BLOCK CHAIN

Chapter 1: Introduction

1.1 Blockchain Technology:

Blockchain technology is the foundation behind cryptocurrencies like Bitcoin and serves various purposes beyond just digital currencies. Here's a breakdown of the key points:

1. **Proof of Work (PoW):** PoW is a consensus mechanism used to secure the blockchain network. Miners solve complex mathematical puzzles to add new blocks to the blockchain. This process requires computational work and ensures the integrity of the network.
2. **New Blocks through Mining:** Mining is the process by which new transactions are grouped together into blocks and added to the blockchain. Miners compete to solve PoW puzzles, and the first one to solve it gets the right to create a new block.
3. **Immutable**: Once data is recorded on the blockchain, it cannot be altered or deleted. This immutability is a core feature that ensures the integrity of transaction history
4. **Validate the Blockchain**: The blockchain network relies on a network of nodes (computers) to validate and verify transactions. This decentralized validation process enhances security.
5. **Retrieve Data:** Users can access data stored on the blockchain, making it a transparent and auditable ledger.

1.2 Features of Block chain:

1. **Standard API:** This provides a way for external applications or services to interact with the blockchain, allowing for various functionalities like sending transactions or querying data.

2. **Upgraded API for Decentralized Network:** This feature suggests that the blockchain can be integrated into a larger, decentralized network, potentially expanding its use cases and reach.

3. **Consensus Algorithm:** The consensus algorithm is a crucial component of blockchain technology. It ensures that all participants agree on the state of the blockchain. Ensuring data integrity is its primary function.

4. **User Interface:** A user-friendly interface allows individuals to interact with the blockchain without needing technical expertise. It can be a web application or software that provides a graphical representation of blockchain data.

1.3 Definition

What is a blockchain?

It is an **immutable distributed Ledger**

Now let us understand each term in the definition

1. Ledger (Financial Accounts): A ledger is a record-keeping system that tracks transactions, financial activities, or other types of data
    - example Roy gives $30 to Joe , Ken gives $70 to Ria ……

2. The key characteristic here is that once these entries are recorded on the blockchain, they cannot be changed (immutable).

3. Distributed vs. Centralized:
    - In a distributed blockchain, data is stored and validated across a network of nodes, and no single entity controls it. This decentralized nature enhances security and transparency.
    - In contrast, a centralized system, like the examples of Joe and Ria, is controlled by a single entity. Any changes or transactions are subject to the

control and decisions of that central authority, which can be less transparent and potentially less secure.

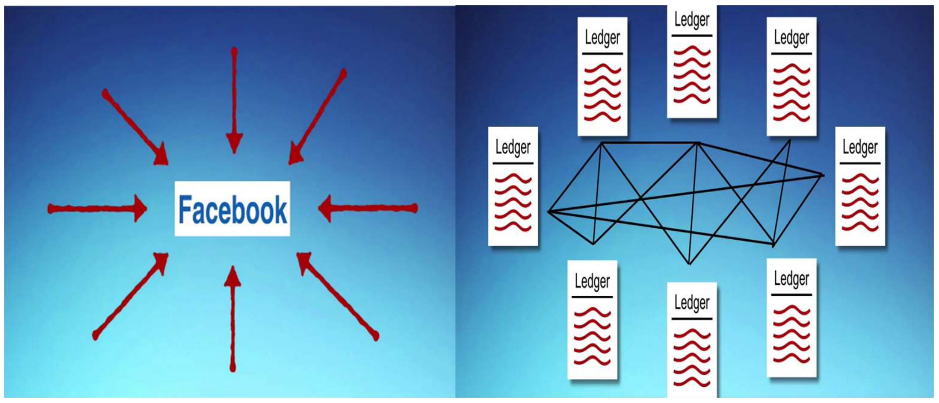


Fig 1.1 Shows a comparison of Centralised and Decentralized System

Overall, blockchain technology provides a decentralized, secure, and transparent way to record and manage data, with various applications extending beyond just financial transactions.

Chapter 2: Creation of Blockchain

2.1 Prerequisite

2.1.1 Constructor Function

A constructor function in JavaScript is like a blueprint for creating and initializing objects. It defines how objects of a certain type should be structured and what properties and methods they should have. When you use the new keyword with a constructor function, it performs several essential tasks:

Object Creation: It creates a new object based on the blueprint defined by the constructor function.

Property Initialization: It initializes the object's properties with the values passed as arguments when creating the object.

this Keyword: Inside the constructor function, this keyword refers to the newly created object. It allows you to set properties and perform other tasks specific to that instance.

Here's a concise example:

```
function User(firstName, lastName, age) {
 this.firstName = firstName;
 this.lastName = lastName;
this.age=age;
}
var person1 = new User("John","Smith",30 );
```

In this code, the Person constructor function defines how Person objects should be structured. When you create a new person1 object using new Person(...), the constructor initializes its name and age properties based on the provided arguments. This pattern is fundamental for creating and configuring objects in JavaScript.

2.1.2 Prototype Function

The prototype function is defined on the constructor function's prototype property. When a new object is created using the constructor function, it inherits the properties and methods from the constructor function's prototype.

User prototype is used to define a method called getEmailAddress. This method is used to generate the email address of a user based on their first name, last name, and email domain.

User.prototype.emailDomain=”example.com”

User.prototype.getEmailAddress = function() {

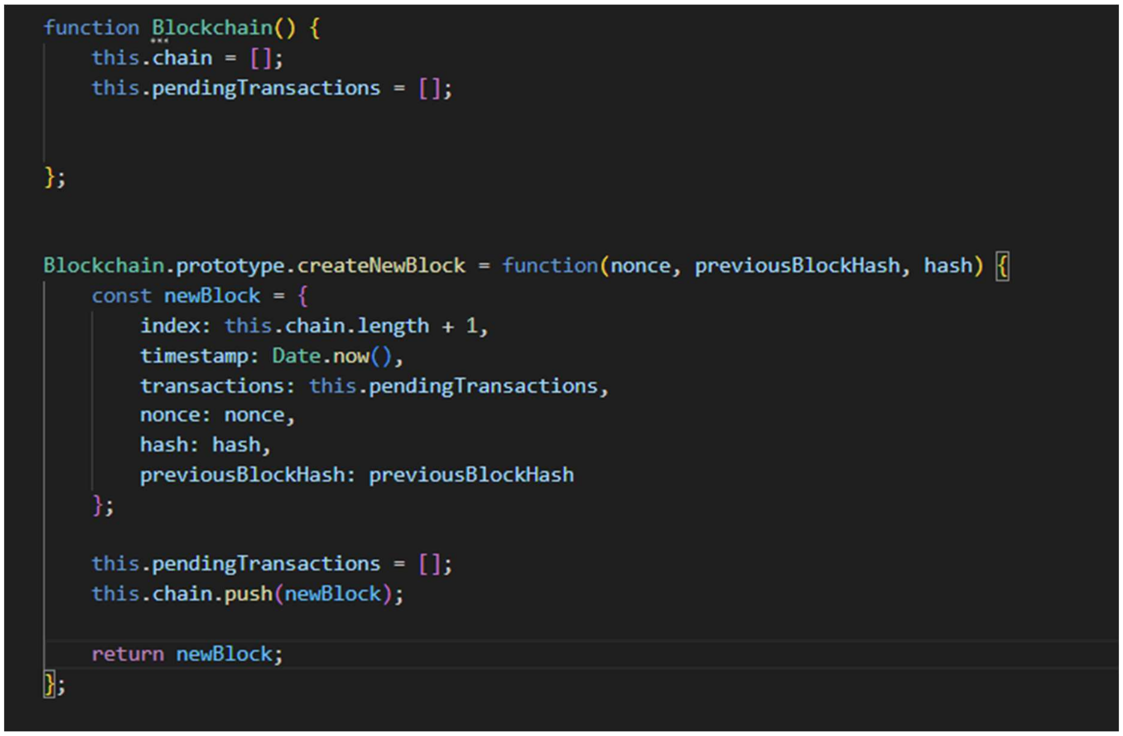
 return this.firstName + '.' + this.lastName + '@' + this.emailDomain;

}

The getEmailAddress method is called on the person1 object to generate the user's email address.

console.log(person1.getEmailAddress()); // Output: jhonsmith@example.com

2.2 Creating a new BlockMethod



```
function Blockchain() {
    this.chain = [];
    this.pendingTransactions = [];


};


Blockchain.prototype.createNewBlock = function(nonce, previousBlockHash, hash) {
    const newBlock = {
        index: this.chain.length + 1,
        timestamp: Date.now(),
        transactions: this.pendingTransactions,
        nonce: nonce,
        hash: hash,
        previousBlockHash: previousBlockHash
    };

    this.pendingTransactions = [];
    this.chain.push(newBlock);

    return newBlock;
};
```

1. **`Blockchain` Function Builder:**

- This code starts by defining the constructor function `Blockchain`. JavaScript uses constructor functions to create objects with specific properties and methods.

- There are three main types in the constructor.

  - `this.chain`: the empty array that will eventually hold the blockchain blocks.

  - `this.pendingTransactions`: An empty array to store transactions not yet added to the block.

2. **The `createNewBlock` method:**

- This method is defined by `Blockchain.prototype` in addition to all instances of the `Blockchain` class.

- Accepts three parameters as input:

  - `nonce`: number representing the proof of work value (PoW).

  - `previousBlockHash`: A string representing the hash of the previous block in the blockchain.

  - `hash`: A string representing the hash of the current block.

- Method: Inside:

  - Created as a new product with many attributes:

    - `index`: represents the position of a block in the blockchain. This is set one longer than the length of the current blockchain, making it unique.

    - `timestamp`: Specifies when the block was created with `Date.now()`.

    - `transactions`: Holds a schedule of pending transactions, effectively adding these transactions to the block.

    - `nonce`: represents the proof of value of the job, which is usually calculated through the mining process.

    - `hash`: Represents the hash value of the current block.

    - `previousBlockHash`: store the hash value of the previous block in the blockchain.

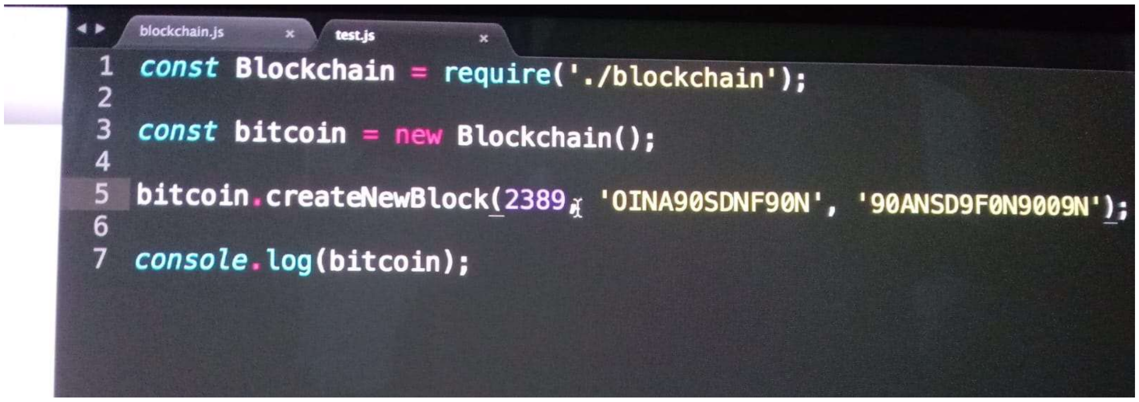
- The `this.pendingTransactions` structure is reset to an empty array because this transaction is now added to the newly created volume.

- Pushing the other half into the `this.chain` structure, effectively adding it to the blockchain.

- Finally, the newly created block is returned.

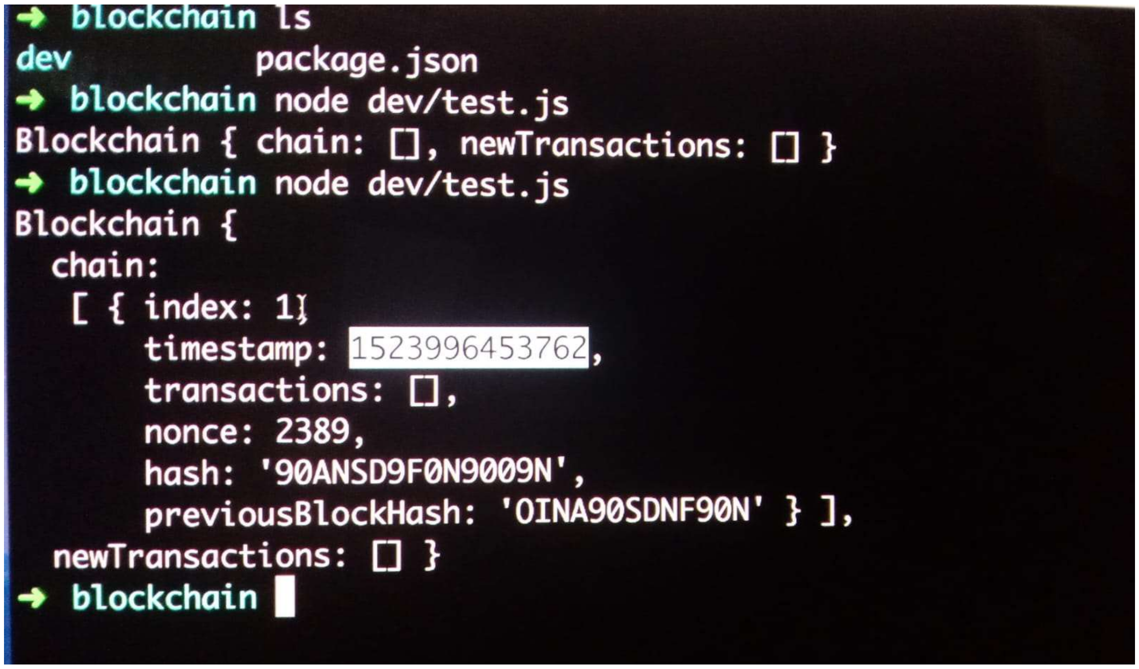
This code provide the basic framework for creating a blockchain, adding new blocks, and adding transactions to those blocks. However, it does not have the options to use mining (calculate the value of `nonce` using PoW) and transactions will be added to the `pendingTransactions` array. Throughout the blockchain implementation, you will find other ways to manage these functions and ensure the security and focus of the blockchain network.

Testing the create block chain method



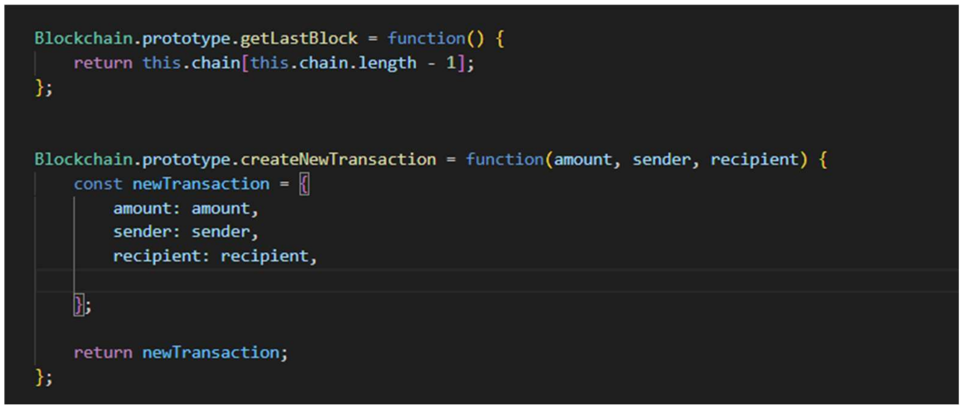
```
const Blockchain = require('./blockchain');

const bitcoin = new Blockchain();

bitcoin.createNewBlock(2389, 'OINA90SDNF90N', '90ANSD9F0N9009N');

console.log(bitcoin);
```

Output



```
➜ blockchain ls
dev          package.json
➜ blockchain node dev/test.js
Blockchain { chain: [], newTransactions: [] }
➜ blockchain node dev/test.js
Blockchain {
  chain:
   [ { index: 1,
       timestamp: 1523996453762,
       transactions: [],
       nonce: 2389,
       hash: '90ANSD9F0N9009N',
       previousBlockHash: 'OINA90SDNF90N' } ],
  newTransactions: [] }
➜ blockchain
```

2.2 Creating a new BlockMethod



```
Blockchain.prototype.getLastBlock = function() {
    return this.chain[this.chain.length - 1];
};


Blockchain.prototype.createNewTransaction = function(amount, sender, recipient) {
    const newTransaction = {
        amount: amount,
        sender: sender,
        recipient: recipient,

    };

    return newTransaction;
};
```

1. **The `getLastBlock` method:**

- This method is defined by `Blockchain.prototype` in addition to all instances of the `Blockchain` class.

- Without any parameters.

- Method: Inside:

- Going to `this.chain[this.chain.length - 1]` retrieves the last block in the blockchain. This expression computes the index of the last row in the `this.chain` array.

- The last block is then returned. This method basically provides an easy way to get the most recent block in the blockchain.

2. **The `createNewTransaction` method:**

- This method is also defined by `Blockchain.prototype`.

- Accepts three parameters as input:

  - `Amount`: An account value representing the amount of any cryptocurrency or asset transferred.

  - `sender`: a string representing the address or identifier of the sender.

  - `recipient`: a string representing the address or identifier of the recipient.

- Method: Inside:

  - Another real-world product with three properties:

  - `Amount`: represents the transfer amount.

  - `sender`: stands for the address of the sender.

  - `recipient`: stands for the address of the recipient.

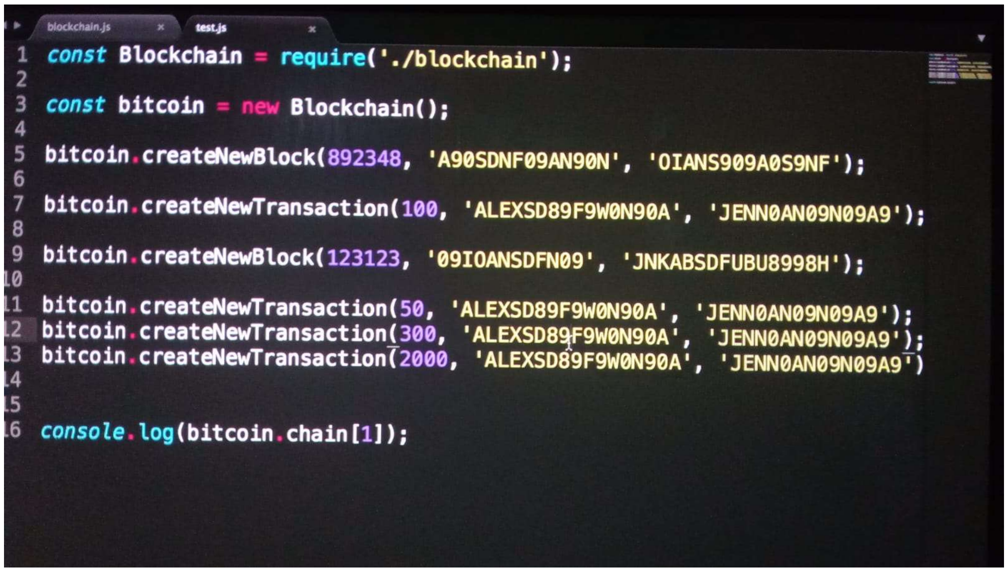
  - The new version is then restored.

These two methods are used to interact with the blockchain:

- `getLastBlock` is useful for various purposes, such as displaying information about the last block or verifying the integrity of the blockchain by checking the hash of that last block

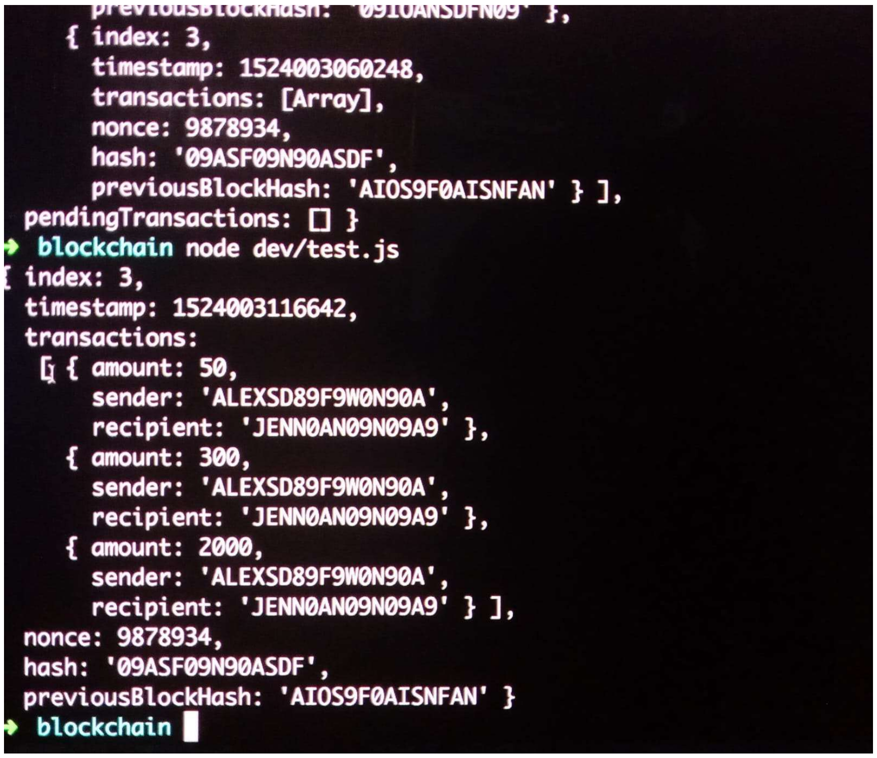
- `createNewTransaction` is a simple method for creating a new transaction. However, it is important to note that in all blockchain applications, this method is often used in conjunction with other methods to implement transaction validation and add to the `pendingTransactions` system, and vice versa , mining systems are developed in addition to other blockchains. This implementation is simplified and lacks those necessary steps for a fully functional blockchain.

Testing the create Transaction Method



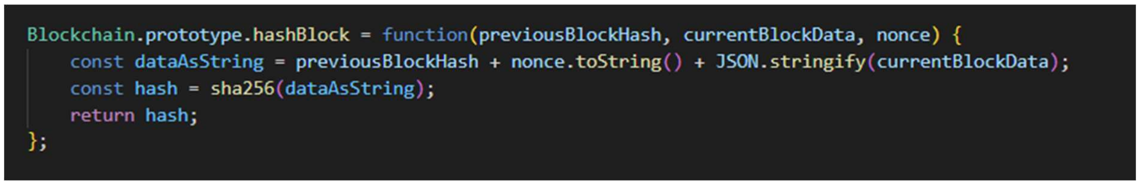
```
const Blockchain = require('./blockchain');

const bitcoin = new Blockchain();

bitcoin.createNewBlock(892348, 'A90SDNF09AN90N', 'OIANS909A0S9NF');

bitcoin.createNewTransaction(100, 'ALEXSD89F9W0N90A', 'JENN0AN09N09A9');

bitcoin.createNewBlock(123123, '09IOANSDFN09', 'JNKABSDFUBU8998H');

bitcoin.createNewTransaction(50, 'ALEXSD89F9W0N90A', 'JENN0AN09N09A9');
bitcoin.createNewTransaction(300, 'ALEXSD89F9W0N90A', 'JENN0AN09N09A9');
bitcoin.createNewTransaction(2000, 'ALEXSD89F9W0N90A', 'JENN0AN09N09A9')


console.log(bitcoin.chain[1]);
```

Output



```
    previousBlockHash: '09IOANSDFN09' },
  { index: 3,
    timestamp: 1524003060248,
    transactions: [Array],
    nonce: 9878934,
    hash: '09ASF09N90ASDF',
    previousBlockHash: 'AIOS9F0AISNFAN' } ],
 pendingTransactions: [] }
➜ blockchain node dev/test.js
{ index: 3,
 timestamp: 1524003116642,
 transactions:
  [ { amount: 50,
      sender: 'ALEXSD89F9W0N90A',
      recipient: 'JENN0AN09N09A9' },
    { amount: 300,
      sender: 'ALEXSD89F9W0N90A',
      recipient: 'JENN0AN09N09A9' },
    { amount: 2000,
      sender: 'ALEXSD89F9W0N90A',
      recipient: 'JENN0AN09N09A9' } ],
 nonce: 9878934,
 hash: '09ASF09N90ASDF',
 previousBlockHash: 'AIOS9F0AISNFAN' }
➜ blockchain
```

2.3 Hashing



```
Blockchain.prototype.hashBlock = function(previousBlockHash, currentBlockData, nonce) {
    const dataAsString = previousBlockHash + nonce.toString() + JSON.stringify(currentBlockData);
    const hash = sha256(dataAsString);
    return hash;
};
```

The hashBlock method you've provided is used to calculate the hash of a block in the blockchain. This hash is typically used as part of the proof of work (PoW) process to validate and add new blocks to the blockchain.

1. Concatenation:

- The method first concatenates these three values together into a single string:

  - `previousBlockHash`: The hash of the previous block.

  - `nonce.toString()`: The `nonce` value converted to a string.

  - `JSON.stringify(currentBlockData)`: The JSON string representation of the current block's data.

- Concatenating these values ensures that any change in the previous block's hash, nonce, or block data will result in a different hash for the current block.

2. Hash Calculation:

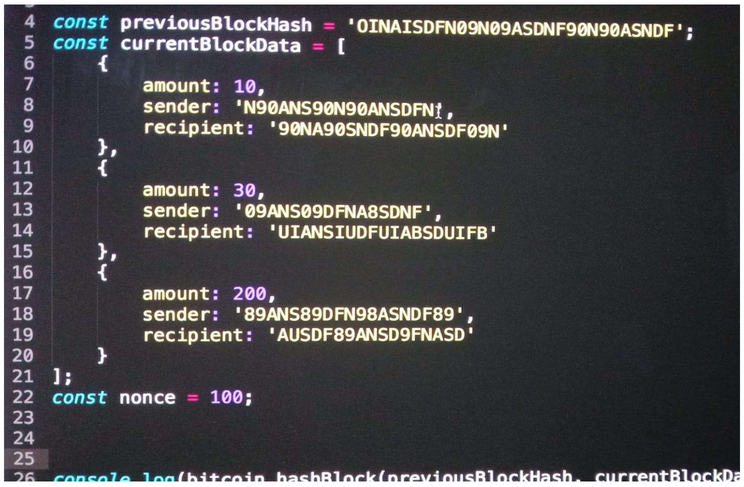
- After concatenating the values, the method calculates the SHA-256 hash of the resulting string using a function called `sha256`. This step produces a unique hash value based on the input data.

3. Return Value:

- Finally, the method returns the calculated hash, which represents the unique identifier for the current block in the blockchain.

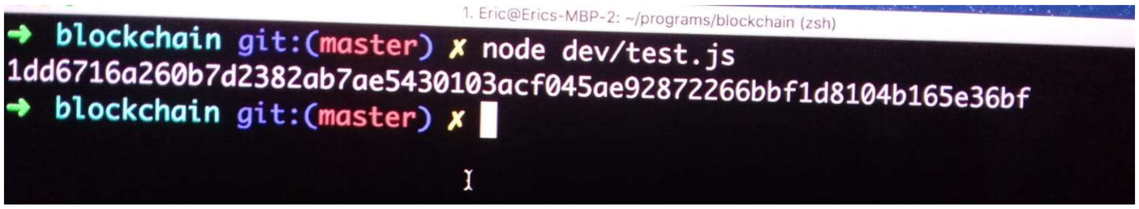
This `hashBlock` method is a crucial component of blockchain technology, as it provides a way to create a hash that uniquely represents a block's contents and can be used as part of the consensus mechanism to validate and add new blocks to the chain. The PoW process involves finding a nonce that, when combined with other block data, results in a hash with specific characteristics, typically starting with a certain number of leading zeros. Miners iterate through nonce values until they find one that meets these criteria, thus proving the computational effort they've put into securing the network.

Testing the Hashing method



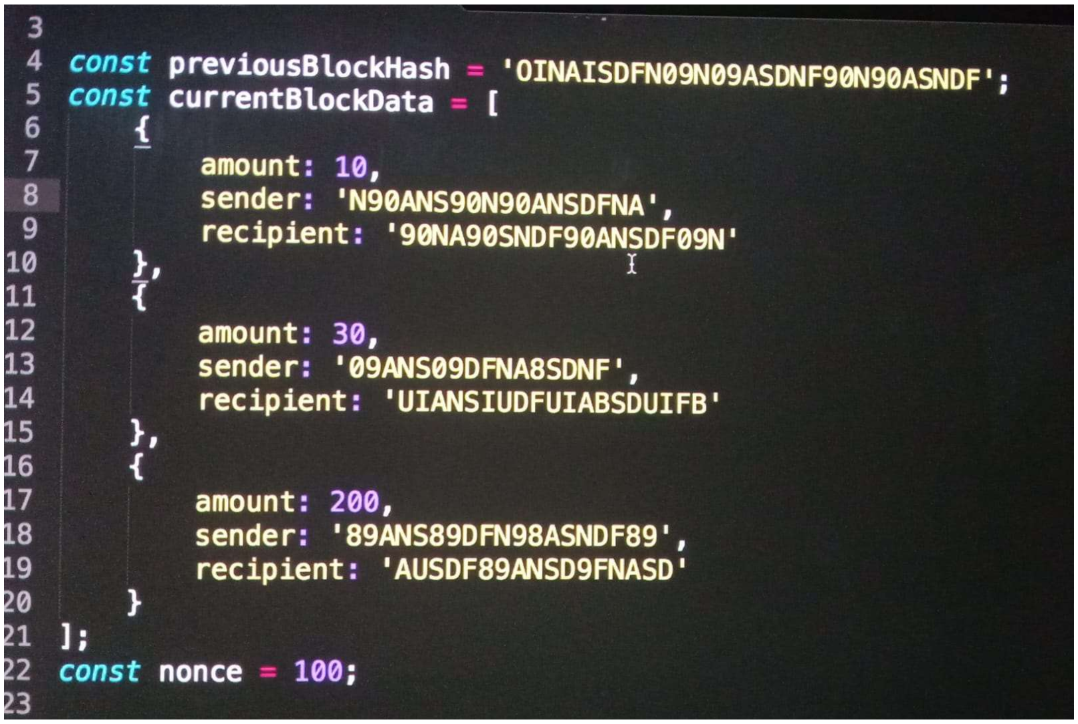
```
const previousBlockHash = '0INAISDFN09N09ASDNF90N90ASNDF';
const currentBlockData = [
    {
        amount: 10,
        sender: 'N90ANS90N90ANSDFN',
        recipient: '90NA90SNDF90ANSDF09N'
    },
    {
        amount: 30,
        sender: '09ANS09DFNA8SDNF',
        recipient: 'UIANSIUDFUIABSDUIFB'
    },
    {
        amount: 200,
        sender: '89ANS89DFN98ASNDF89',
        recipient: 'AUSDF89ANSD9FNASD'
    }
];
const nonce = 100;



console.log(bitcoin.hashBlock(previousBlockHash, currentBlockDa
```

Output



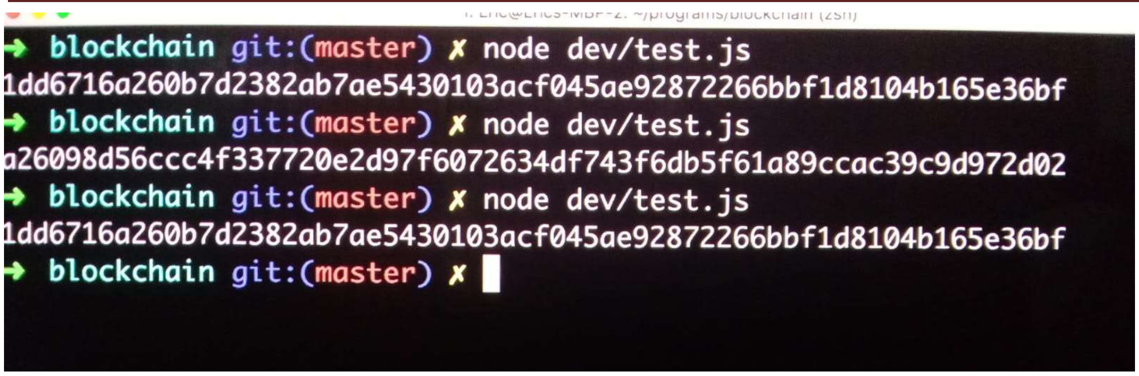
```
1. Eric@Erics-MBP-2: ~/programs/blockchain (zsh)
➜  blockchain git:(master) ✗ node dev/test.js
1dd6716a260b7d2382ab7ae5430103acf045ae92872266bbf1d8104b165e36bf
➜  blockchain git:(master) ✗ 
```

If we add an A to the sender as shown the hash value changes if we remove A the original hash value is restored



```
const previousBlockHash = '0INAISDFN09N09ASDNF90N90ASNDF';
const currentBlockData = [
    {
        amount: 10,
        sender: 'N90ANS90N90ANSDFNA',
        recipient: '90NA90SNDF90ANSDF09N'
    },
    {
        amount: 30,
        sender: '09ANS09DFNA8SDNF',
        recipient: 'UIANSIUDFUIABSDUIFB'
    },
    {
        amount: 200,
        sender: '89ANS89DFN98ASNDF89',
        recipient: 'AUSDF89ANSD9FNASD'
    }
];
const nonce = 100;

```

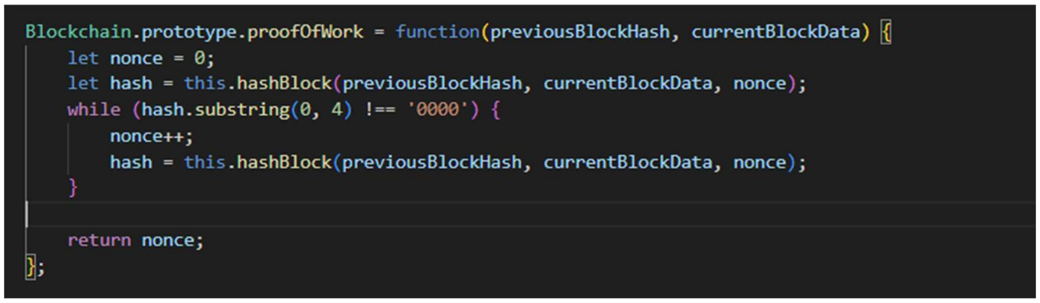
Output



```
→ blockchain git:(master) ✗ node dev/test.js
1dd6716a260b7d2382ab7ae5430103acf045ae92872266bbf1d8104b165e36bf
→ blockchain git:(master) ✗ node dev/test.js
a26098d56ccc4f337720e2d97f6072634df743f6db5f61a89ccac39c9d972d02
→ blockchain git:(master) ✗ node dev/test.js
1dd6716a260b7d2382ab7ae5430103acf045ae92872266bbf1d8104b165e36bf
→ blockchain git:(master) ✗
```

2.4 Proof of work

The `proofOfWork` mechanism is central to the concept of blockchain, especially the proof-of-work (PoW) algorithm. Its purpose is to find a suitable nonce (a number) that together with other block data forms a hash with a certain attribute, usually a predetermined number of leading zeros This process is computationally complex and it ensures that the creation of new blocks in the blockchain requires more computational effort



```
Blockchain.prototype.proofOfWork = function(previousBlockHash, currentBlockData) {
    let nonce = 0;
    let hash = this.hashBlock(previousBlockHash, currentBlockData, nonce);
    while (hash.substring(0, 4) !== '0000') {
        nonce++;
        hash = this.hashBlock(previousBlockHash, currentBlockData, nonce);
    }

    return nonce;
};
```

Here is a step-by-step explanation of the `proofOfWork` method:

1. **Initialization:**

  - The method begins by initializing `nonce` to 0. This is the starting point for the certificate-of-work process.

2. **Hash Calculation and Analysis:**

  - In the `while` loop, the method calculates the hash of the current block using the `hashBlock` method (mentioned earlier).

- Now check if the first four digits of the hash equal '0000'. This is the standard for valid hash in many PoW-based blockchains. The number of zeros can vary depending on the complexity of the blockchain.

3**. Proof-of-Work Loop:**

- If the hash does not meet the criteria (i.e., it doesn't start with '0000'), the `nonce` is incremented by 1.

- The `hashBlock` method is called again with the updated `nonce` to recalculate the hash.

- This loop continues until a valid hash is found, one that starts with '0000'.

4. Nonce Return:

- Once a suitable nonce is found, the loop exits, and the method returns the nonce.

- This nonce is a crucial part of the new block being added to the blockchain. It represents the proof that significant computational work was done to create the block.

In summary, the `proofOfWork` method plays a key role in ensuring the security and consensus of the blockchain network by requiring miners to find a nonce that, when combined with the block's data, results in a hash meeting specific criteria. This process ensures that adding new blocks to the blockchain is resource-intensive and helps prevent malicious actors from easily manipulating the blockchain.

Chapter 3

Accessing Blockchain through API

3.1 Introduction

Building an API for blockchain-related applications involves creating a set of endpoints that allow other software to interact with the blockchain network. This API enables developers to integrate blockchain functionality into their applications, such as creating and managing transactions, querying blockchain data, and more.

Express.js is a popular and widely used Node.js web application framework that will helped us build our blockchain API efficiently. Here's an overview of the steps to build a blockchain API using Express.js:

1. Set Up Your Environment:

- Install Node.js and npm (Node Package Manager) on your development machine if you haven't already.

- Create a new directory for your project and initialize it using `npm init` to set up your project with a `package.json` file.

2. Install Express:

- Install Express.js by running `npm install express` in your project directory.

3. Installing postman

3. Create the Express Application:

- Create a main JavaScript file (e.g., `app.js` or `server.js`) and require Express.

Javascript code

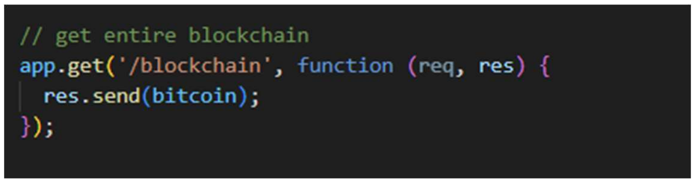
const express = require('express');

const app = express();

4. Define Your API Endpoints:

- Define the API endpoints that will interact with the blockchain. For example, you can create routes for sending transactions, checking account balances, or querying blockchain data.

Blockchain end point



```
// get entire blockchain
app.get('/blockchain', function (req, res) {
  res.send(bitcoin);
});
```

This code appears to be part of a web application that uses Node.js and Express.js, and describes how to handle HTTP GET requests to the "/blockchain" endpoint. The main goal of this method is to retrieve and send the entire blockchain, which is thought to be associated with the cryptocurrency application as it refers to the variable "Bitcoin" Let's break down the code step by step:

app.get('/blockchain', function (req, res) { ... }): This line sets up an Express.js method that listens for HTTP GET requests at the end of "/blockchain". When a client makes a GET request to this endpoint, the operation defined in the method will be performed.

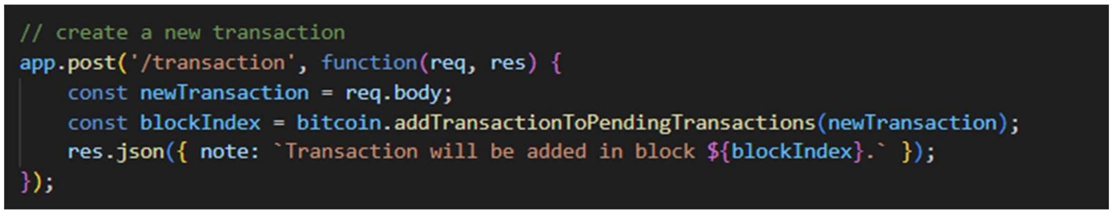
function (req, res) { ... }: This is the callback function that is executed when a GET request is sent to "/blockchain". It accepts two parameters: req (request object) and res (response object), provided by Express.js.

res.send(bitcoin);: In the callback function, the code sends a response using the res.send method. It returns the value of the variable "bitcoin" as a response.

bitcoin: Likely, "bitcoin" is a variable or entity that holds all of the blockchain data. This code sends blockchain data in response to a client making a GET request.

Overall, when a client makes a GET request to the "/blockchain" endpoint, the server will respond with the blockchain data stored in the "bitcoin" variable. It is important that the blockchain and its data are not usually exposed to the public in this way in a real-world cryptocurrency application, as it can cause security and privacy concerns In practice, API endpoints that provide blockchain data typically have authentication and access Maybe there is a way.

Transaction end point



```
// create a new transaction
app.post('/transaction', function(req, res) {
    const newTransaction = req.body;
    const blockIndex = bitcoin.addTransactionToPendingTransactions(newTransaction);
    res.json({ note: `Transaction will be added in block ${blockIndex}.` });
});
```

The code provided seems to define an Express.js method to handle HTTP POST requests to the "/transaction" endpoint. This approach aims to create a new transaction for the blockchain, possibly in a cryptocurrency application. Let’s break down the code step by step:

1. `app.post('/transaction', function (req, res) { ... })`: This line configures an Express.js method that listens for HTTP POST requests at the end of "/transaction". When a client makes a POST request to this endpoint, the operation defined in the method will be performed.

2. `const newTransaction = req.body;`: In the method call return function, this line attempts to retrieve the request body (data sent by the client in the POST request) and assign it to a calling variable no "newTransaction" This is how the code expects to get the transaction information from the recipient .

3. `const blockIndex = bitcoin.addTransactionToPendingTransactions(newTransaction);`: This line calls the `addTransactionToPendingTransactions` method on the object named "bitcoin". It takes the "newTransaction" data as an argument and adds this new transaction to the pending transaction associated with the "bitcoin" object.

4. `res.json({ note: `The article will be added to the ${blockIndex} block.` });`: After adding transactions to the pending connection, the code de JSON response is sent back to the client. This response includes a "note" property, which contains a message indicating that the statement will be added to a specific block (denoted by "blockIndex").

Overall, this code is intended to handle the creation of new transactions for the blockchain. If the client sends a POST request to the "/transaction" endpoint with the transaction data in the request body, the server will add that request to the pending list

Mining a new block end point

The code provided appears to be part of the web application, and describes an Express.js way to handle HTTP GET requests to the "/mine" endpoint. The rules seem to be in charge of mining new blocks in the blockchain and sending the new blocks to other nodes in the network. Let’s break down the code step by step:

1. `app.get('/mine', function (req, res) { ... })`: This line sets up an Express.js method that listens for HTTP GET requests at the end of "/mine". When a client makes a GET request to this endpoint, the operation defined in the method will be performed.

2. `const lastBlock = bitcoin.getLastBlock();`: This line retrieves the last block in the blockchain by calling the `getLastBlock` method on the "bitcoin" object. It stores the last block information in the "lastBlock" variable.

3. `const previousBlockHash = lastBlock['hash'];`: This line extracts the hash of the previous block from the "lastBlock" variable and stores it in the "previousBlockHash" variable.

4. `const currentBlockData = { ... };`: Here, it initializes the "currentBlockData" object, which is meant to store information about the current block, such as links and indexes

5. `transactions: bitcoin.pendingTransactions,`: The "transactions" property of "currentBlockData" is used to list the pending transactions stored in the "bitcoin" object.

6. `index: lastBlock['index'] + 1`: The "index" property of "currentBlockData" is set to the next index, which is one more than the index of the last block.

7. `const nonce = bitcoin.proofOfWork(previousBlockHash, currentBlockData);`: Calculates the nonce using the proof-of-work-work algorithm by calling the `proofOfWork` method on the "bitcoin" object This is the step especially in mining the internal process, where miners valid nonce They perform calculations to find.

8. `const blockHash = bitcoin.hashBlock(previousBlockHash, currentBlockData, nonce);`: After finding a valid nonce, this line calculates the hash of the new block using the hash of the previous block, data of the current block, and the na nonce

9. `const newBlock = bitcoin.createNewBlock(nonce, previousBlockHash, blockHash);`: This creates a new block containing the computed nonce, the previous block hash, and the block hash by calling the `createNewBlock` method in " . bitcoin" thing on it

10. `const requestPromises = [];`: This initializes an empty "requestPromises" to store promises to send new blocks to other network nodes.

11. `bitcoin.networkNodes.forEach(networkNodeUrl => { ... })`: Iterates over the network node URLs stored in the "bitcoin" object. For each network node, it places a request to send a new block to that node.

12. `const requestOptions = { ... }`: In the loop, initializes the object "requestOptions" to define the requests to send to the network node.

13. `uri: networkNodeUrl + '/get-new-block',`: Set this URI to the "get-new-block" end of the network node.

14. `path: 'POST',`: Indicates that this is a POST request.

15. `body: { newBlock: newBlock },`: This request body adds a new block.

16. `json: true`: Specifies that the response must be JSON.

17. `requestPromises.push(rp(requestOptions));`: This pushes request promises (the result of the rp operation) into the "requestPromises" structure.

18. `Promise.all(requestPromises).then(data -> { ... });`: Uses `Promise.all` to wait for all requests from other network nodes to complete. If all requests are rejected, the callback function is executed.

19. In the callback function, it defines a new "requestOptions" object to pass on the other side to the network.

20. `uri: bitcoin.currentNodeUrl + '/transaction/broadcast',`: This sets the URI to the "transaction/broadcast" endpoint of the current node.

21. `path: 'POST',`: Indicates that this is a POST request.

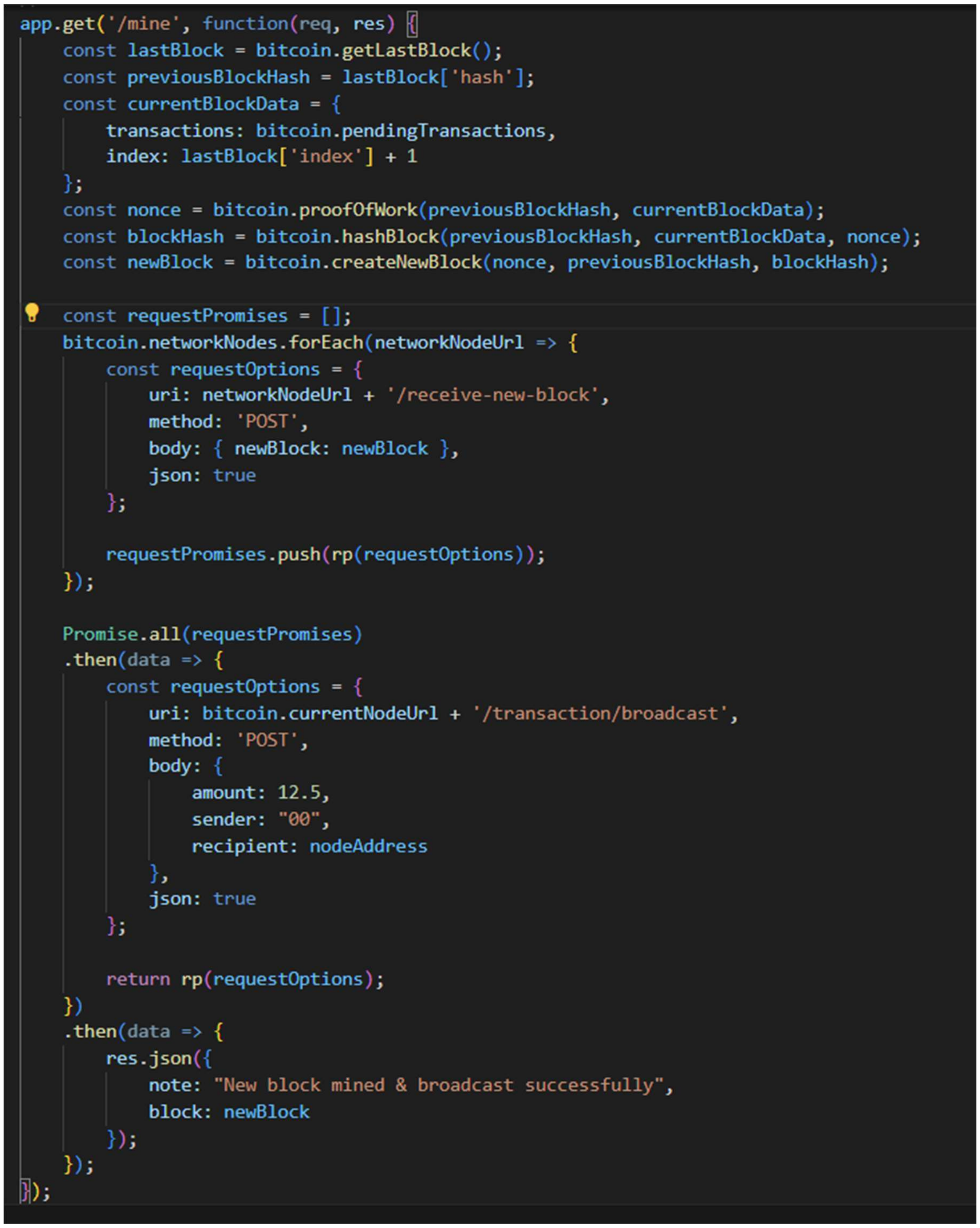
22. `body: { amount: 12.5, sender: "00", recipient: nodeAddress },`: This request body contains transaction data, which means that you will receive a mining reward of 12.5 units when approaching a recipient with "nodeAddress". has been shown to him.

23. `json: true`: Specifies that the response must be JSON.

24. `return rp(requestOptions);`: Sends a broadcast request and returns a promise.

25. Finally, when the part is moved back, the code sends a JSON message confirming that the new part has been migrated successfully and provides information about the moved part

This code is a simplified representation of the blockchain mining process and network connectivity. In a real-world scenario, you would have a robust implementation of connectivity, security, and authentication. Furthermore, the code assumes that there are methods and objects, such as "bitcoin", "proofOfWork," and "createNewBlock", that will be part of the blockchain implementation



```
app.get('/mine', function(req, res) {
    const lastBlock = bitcoin.getLastBlock();
    const previousBlockHash = lastBlock['hash'];
    const currentBlockData = {
        transactions: bitcoin.pendingTransactions,
        index: lastBlock['index'] + 1
    };
    const nonce = bitcoin.proofOfWork(previousBlockHash, currentBlockData);
    const blockHash = bitcoin.hashBlock(previousBlockHash, currentBlockData, nonce);
    const newBlock = bitcoin.createNewBlock(nonce, previousBlockHash, blockHash);

    const requestPromises = [];
    bitcoin.networkNodes.forEach(networkNodeUrl => {
        const requestOptions = {
            uri: networkNodeUrl + '/receive-new-block',
            method: 'POST',
            body: { newBlock: newBlock },
            json: true
        };

        requestPromises.push(rp(requestOptions));
    });

    Promise.all(requestPromises)
    .then(data => {
        const requestOptions = {
            uri: bitcoin.currentNodeUrl + '/transaction/broadcast',
            method: 'POST',
            body: {
                amount: 12.5,
                sender: "00",
                recipient: nodeAddress
            },
            json: true
        };

        return rp(requestOptions);
    })
    .then(data => {
        res.json({
            note: "New block mined & broadcast successfully",
            block: newBlock
        });
    });
});
```

5. Handle Blockchain Logic:

- Integrate your blockchain node or library into the API routes you defined. Use appropriate libraries and SDKs to interact with the blockchain.

6. Implement Authentication and Authorization:

- Protect your API by implementing authentication and authorization mechanisms to ensure only authorized users or applications can access it.

7. Serialize and Deserialize Data:

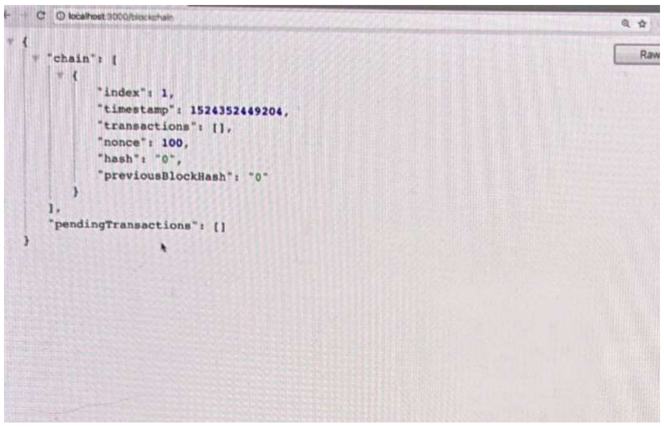
- Ensure data sent to and received from your API is properly serialized and deserialized, typically in JSON format.

8. Error Handling:

- Implement error handling middleware to provide meaningful error messages and appropriate HTTP status codes.

3.2 Testing of End points

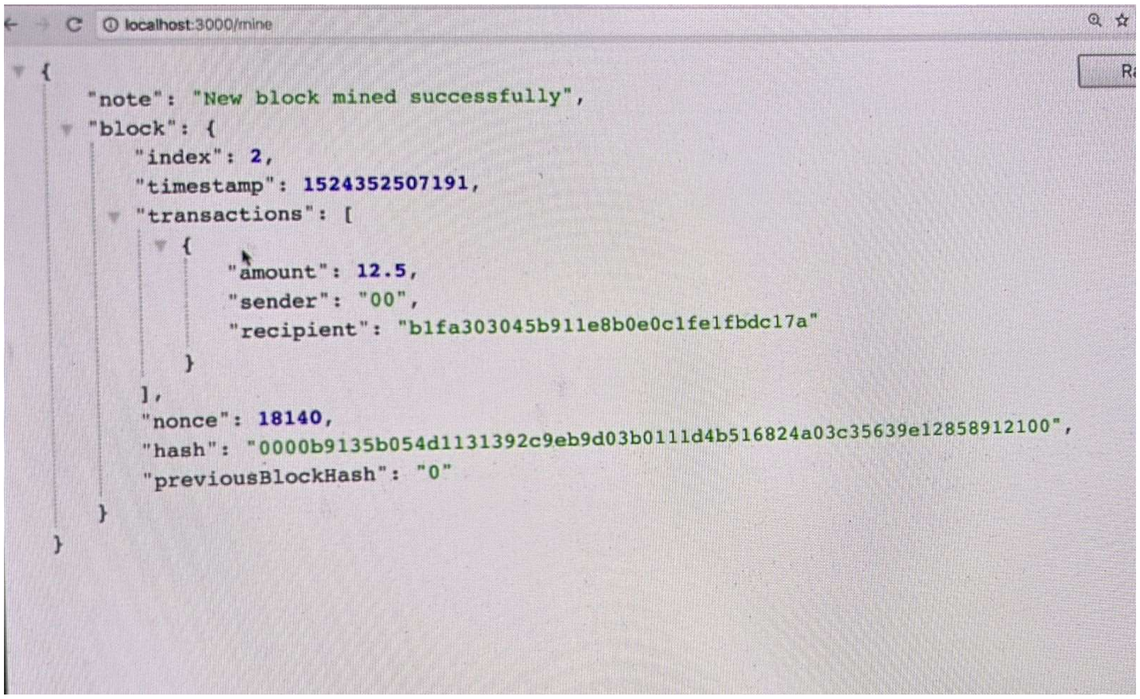
Testing \blockchain end point



```
{
  "chain": [
    {
      "index": 1,
      "timestamp": 1524352449204,
      "transactions": [],
      "nonce": 100,
      "hash": "0",
      "previousBlockHash": "0"
    }
  ],
  "pendingTransactions": []
}
```

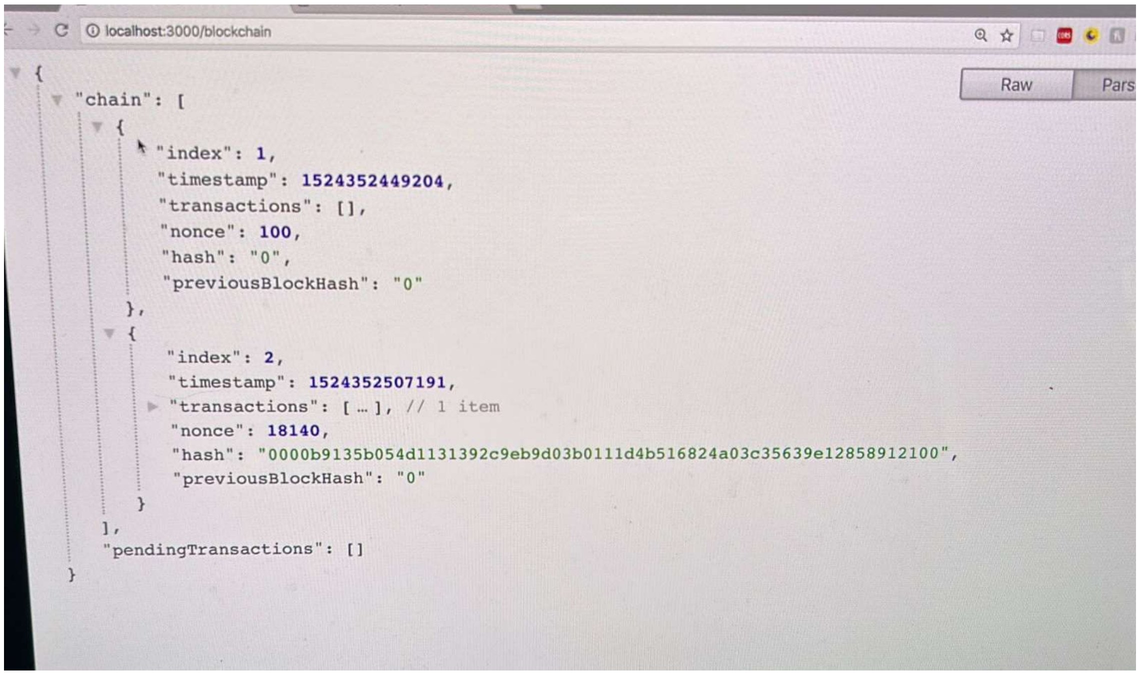
No pending transaction are found in the block chain

Testing /mine endpoint here a transaction reward is given as soon as a block is mined



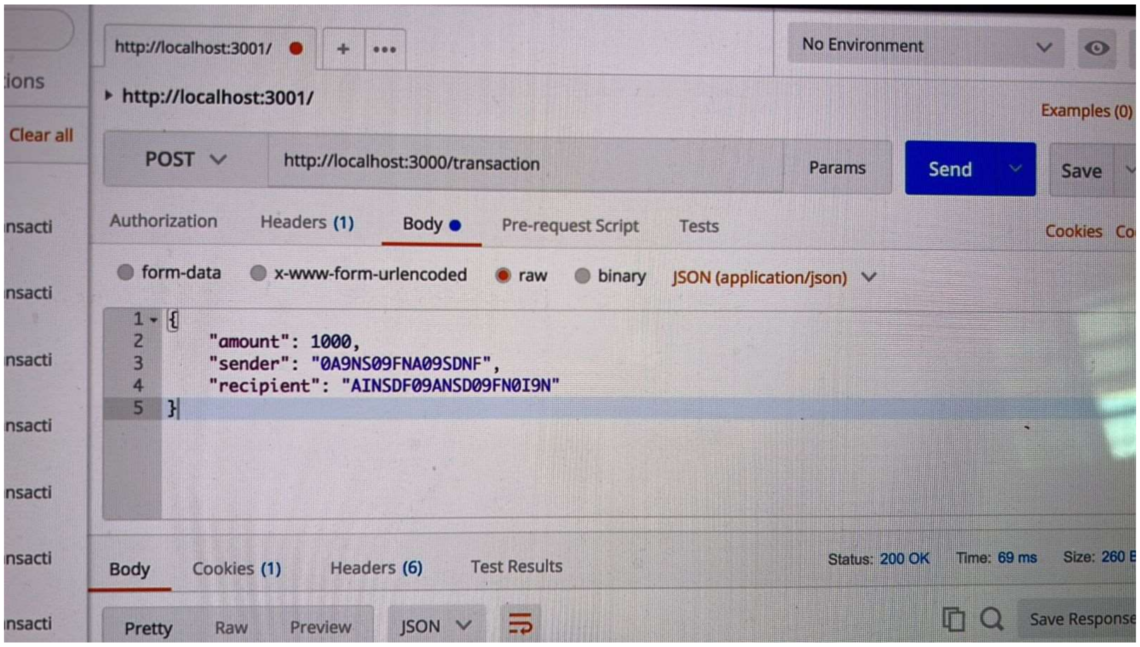
```
{
  "note": "New block mined successfully",
  "block": {
    "index": 2,
    "timestamp": 1524352507191,
    "transactions": [
      {
        "amount": 12.5,
        "sender": "00",
        "recipient": "b1fa303045b911e8b0e0c1fe1fbdc17a"
      }
    ],
    "nonce": 18140,
    "hash": "0000b9135b054d1131392c9eb9d03b0111d4b516824a03c35639e12858912100",
    "previousBlockHash": "0"
  }
}
```

Now when you refresh /blockchain endpoint you can find this new block added



```
{
  "chain": [
    {
      "index": 1,
      "timestamp": 1524352449204,
      "transactions": [],
      "nonce": 100,
      "hash": "0",
      "previousBlockHash": "0"
    },
    {
      "index": 2,
      "timestamp": 1524352507191,
      "transactions": [ … ], // 1 item
      "nonce": 18140,
      "hash": "0000b9135b054d1131392c9eb9d03b0111d4b516824a03c35639e12858912100",
      "previousBlockHash": "0"
    }
  ],
  "pendingTransactions": []
}
```

We can add transactions by using “postman” application.



POST http://localhost:3000/transaction

```
{
    "amount": 1000,
    "sender": "0A9NS09FNA09SDNF",
    "recipient": "AINSDF09ANSD09FN0I9N"
}
```

Status: 200 OK Time: 69 ms

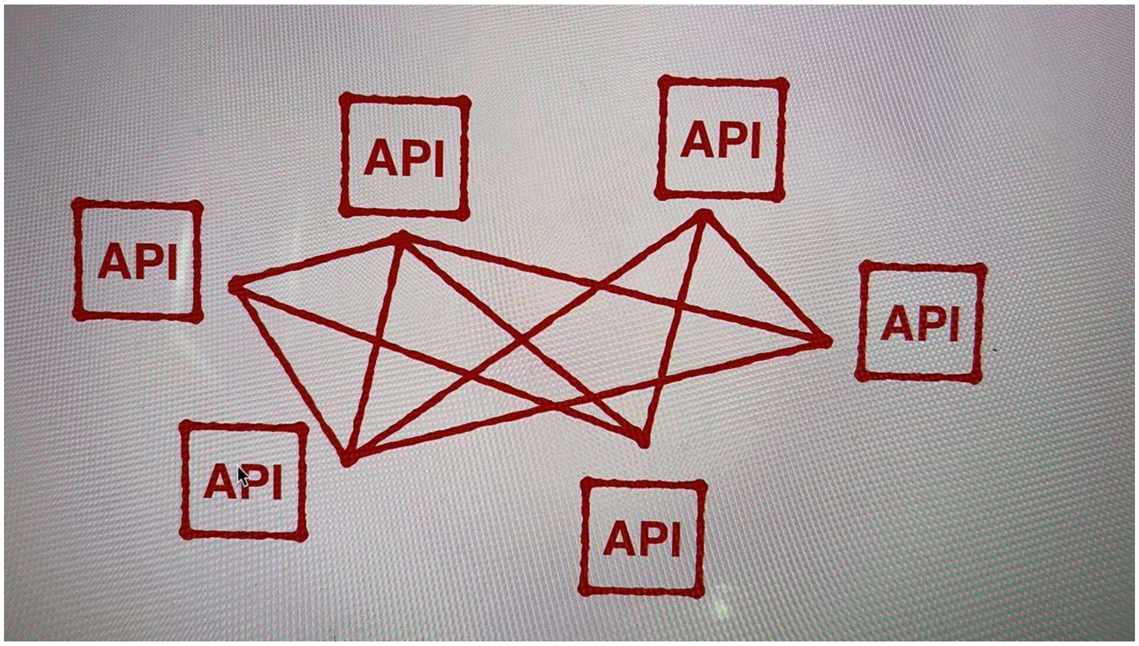
This can viewed in the pending transactions array of block chain it will be added to a blockchain when a new block is mined

CHAPTER 4

DECENTRALIZING THE BLOCK CHAIN

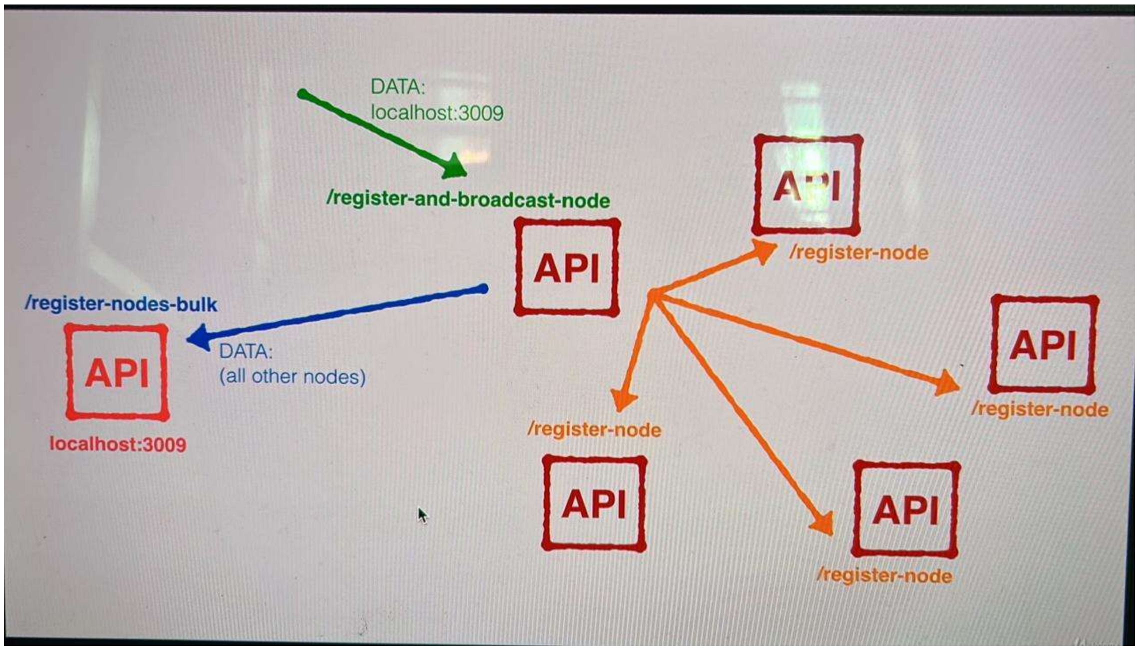
4.1 Introduction

A decentralized network, in the context of blockchain technology, refers to a network architecture where control, decision-making, and data storage are distributed across multiple nodes (computers) rather than being controlled by a single central authority. In a decentralized network, there is no central server or administrator that manages the entire system. Instead, nodes in the network work together to maintain the network's integrity, security, and functionality. Each node stores a copy of the entire blockchain ledger and participates in the consensus process.

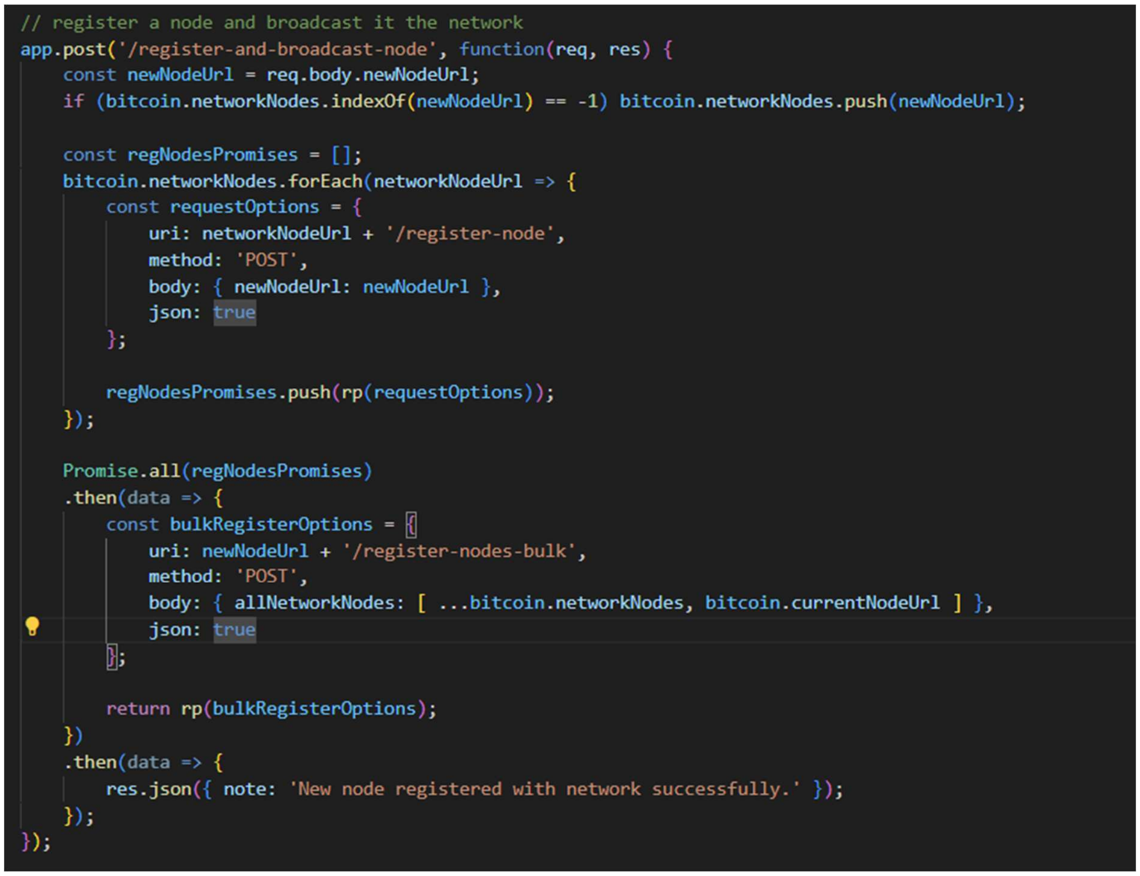


To create a decentralized network we have have created three end points

- ❖ /register-and-broadcast-node
- ❖ /register-node
- ❖ /register-node-bulk



4.2 /register-and-broadcast-node end point



```
// register a node and broadcast it the network
app.post('/register-and-broadcast-node', function(req, res) {
    const newNodeUrl = req.body.newNodeUrl;
    if (bitcoin.networkNodes.indexOf(newNodeUrl) == -1) bitcoin.networkNodes.push(newNodeUrl);

    const regNodesPromises = [];
    bitcoin.networkNodes.forEach(networkNodeUrl => {
        const requestOptions = {
            uri: networkNodeUrl + '/register-node',
            method: 'POST',
            body: { newNodeUrl: newNodeUrl },
            json: true
        };

        regNodesPromises.push(rp(requestOptions));
    });

    Promise.all(regNodesPromises)
    .then(data => {
        const bulkRegisterOptions = {
            uri: newNodeUrl + '/register-nodes-bulk',
            method: 'POST',
            body: { allNetworkNodes: [ ...bitcoin.networkNodes, bitcoin.currentNodeUrl ] },
            json: true
        };

        return rp(bulkRegisterOptions);
    })
    .then(data => {
        res.json({ note: 'New node registered with network successfully.' });
    });
});
```

code for registering and broadcasting a new node in a blockchain network, broken down into points:

1. **HTTP POST Request Handler:**

The code is executed when the server receives an HTTP POST request to the endpoint /register-and-broadcast-node.

2. **Retrieve New Node URL:**

Extract the newNodeUrl from the request's body. This URL represents the new node that wants to join the blockchain network.

3. **Check if New Node URL Exists:**

Verify if the newNodeUrl is not already in the list of known network nodes (bitcoin.networkNodes). If it's not in the list, proceed with the registration.

4. **Prepare Registration Promises:**

Create an array, regNodesPromises, to store promises for registration with existing network nodes.

5. **Register with Existing Nodes:**

Loop through each existing network node in bitcoin.networkNodes.

For each existing node, create a registration request, which includes the newNodeUrl.

Add the registration request as a promise to the regNodesPromises array.

6. **for Promises to Resolve:**

Utilize Promise.all to wait for all promises in the regNodesPromises array to complete. This ensures that all existing nodes acknowledge the presence of the new node.

7. **Bulk Registration:**

After successful registration with existing nodes, create a bulk registration request (bulkRegisterOptions).

This request includes the entire list of network nodes, which includes the new node ([...bitcoin.networkNodes, bitcoin.currentNodeUrl]).

8. Send Bulk Registration Request to New Node:

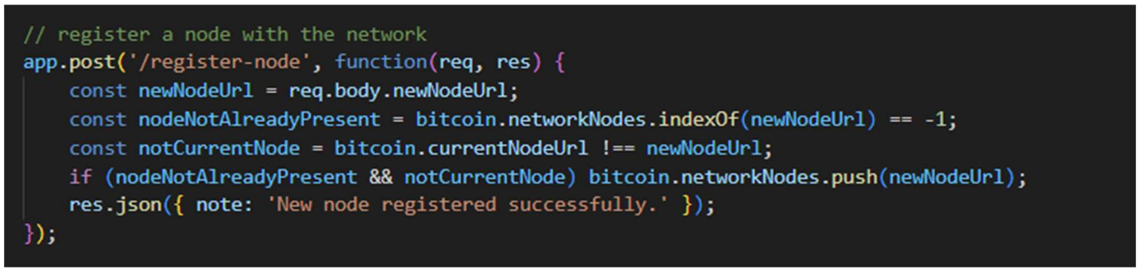
Send the bulk registration request to the new node, informing it about all other nodes in the network.

9. Send Response to Client:

Upon successful registration and broadcasting, send a response to the client with the message "New node registered with the network successfully."

This code is commonly used in blockchain networks to add new nodes and ensure that all nodes in the network are aware of each other. It plays a key role in maintaining the decentralized nature of the blockchain network.

4.3 /register-node



```
// register a node with the network
app.post('/register-node', function(req, res) {
    const newNodeUrl = req.body.newNodeUrl;
    const nodeNotAlreadyPresent = bitcoin.networkNodes.indexOf(newNodeUrl) == -1;
    const notCurrentNode = bitcoin.currentNodeUrl !== newNodeUrl;
    if (nodeNotAlreadyPresent && notCurrentNode) bitcoin.networkNodes.push(newNodeUrl);
    res.json({ note: 'New node registered successfully.' });
});
```

This code represents the registration of a new node with an existing blockchain network. Here's a breakdown of what this code does:

1. HTTP POST Request Handler:

- The code is executed when the server receives an HTTP POST request to the `/register-node` endpoint.

2. Retrieve New Node URL:

- Extract the `newNodeUrl` from the request's body. This URL represents the node that wants to join the blockchain network.

3. Check Node Presence:

- The code checks whether the `newNodeUrl` is not already present in the `bitcoin.networkNodes` array. It does this by searching for the URL in the array and evaluating if the index is -1 (indicating that the URL is not present).

4. Check Current Node:

- It also checks if the `newNodeUrl` is not the same as the current node's URL (`bitcoin.currentNodeUrl`). This ensures that the current node is not trying to register itself.

5. Register the New Node:

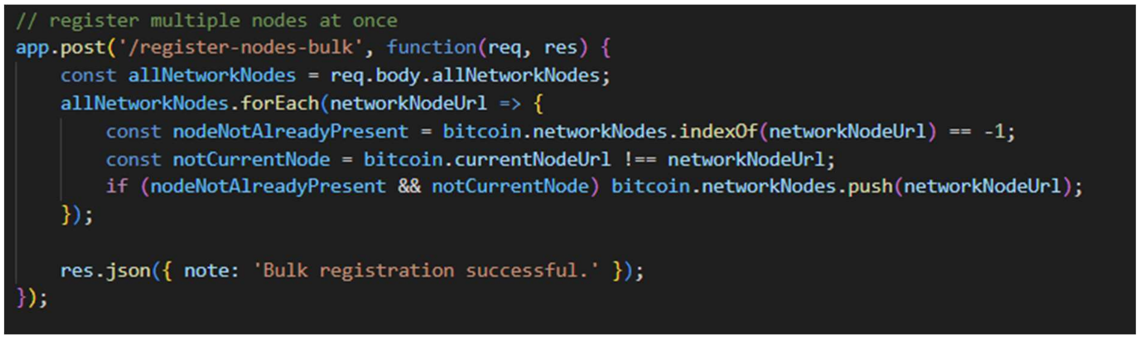
- If the `newNodeUrl` is not already present in the network and is not the current node, it adds the `newNodeUrl` to the `bitcoin.networkNodes` array. This array is used to keep track of all nodes in the network.

6. Send Response:

- Respond to the client with a JSON message stating "New node registered successfully."

This code is a part of the process where new nodes can join an existing blockchain network. It helps maintain the list of network nodes, allowing the blockchain to function as a decentralized network.

4.4 /register-node-bulk



```
// register multiple nodes at once
app.post('/register-nodes-bulk', function(req, res) {
    const allNetworkNodes = req.body.allNetworkNodes;
    allNetworkNodes.forEach(networkNodeUrl => {
        const nodeNotAlreadyPresent = bitcoin.networkNodes.indexOf(networkNodeUrl) == -1;
        const notCurrentNode = bitcoin.currentNodeUrl !== networkNodeUrl;
        if (nodeNotAlreadyPresent && notCurrentNode) bitcoin.networkNodes.push(networkNodeUrl);
    });

    res.json({ note: 'Bulk registration successful.' });
});
```

This code allows for the bulk registration of multiple nodes at once with an existing blockchain network. Here's a breakdown of what this code does:

1. HTTP POST Request Handler:

- The code is executed when the server receives an HTTP POST request to the `/register-nodes-bulk` endpoint.

2. Retrieve All Network Nodes:

- Extract the `allNetworkNodes` from the request's body. This array contains the URLs of all the nodes that want to join the blockchain network.

3. Iterate Through Network Nodes:

- The code loops through each `networkNodeUrl` in the `allNetworkNodes` array.

4. Check Node Presence:

- For each `networkNodeUrl`, it checks whether the URL is not already present in the `bitcoin.networkNodes` array. It does this by searching for the URL in the array and evaluating if the index is -1 (indicating that the URL is not present).

5. Check Current Node:

- It also checks if the `networkNodeUrl` is not the same as the current node's URL (`bitcoin.currentNodeUrl`). This ensures that the current node is not trying to register itself.

6. Register New Nodes:

- If the `networkNodeUrl` is not already present in the network and is not the current node, it adds the `networkNodeUrl` to the `bitcoin.networkNodes` array. This array is used to keep track of all nodes in the network.

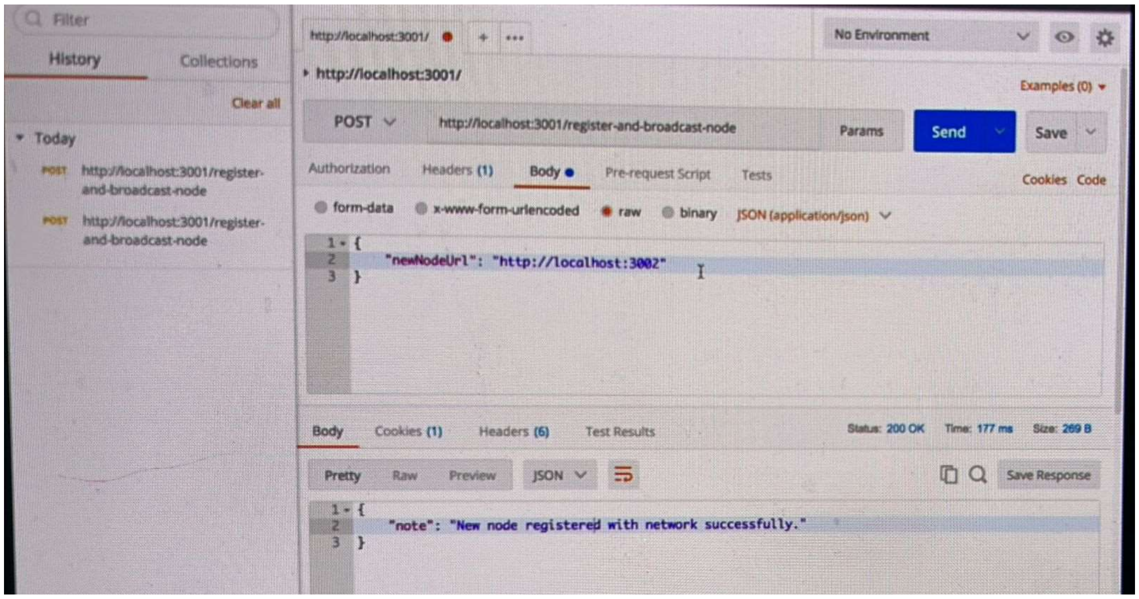
7. Send Response:

- Respond to the client with a JSON message stating "Bulk registration successful."

This code streamlines the process of registering multiple nodes with an existing blockchain network in a single request, ensuring that all nodes are aware of each other. It is a key part of maintaining the decentralized nature of a blockchain network.

4.5 Testing all the network Endpoints

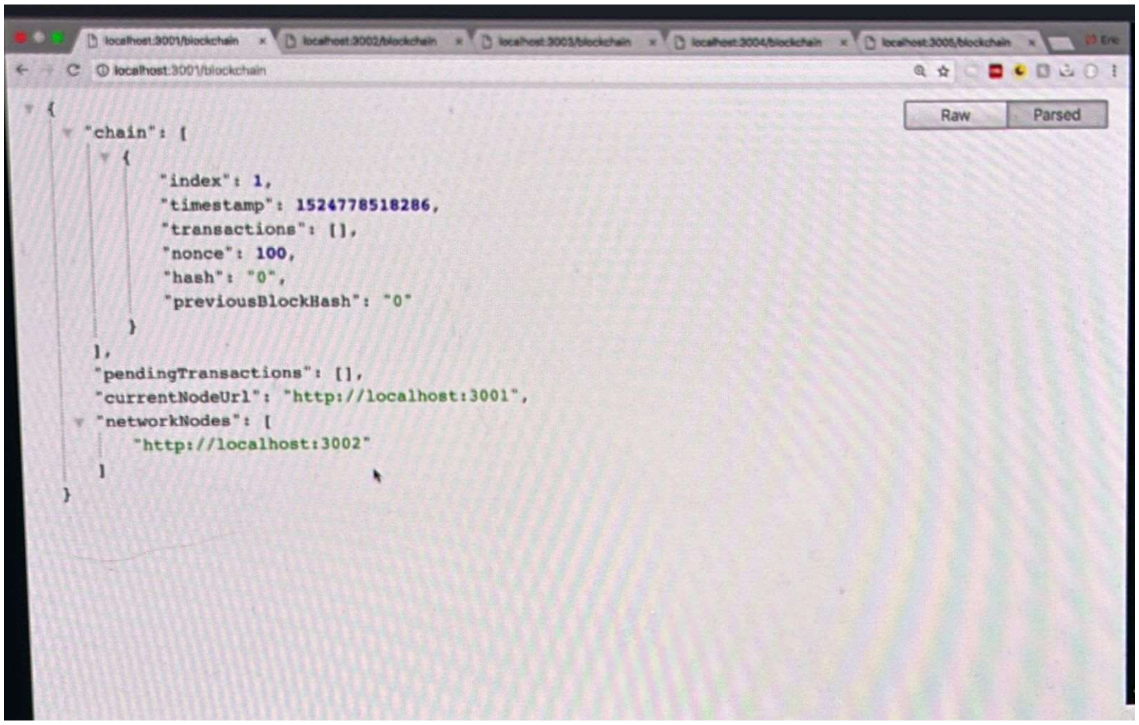
Here we are registering node 2 with node 1



```
{
    "newNodeUrl": "http://localhost:3002"
}
```

```
{
    "note": "New node registered with network successfully."
}
```

we can observe that node 2 has been added to node 1 in the network nodes array



```
{
  "chain": [
    {
      "index": 1,
      "timestamp": 1524778518286,
      "transactions": [],
      "nonce": 100,
      "hash": "0",
      "previousBlockHash": "0"
    }
  ],
  "pendingTransactions": [],
  "currentNodeUrl": "http://localhost:3001",
  "networkNodes": [
    "http://localhost:3002"
  ]
}
```