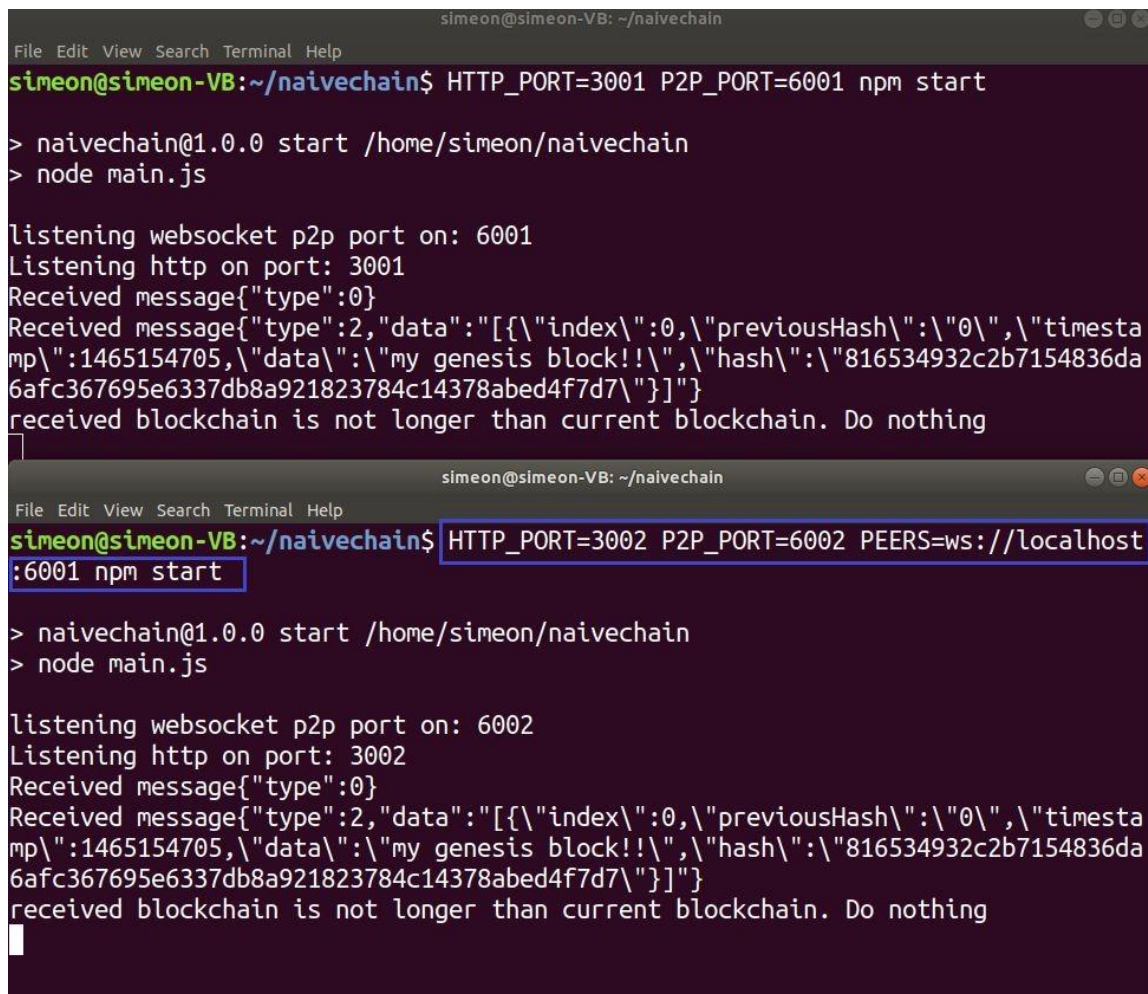
3. Now, open a **second terminal** window and set the **second peer** in the chain. The **second peer** will listen for **signals from other nodes on port 6002** and will listen for commands via HTTP interface on **port 3002**. This node will receive information from the first peer via P2P communication **by port 6001**. We can see the node's info on address: **localhost:3002**. Here on the picture are both nodes terminals.

Linux/MacOS:

```
HTTP_PORT=3002 P2P_PORT=6002 PEERS=ws://localhost:6001 npm start
```

Windows:

```
set HTTP_PORT=3002 && set P2P_PORT=6002 && set  
PEERS=http://localhost:6001 && npm start
```



```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB:~/naivechain$ HTTP_PORT=3001 P2P_PORT=6001 npm start

> naivechain@1.0.0 start /home/simeon/naivechain
> node main.js

listening websocket p2p port on: 6001
Listening http on port: 3001
Received message{"type":0}
Received message{"type":2,"data":[{"index":0,"previousHash":"","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}]}
received blockchain is not longer than current blockchain. Do nothing

simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB:~/naivechain$ HTTP_PORT=3002 P2P_PORT=6002 PEERS=ws://localhost:6001 npm start

> naivechain@1.0.0 start /home/simeon/naivechain
> node main.js

listening websocket p2p port on: 6002
Listening http on port: 3002
Received message{"type":0}
Received message{"type":2,"data":[{"index":0,"previousHash":"","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}]}
received blockchain is not longer than current blockchain. Do nothing
```

4. In the browser, retrieve the **lists of all blocks** of the two nodes. Notice that they have **identical blockchain** because they based off of the same genesis block.



- Now comes the most interesting part. Open Postman and send a POST request to `http://localhost:3001/mineBlock`.

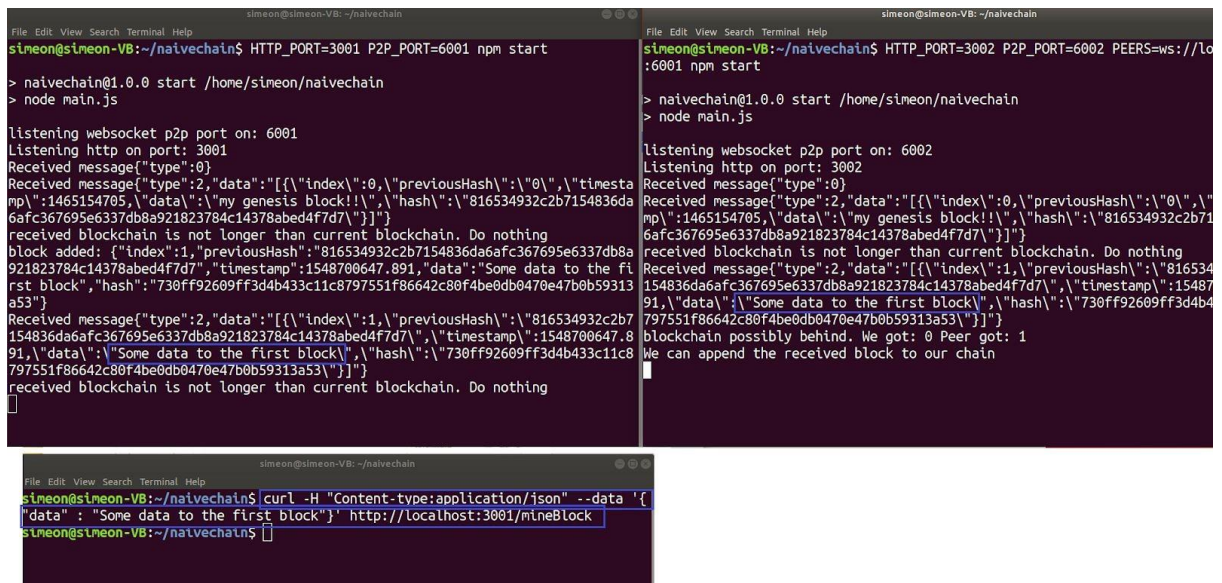
Alternatively, if Postman is not available for your platform, open a new terminal and issue the curl command.

With this POST request, we command the first node to mine a new block. Its index will be the index of the previous block +1. It contains the following data: Data to the first block.

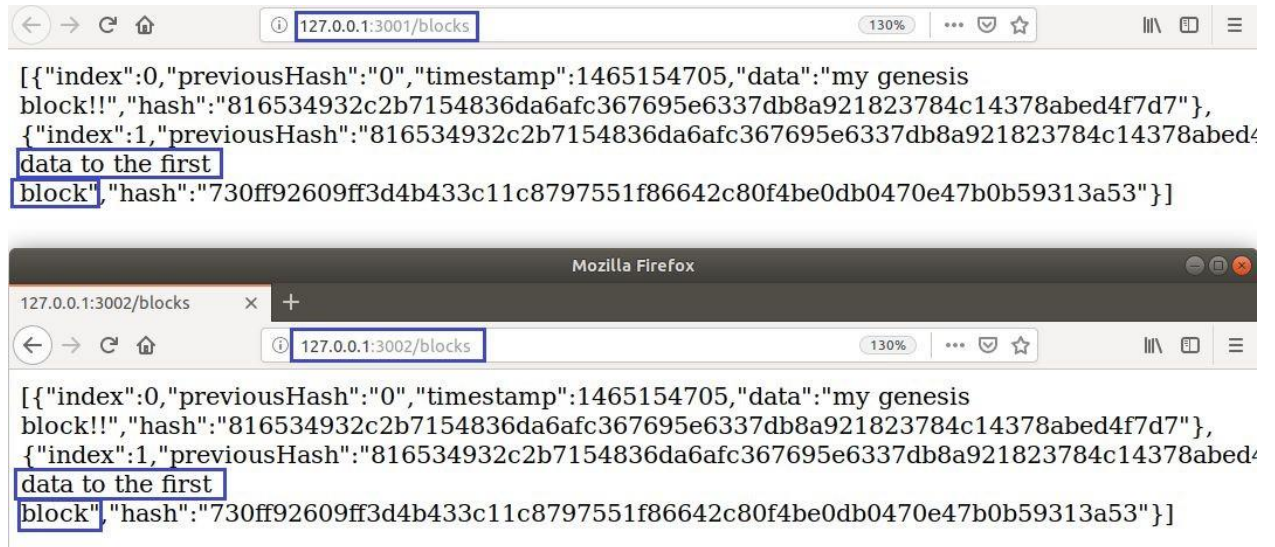
cURL:

```
curl -H "Content-type:application/json" --data '{"data": "Data to the first block"}' http://localhost:3001/mineBlock
```

Postman:



6. Let's examine how blockchain has changed. Retrieve the **lists of all blocks** for both nodes. The **new block** comes just right after the genesis block with **index 1**.



7. Now, let's mine the third block.

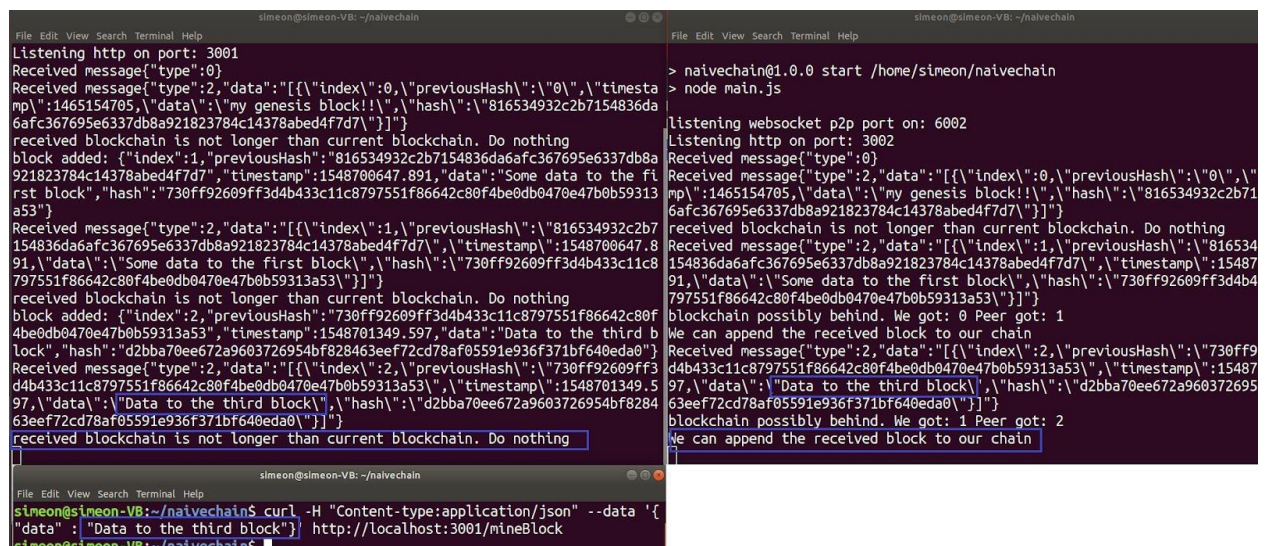
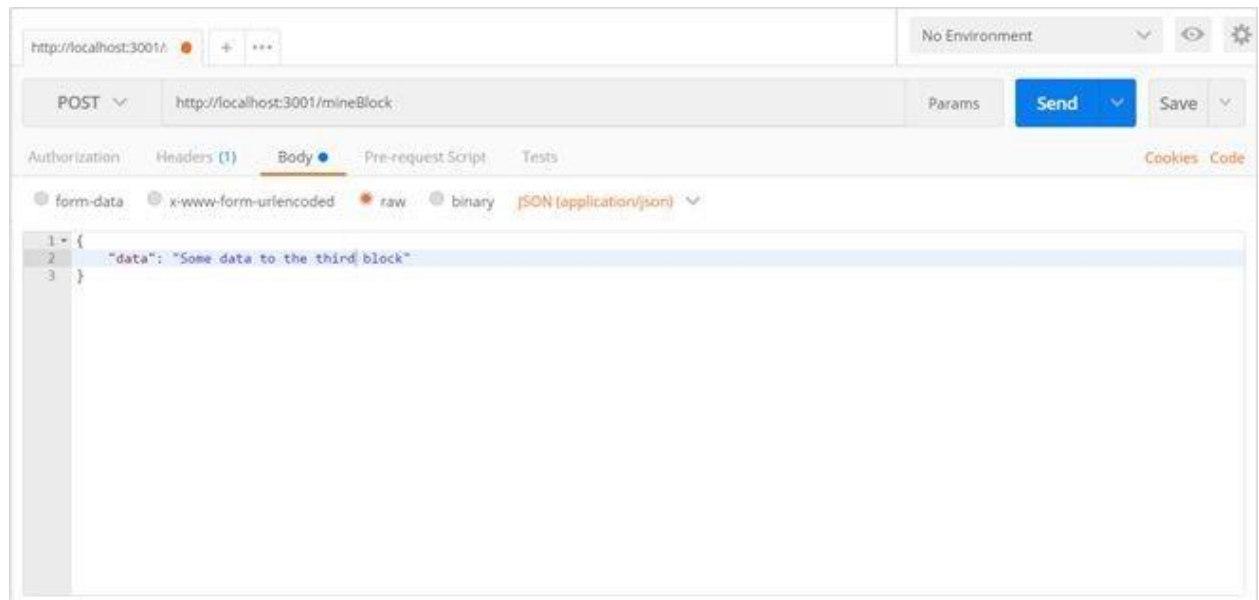
Pay attention since the first node is the emitter of the block so it will report that new blocks sequence is no longer than the existing blockchain with the following message:
Received blockchain is not longer than current blockchain. Do nothing.

But the other node will report that **"We can append the received block to our chain"**.

cURL:

```
curl -H "Content-type:application/json" --data '{"data": "Data to the third block"}' http://localhost:3001/mineBlock
```

Postman:



8. The next task is to create a third node.

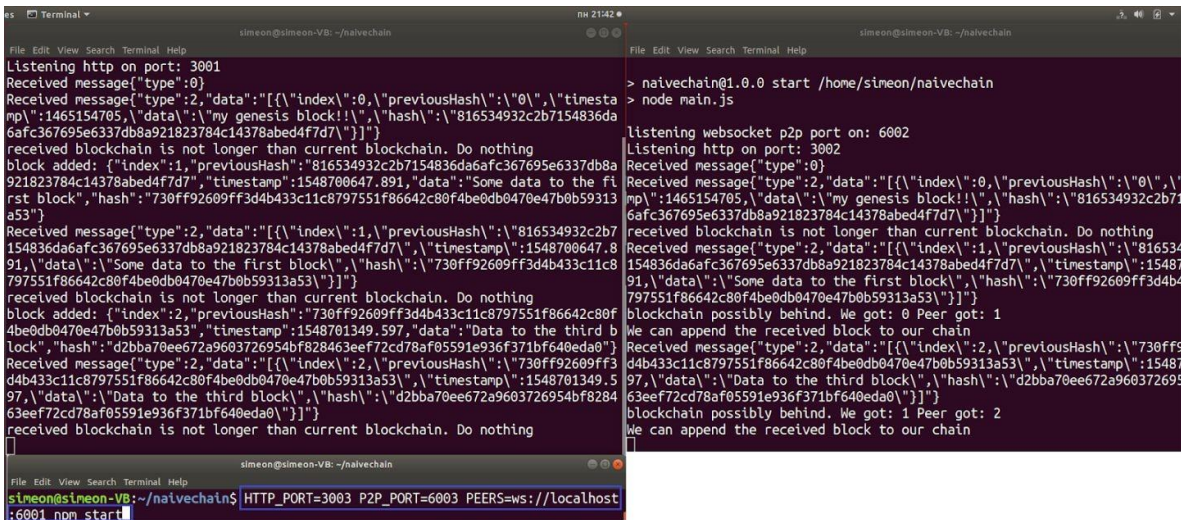
Linux:

```
HTTP_PORT=3003 P2P_PORT=6003 PEERS=ws://localhost:6001 npm start
```

Windows:

```
set HTTP_PORT=3003 && set P2P_PORT=6003 && set PEERS=http://localhost:6001  
&& npm start
```

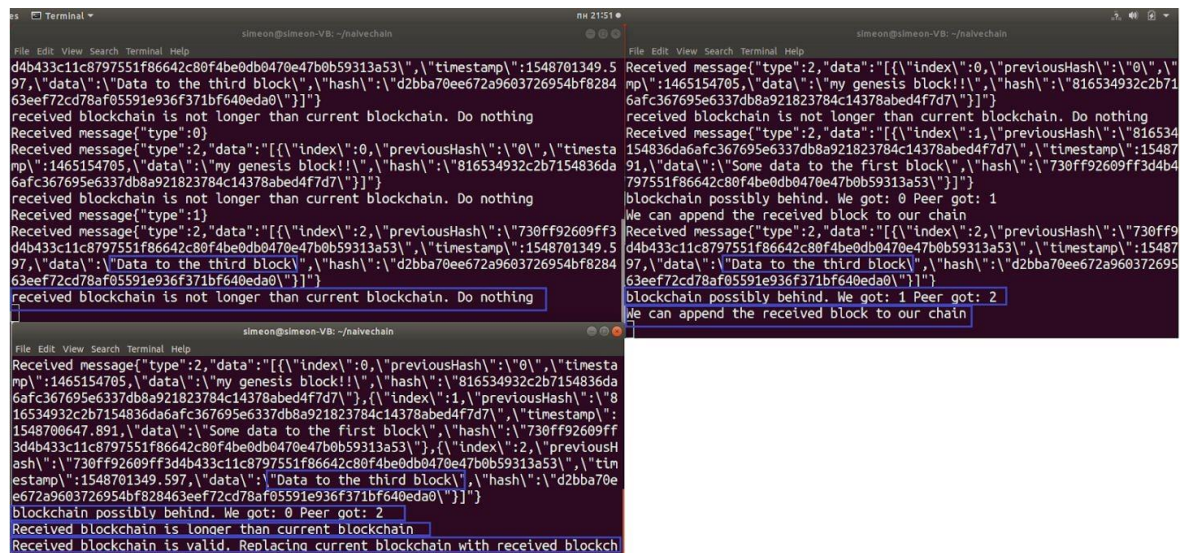
This is the screen before executing the command.



```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
Listening http on port: 3001
Received message{"type":0}
Received message{"type":2,"data":{"index":0,"previousHash":"","timestamp":1465154705,"data":{"my genesis block!!"},"hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}}}
received blockchain is not longer than current blockchain. Do nothing
block added: {"index":1,"previousHash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7","timestamp":1548700647.891,"data":{"Some data to the first block"},"hash":"730ff92609ff3d4b433c11c8797551f86642c80f4be0db0470e47b0b59313a53"}}
Received message{"type":2,"data":{"index":1,"previousHash":"","timestamp":1548700647.891,"data":{"Some data to the first block"},"hash":"730ff92609ff3d4b433c11c8797551f86642c80f4be0db0470e47b0b59313a53"}}}
received blockchain is not longer than current blockchain. Do nothing
block added: {"index":2,"previousHash":"730ff92609ff3d4b433c11c8797551f86642c80f4be0db0470e47b0b59313a53","timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}
Received message{"type":2,"data":{"index":2,"previousHash":"","timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}}
received blockchain is not longer than current blockchain. Do nothing

simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
> naivechain@1.0.0 start /home/simeon/naivechain
> node main.js
Listening websocket p2p port on: 6002
Listening http on port: 3002
Received message{"type":0}
Received message{"type":2,"data":{"index":0,"previousHash":"","timestamp":1465154705,"data":{"my genesis block!!"},"hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}}}
received blockchain is not longer than current blockchain. Do nothing
Received message{"type":2,"data":{"index":1,"previousHash":"","timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}}
blockchain possibly behind. We got: 0 Peer got: 1
We can append the received block to our chain
Received message{"type":2,"data":{"index":2,"previousHash":"","timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}}
blockchain possibly behind. We got: 1 Peer got: 2
We can append the received block to our chain
```

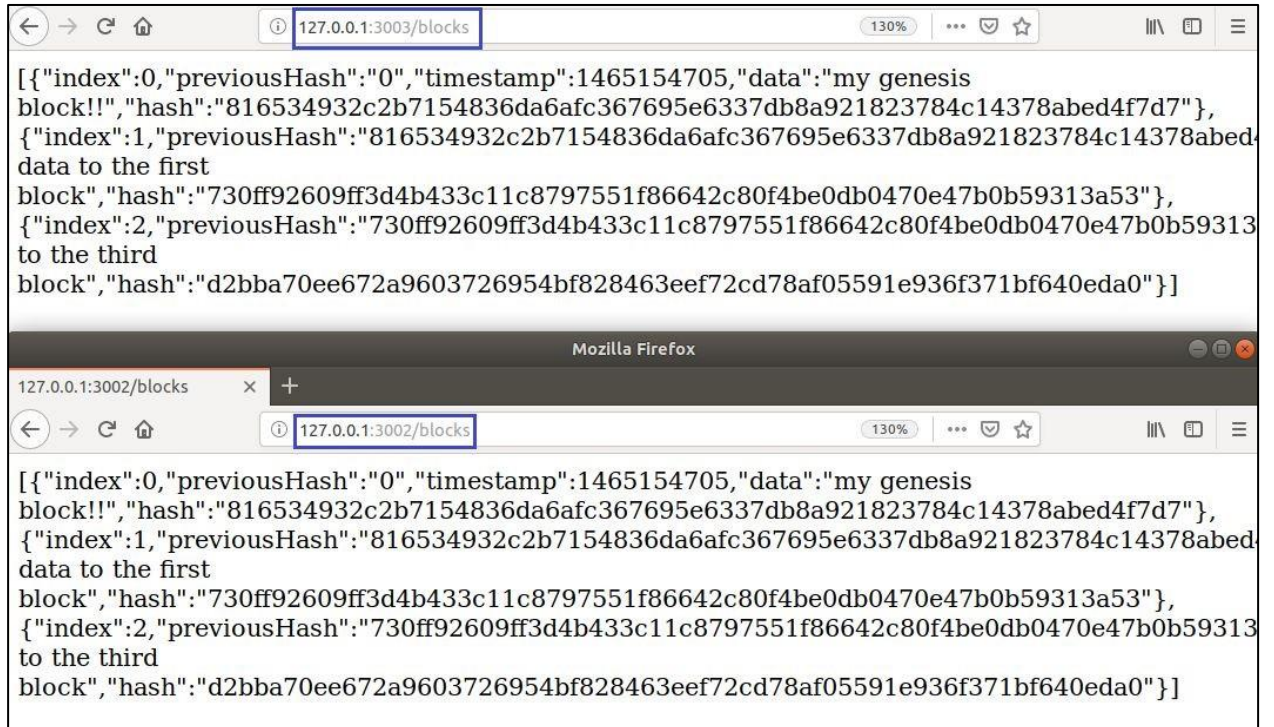
And this is the screen after enter key was pressed. The **third** node was created and is now participating in the blockchain.



```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
d4b433c11c8797551f86642c80f4be0db0470e47b0b59313a53"},"timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}}
received blockchain is not longer than current blockchain. Do nothing
Received message{"type":0}
Received message{"type":2,"data":{"index":0,"previousHash":"","timestamp":1465154705,"data":{"my genesis block!!"},"hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}}}
received blockchain is not longer than current blockchain. Do nothing
Received message{"type":1}
Received message{"type":2,"data":{"index":2,"previousHash":"","timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}}
received blockchain is not longer than current blockchain. Do nothing

simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
Received message{"type":2,"data":{"index":0,"previousHash":"","timestamp":1465154705,"data":{"my genesis block!!"},"hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}}}
received blockchain is not longer than current blockchain. Do nothing
Received message{"type":2,"data":{"index":1,"previousHash":"","timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}}
blockchain possibly behind. We got: 0 Peer got: 1
We can append the received block to our chain
Received message{"type":2,"data":{"index":2,"previousHash":"","timestamp":1548701349.597,"data":{"Data to the third block"},"hash":"d2bba70ee672a9603726954bf828463eef72cd78af05591e936f371bf640eda0"}}}
blockchain possibly behind. We got: 1 Peer got: 2
We can append the received block to our chain
```


9. Open the **third node address** in the browser and compare it with one of the old nodes.
The third node would now have synced and have the **same blocks**.



6. Implement Proof-of-Work Mining (optional)

Now, if you want to continue, you can improve your block **implement proof-of-work mining** in the program.

We will implement the **“/mineBlock”** REST endpoint to calculate a block hash with a **few leading zeroes** (the proof of work), instead of just the block hash.

1. **Open** the file **“main.js”** in your preferred text editor or IDE. Modify the code and **implement proof-of-work**.

First, we should create variable to hold the network **difficulty**. This is the required **number of zeroes** at the beginning of the next block **hash**.

```
'use strict';
var CryptoJS = require("crypto-js");
var express = require("express");
var bodyParser = require('body-parser');
var WebSocket = require("ws");
var difficulty = 4;
```

2. Next, we should add the **“nonce”** property and the **“difficulty”** property in the **Block** constructor. The nonce is the number which we will **increase** in an attempt to find the appropriate proof-of-work hash and **mine the next block**. The difficulty is important because without its value nobody will be able to prove that the found hash is valid or not.

```
class Block {
  constructor(index, previousHash, timestamp, data, hash, difficulty, nonce) {
    this.index = index;
    this.previousHash = previousHash.toString();
    this.timestamp = timestamp;
    this.data = data;
    this.hash = hash.toString();
    this.difficulty = difficulty;
    this.nonce = nonce;
  }
}
```

3. We will add **nonce** and **difficulty** in every block our program creates. In this case, we can modify the **function**, which **creates the hardcoded genesis block**: add **nonce** and **difficulty** with 0 values.

```
var getGenesisBlock = () => {
  return new Block(0, "0", 1465154705, "my genesis block!!",
    "816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7", 0, 0);
};
```

4. Then we should modify functions **calculateHashForBlock** and **calculateHash**:

```
var calculateHash = (index, previousHash, timestamp, data, nonce) => {
  return CryptoJS.SHA256(index + previousHash + timestamp + data + nonce).toString();
};

var calculateHashForBlock = (block) => {
  return calculateHash(block.index, block.previousHash, block.timestamp, block.data, block.nonce);
};
```

5. Now, let's create the function **mineBlock**. It receives the **block info** and goes through a while loop that must satisfy a given hash condition.
If the generated **hash** does not satisfy the **difficulty** requirements at each loop cycle, the **nonce** is incremented, otherwise, the loop terminates and a new **block** is created.

```
var http_port = process.env.HTTP_PORT || 3001;
var sockets = [];

var mineBlock = (blockData) => {
  var previousBlock = getLatestBlock();
  var nextIndex = previousBlock.index + 1;
  var nonce = 0;
  var nextTimestamp = new Date().getTime() / 1000;
  var nextHash = calculateHash(nextIndex, previousBlock.hash, nextTimestamp, blockData, nonce);
  while (nextHash.substring(0, difficulty) !== Array(difficulty + 1).join("0")){
    nonce++;
    nextTimestamp = new Date().getTime() / 1000;
    nextHash = calculateHash(nextIndex, previousBlock.hash, nextTimestamp, blockData, nonce)

    console.log("\nindex\":" + nextIndex + ",\npreviousHash\":"+previousBlock.hash+
      "\ntimestamp\":"+nextTimestamp+",\ndata\":"+blockData+
      ",\x1b[33mhash: " + nextHash + " \x1b[0m," + "\ndifficulty\":"+difficulty+
      " \x1b[33mnonce: " + nonce + " \x1b[0m ");
  }

  return new Block(nextIndex, previousBlock.hash, nextTimestamp, blockData, nextHash, difficulty, nonce);
};
```

In real blockchains, the **Proof-Of-Work** is a piece of data which is **difficult** (costly, time-consuming) to produce but easy for others to **verify** and which **satisfies** certain requirements. Producing a Proof-Of-Work is a **random process** with **low probability** so that a lot of trial and error is required on average before a valid proof of work is generated. In order for a block to be **accepted** by network participants, miners must

complete the Proof-Of-Work which covers all of the data in the block. Due to the very low probability of successful generation, this makes it **unpredictable** to know which mining computer in the network will be able to generate the next block.

6. Before, without the Proof-of-Work mechanism when executing the command “**mineBlock**”, we were just creating a new block immediately.

Now, let’s use the new **function “mineBlock”** responsible for mining integrated with Proof-of-Work.

```
var initHttpServer = () => {
  var app = express();
  app.use(bodyParser.json());

  app.get('/blocks', (req, res) => res.send(JSON.stringify(blockchain)));
  app.post('/mineBlock', (req, res) => {
    var newBlock = mineBlock(req.body.data);
    //var newBlock = generateNextBlock(req.body.data);
    addBlock(newBlock);
    broadcast(responseLatestMsg());
    console.log('block added: ' + JSON.stringify(newBlock));
    res.send();
  });
  app.get('/peers', (req, res) => {
    res.send(sockets.map(s => s._socket.remoteAddress + ':' + s._socket.remotePort));
  });
  app.post('/addPeer', (req, res) => {
    connectToPeers([req.body.peer]);
    res.send();
  });
  app.listen(http_port, () => console.log('Listening http on port: ' + http_port));
};
```

7. Run Two “Naivechain” Nodes and Run PoW Mining

The code is ready. Now, let’s test the modified program.

1. Run two Naivechain nodes at the following ports:

- REST: 3001, P2P:6001
- REST: 3002, P2P:6002


```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB:~/naivechain$ HTTP_PORT=3001 P2P_PORT=6001 npm start

> naivechain@1.0.0 start /home/simeon/naivechain
> node main.js

listening websocket p2p port on: 6001
Listening http on port: 3001
Received message{"type":0}
Received message{"type":2,"data":[{"index":0,"previousHash":"","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7","difficulty":0,"nonce":0}]}
received blockchain is not longer than current blockchain. Do nothing

simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB:~/naivechain$ HTTP_PORT=3002 P2P_PORT=6002 PEERS=ws://localhost:6001 npm start

> naivechain@1.0.0 start /home/simeon/naivechain
> node main.js

listening websocket p2p port on: 6002
Listening http on port: 3002
Received message{"type":0}
Received message{"type":2,"data":[{"index":0,"previousHash":"","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7","difficulty":0,"nonce":0}]}
received blockchain is not longer than current blockchain. Do nothing
```

2. Open the two nodes' blockchains in the browser. In the browser URL bar, write "**localhost:3001/blocks**" and "**localhost:3002/blocks**". Here, the only block is the hardcoded **genesis** block with **index 0**.

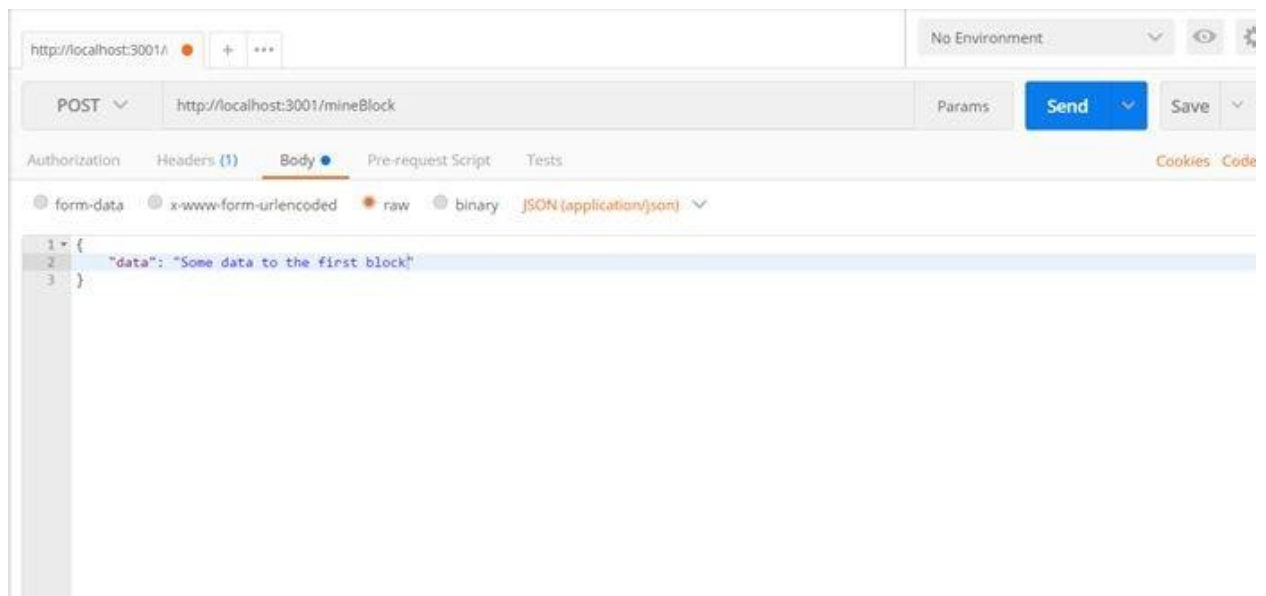


- Now, it's time to mine. Open a new console and issue the following POST request. With this, we command the **first node to mine a new block**:

cURL:

```
curl -H "Content-type:application/json" --data '{"data": "Some data to the first block"}' http://localhost:3001/mineBlock
```

Postman:



- Look at the **first node's console**. The miner starts mining and **increments the nonce** searching for the **hash that satisfies the difficulty** requirements.

```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB:~/naivechain$ curl -H "Content-type:application/json" --data '{"data": "MyInfo 1"}' http://localhost:3001/mineBlock
simeon@simeon-VB:~/naivechain$
```

```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
abed4f7d7"timestamp":1548757409.872,"data":MyInfo 1,hash: 32cc1d70f1c4bc6cc97bba13035609cde6c8203a22abf2a69284a75c0364b9ba , "difficulty":4 nonce: 21052
"index":1,"previousHash":816534932c2b7154836da6afc367695e6337db8a921823784c14378
abed4f7d7"timestamp":1548757409.872,"data":MyInfo 1,hash: 8bc1e4aa5f0d649217dbf709081f88e2589220d9548d30fb073b3043e77dbae6 , "difficulty":4 nonce: 21053
"index":1,"previousHash":816534932c2b7154836da6afc367695e6337db8a921823784c14378
abed4f7d7"timestamp":1548757409.872,"data":MyInfo 1,hash: 4e6f6973cc0e1a400fb94daa943025d008a96dba0f07c3d8a45de1276d4bd051 , "difficulty":4 nonce: 21054
"index":1,"previousHash":816534932c2b7154836da6afc367695e6337db8a921823784c14378
abed4f7d7"timestamp":1548757409.872,"data":MyInfo 1,hash: 10cd48a3fc8936b675b3b58eaae57a26caf96d639213fc640594f15b02055701 , "difficulty":4 nonce: 21055
"index":1,"previousHash":816534932c2b7154836da6afc367695e6337db8a921823784c14378
abed4f7d7"timestamp":1548757409.872,"data":MyInfo 1,hash: 0000202aa01af3f69262a697865bb56a7d330af106b0c44028ac2f673805d4d8 , "difficulty":4 nonce: 21056
block added: {"index":1,"previousHash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"."timestamp":1548757409.872."data":"MyInfo 1","hash":"0000202aa01af3f69262a697865bb56a7d330af106b0c44028ac2f673805d4d8","difficulty":4,"nonce":21056}
Received message{"type":2,"data":[{"index":1,"previousHash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7","timestamp":1548757409.872,"data":"MyInfo 1","hash":"0000202aa01af3f69262a697865bb56a7d330af106b0c44028ac2f673805d4d8","difficulty":4,"nonce":21056}]}
received blockchain is not longer than current blockchain. Do nothing
```


Note the following results:

- Block mining **takes some time** (a few seconds). This is due to the proof-of-work algorithm.
- The last block hash has **several leading zeroes** depending on **difficulty**. This is the proof-of-work result.

5. The second node also accepted a new block in the blockchain.

```
simeon@simeon-VB: ~/naivechain
File Edit View Search Terminal Help
simeon@simeon-VB:~/naivechain$ HTTP_PORT=3002 P2P_PORT=6002 PEERS=ws://localhost:6001 npm start

> naivechain@1.0.0 start /home/simeon/naivechain
> node main.js

listening websocket p2p port on: 6002
Listening http on port: 3002
Received message{"type":0}
Received message{"type":2,"data":[{"index":0,"previousHash":"","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7","difficulty":0,"nonce":0}]}
received blockchain is not longer than current blockchain. Do nothing
Received message{"type":2,"data":[{"index":1,"previousHash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7","timestamp":1548757409.872,"data":"MyInfo 1","hash":"0000202aa01af3f69262a697865bb56a7d330af106b0c44028ac2f673805d4d8","difficulty":4,"nonce":21056}]}
blockchain possibly behind. We got: 0 Peer got: 1
We can append the received block to our chain
```


- Look at the nodes in the browser. Here is the new block with **several leading zeroes**, **difficulty** and **nonce**.

Using this info, the hash that was found can be easily verified by other miners.



What to Submit?

Create a **zip file** (e.g. your-name-simple-miner-extended-exercise.zip) holding the screenshots of the following:

- Your browser requests/responses
 - GET
 - POST
- Your terminal
 - Peer syncing
 - Mining

Submit your **ZIP** file as **homework** at the course platform.