Blockchain implemented in JavaScript tested using Postman and Jest Testing Framework

Kavya Nannaka, Sherief Elsowiny

Introduction: 3

Objective: 3
- Jest 4
- Postman 4

Test Plan and flow: 7
- Black box testing partitions 7
- White box testing plan 8
- Essential units to test 9

Test case design 9

Test Execution and results 11
- Black Box Results 11
- White Box Results 14

Conclusion: 19

Appendix: 19
- Block Class 20
- Blockchain Class 21
- Code Coverage Report 22
- Block test (a) 22
- Block test (b) 23
- Blockchain test (a) 24
- Blockchain test (b) 25
- Blockchain test (c) 26
- Blockchain test (d) 27
- Blockchain test (d) 28

References: 28

Introduction:

Blockchain technology has seen a rise in popularity with many startups trying to capitalize on the trending technology and build communities and products around it. Why is this trending technology of Blockchain so enticing however? Well, Blockchain is known for its powerful attributes of being transparent, having a public ledger available for viewing, and immutability [1]. Many cryptocurrencies are built on top of Blockchain, and take these powerful attributes and market it as a reliable form of currency where transactions can be easily viewed, and the Blockchain reliable. It's known to possess, internally, certain functionality that allows it to be resilient in tampering or intelligent enough to handle the mine rate. Luckily for us, we have a simple Blockchain technology on hand built in JavaScript that we have tested. This simple software has public endpoints available for testing as an end user, in which a user would want to create or view blocks on the Blockchain. The software is also open-source, allowing us to test in a white box manner the functionalities that are integral to a solid Blockchain. In our requirements analysis, we decided to test the important factors of what a Blockchain should have, such as what occurs when the chain is tampered with in various ways, or the event of the chain being replaced successfully. We used Jest for this testing, allowing us to test the individual code blocks and produce an accompanying coverage report. When it came to the front end portion of what an end user would come across, we used Postman to test the API functionality in a black box manner ignoring the intrinsics of the software, focusing on I/O operations that a user would do.

Objective:

We plan to implement both white box and black box testing and other various testing. In whitebox testing we will focus on code coverage. For blackbox testing, we aim to focus on testing the functional requirements of the software. Several techniques we may use include requirements-based testing, boundary value testing, and negative-positive testing.

Software tools:

Jest

Jest is a popular testing library built by Facebook and used by many famous companies. It simplifies a lot of configuration and allows us to create easy tests by creating files ending in .test. We can define test suites for our code where related tests are contained by using a Jest function called “describe”. Inside our test suites, we create various tests, mocking different functionality or logic. An example taken

Arrays and iterables

You can check if an array or iterable contains a particular item using `toContain`:

```
const shoppingList = [
  'diapers',
  'kleenex',
  'trash bags',
  'paper towels',
  'milk',
];

test('the shopping list has milk on it', () => {
  expect(shoppingList).toContain('milk');
  expect(new Set(shoppingList)).toContain('milk');
});
```

from the site is given below [2].

Postman

Postman is a tool used to test API’s and their responses, coming equipped with all the needs an application would need to test an API’s endpoints. In our testing, we make use of Postman for the testing of the public facing endpoints of the Blockchain API. We are able to test what a user would do when requesting the API, as well as test the other features associated with API’s such as the protocol used and how the API handles it. This will allow us to mimic whether the user is

creating or trying to receive a resource, as well as what happens when improper fields are put in.

Steps followed:

- Defining the Blockchain Cryptocurrency requirements and specific software requirements.
- Understanding the functionality of the Jest and postman tools

Description:

In terms of black box testing, we focus on two front facing endpoints that the software offers and explore two perspectives as the end user.
In case one, we consider the end user as anybody who would want to access the API and its accompanying functionality. In this regard, we simply need to test the public endpoints of what is offered by the software: retrieving the blockchain and the creation of a new block. This can be tested easily using Postman.

In the second case, we consider the end user to be for example another developer who may decide to adopt this API into their own software. We then expand on our test cases to include a few more cases where we explore specific protocols used in API communication. When using Postman, the tool to test APIs, we can mimic a realistic interaction that a user would have in the browser. Here we will be assessing the responses of the server that is running the program with specific protocols that a browser would implement. To be more precise, when a user accesses a resource on the browser, they use a method known as GET request. In our exploration as the end user, we can mimic the behaviors a realistic user would use by performing these actions through Postman, albeit with the appropriate protocols attached.

When looking deeper into the source code, we come across white box testing where our knowledge of the code is apparent, and we can dive into the underlying functions and logic that drives the application. Take into consideration the necessities to deliver proper software. A lot of error handling and proper structure of the code is needed to not only deliver something sustainable, but also something less prone to break in production. In our viewing of the Blockchain through the

lens of white box testing, we can observe specific requirements that a blockchain software or any software in that manner would need. When one wants to use a Blockchain, they typically expect their data to be secured. The objective of Blockchain technology is to rely on cryptography to hash and construct valid blocks in a link that all relate to each other in some capacity while also containing data. Given that we know our software's source code, we can explore first the block class that encompasses the Blockchain, then expand into the Blockchain itself testing for cases of functionality and adequate handling of tampering.

When consolidating this into a test plan, our analysis of our requirements is split into considering the software itself, and the public endpoints that an end user would use. The following are what we decided on for our requirements.

We decided on the following requirements from the front facing API of the Blockchain.

- Ability to create a new block on the Blockchain
- Ability to view the Blockchain(transparency requirement)
- Proper protocols used in our requests I.E
    - POST - Request (Application should respond correctly)
    - GET - Request
- Error Handling and validation

When it came to testing the block and Blockchain itself, we decided on the following requirements.

Block

- Should create a block based on adequate inputs
    - Returns a block instance
    - References the last hash of the previous block
    - Creates a hash itself/stores data inside the block
    - Adjusts difficulty for quickly mined block

Blockchain

- Should always start with the same genesis block to indicate the beginning of the ledger
- Ability to add Block with data to Blockchain
- Ability to replace Blockchain with a new valid chain

Testing for validity of chain

- Testing for ability to view chain
- Test for ability to mine block
- Test for validity of block
- A valid hash

Limitations:

More features of the code are available for testing and have been covered in the coverage report. Although more details and testing would have been desirable, we both faced time constraints and decided to focus on the practical aspects of the software.

Test Plan and flow:

The software is expected to work in a way the user creates a new block and can view it. This is the flow of what the user can expect in simplest terms. When it comes to testing this, we can derive partitions to describe validity and use them to help our test cases later. We can similarly do the same when it comes to constructing tests for our white box testing.

Black box testing partitions

Module: **api/mine**
Notes: Creating a new block as a user

Partition Type	**Partition Description**	**Test cases**
Valid	Value is part of valid data sets	{“data”:”string value”} {“data”: 4} {“data”: {“object”:”value”} } {“data”: [“list”,0,”1”]}
Invalid	No data supplied	Empty JSON

Module: **api/mine**
Notes: Creating a new block API methods

Partition Type	**Partition Description**	**Test cases**
Valid	Method is GET request	GET /api/mine
Invalid	Method is anything else	POST /api/mine, Delete, Put

Module: **api/blocks**
Notes: Viewing the blocks on the blockchain using the correct method

Partition Type	**Partition Description**	**Test cases**
Valid	Method is GET request	GET /api/blocks
Invalid	Method is anything else	POST /api/blocks, Delete, Put

White box testing plan

With our white box testing plan, we decided to achieve 100% code coverage by testing the individual units of code and their respective conditions. Being able to see the code allowed us to decide on what was optimal for testing. We decided that in terms of the flow for our block, we have the flow of a user mining a block and retrieving a valid block, as well as the case of what occurs when a block is mined too quickly. The difficulty of the mining should either increase or decrease

depending on the speed of the previous mined block. When it comes to the flow of the blockchain, most notably we have the event of adding a new block to the Blockchain to which we can test for possible situations of data being tampered with in the Blockchain. Similarly, we have the event of the chain being replaced but in the conditions it is valid. We were able to derive test cases for this as well by planning to test for validity when replacing the chain. To reduce redundancy in terms of explaining the simple test cases needed to achieve full coverage, we will simply go over the key essential tests and include the coverage at the end. Nonetheless, we have the following units to test.

Essential units to test

Block - Ensuring proper fields are computed for coverage report, and testing for difficulty adjustments

Blockchain - Ensuring proper fields are computed for coverage report, and testing for chain replacement, and specifically tampering when replacing the chain

Test case design

Test case Id	Unit to test	Assumptions	Test Data	Steps to be executed	Expected Result	Type	Method covered
1	Api - mine block	Should be able to create a block with data	data = {dummy}	Create a request to endpoint with data	Block with data	Black Box	x
2	Api - mine block	Should not be able to create a block without data	none	Create a request with no data	Error w/ message	Black Box	x
3	Api - mine block	Should not be able to create a block using this method	Data = any, Method = !POST	Create a request with any data not using POST method	Error w/ message	Black Box	x
4	Api - view blocks	Should be able to view the blockchain	None needed/any/ DC	Create a request to view the blockchain	Blockchain is returned	Black Box	x
5	Api - view blocks	Should not be able to view the blockchain	No data needed/DC Method =	Create a request to view the blockchain	Error w/ message	Black Box	x

		unless using a GET request	!GET	not using a GET request			
6	Block	Should be able to create a block	Genesis block for testing: Mined block: const data = 'mined data';	Retrieve last block on the blockchain Use dummy data to mine new block	New mined Block on the Blockchain	White Box	mineBlock
7	Block	Should raise difficulty on quickly mined block	Test Block, Timestamp difference being less than the config's required I.E 1000ms	Use data to call adjust function on the block Assess the returned difficulty	Difficulty should increase by 1	White Box	adjustDifficulty
8	Block	Should lower difficulty on slowly mined block	Test Block, Timestamp difference being greater than the config's required	Use data to call adjust function on the block, assessing the returned difficulty	Difficulty should decrease by 1	White Box	adjustDifficulty
9	Block	Should have lower limit bound of 1	Lower difficulty beyond lower bound such as 0, -1 , etc	Call adjust function on new block with lowered difficulty	Difficulty should be 1	White Box	adjustDiffculty
10	Blockchain	Should be able to add a block	newData = any	Call function on Blockchain to add new Block with new data	Blockchain should add new block	White Box	addBlock
11	Block Chain	Invalid chain when first block is not genesis block	Anything except the original chain's data	Call on the first block of the blockchain and change its data to anything other than the current data	Blockchain should return invalid	White Box	isValidChain
12	Block Chain	Invalid chain when a blocks hash on the chain is tampered	Any other hash aside from the one you are altering	Call a block and alter it's hash	Blockchain should return invalid	White Box	isValidChain

13	Block Chain	Invalid chain when a block has an invalid field	Any other data than the original you are altering	Call a block on the blockchain and change its data to anything other than the current data it holds	Blockchain should return invalid	White Box	isValidChain
14	Block Chain	Replaces Blockchain if Blockchain is longer and valid	Longer and Valid Blockchain	Create new Blockchain that is valid I.E similar to the old chain Add new blocks to the Blockchain Call replace function	Blockchain should be replaced	White Box	replaceChain
15	Block Chain	Does not replace chain when new chain is not longer	Incoming chain is not longer	Create a new Blockchain not longer than the chain you are replacing and call to replace the chain	Blockchain should not be replaced	White Box	replaceChain
16	Block Chain	Does not replace chain when new chain is longer but invalid	Incoming chain is longer but invalid	Create a new Blockchain longer, but invalidate it by tampering the data	Blockchain should not be replaced	White Box	replaceChain

Test Execution and results

Black Box Results

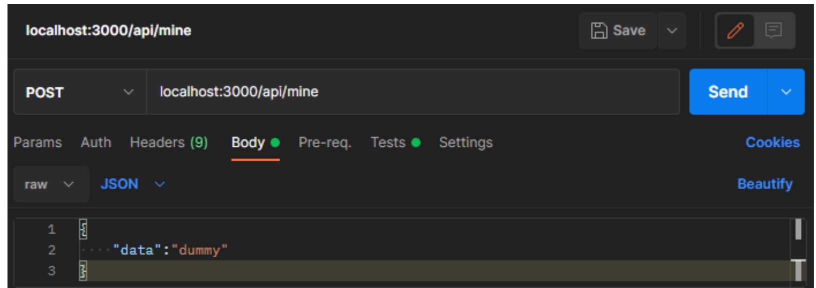
METHOD
localhost:3000/api/mine
GET
POST
PUT
PATCH
DELETE
Body
Pre-re


```
1  [
2      {
3          "timestamp": 1,
4          "lastHash": "--",
5          "hash": "sample-hash",
6          "data": [],
7          "nonce": 0,
8          "difficulty": 5
9      },
10     {
11         "timestamp": 1638818916546,
12         "lastHash": "sample-hash",
13         "hash": "085ce824f415f63948
14         "data": "dummy",
15         "nonce": 25,
16         "difficulty": 4
17     }
18 ]
```


Here we performed manual testing with Postman and included snippets of what our test looks like. When we make a request to test for validity, we use the appropriate header method, while later testing for the differences in methods.

```
    "timestamp": 1638818916546,
    "lastHash": "sample-hash",
    "hash": "085ce824f415f63948bfe824
    "data": "dummy",
    "nonce": 25,
    "difficulty": 4
  },
  {
    "timestamp": 1638819179752,
    "lastHash": "085ce824f415f63948bf
    "hash": "0f8271af70c20028ed3d4316
    "nonce": 10,
    "difficulty": 3
  }
]
```

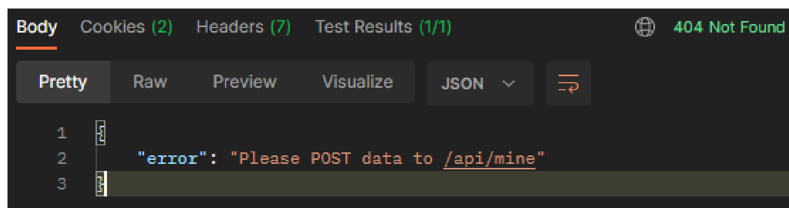

Here is an image displaying how we structured our data to be called with our first test being executed calling to create a new block.
Our response can be seen in the following image, where we see the blockchain returned. We marked this test successful and continued.



For the following test we omitted data, expecting to incur an error. However, we realized we obtained the Blockchain still with the particular Block lacking a data field. We concluded this was an error, as there should be a middleware field that checks for data being inserted.
For the following tests we tested using dummy data and focused on changing the method in which we obtain the data.

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <meta charset="utf-8">
6      <title>Error</title>
7  </head>
8
9  <body>
10     <pre>Cannot GET /api/mine</pre>
11 </body>
12
13 </html>
```

We included an image showcasing how we easily changed from a POST request to a GET request, subsequently testing for the different methods available.



The results of the different methods showcased a 404 error, however no proper error message or such occurred other than the one that is naturally produced. We concluded the test was a fail, as we
required the chain to produce a better detailed error response for any API user. The error did return a 404, which could be used to handle control flow, but with the methods available for API's and the methods needed of the API, we required that the API itself send the error and not one be generated by the testing tool.

Nonetheless, we included an image to showcase the error response we received, where the method we tested for other than POST was included in the error response(generated automatically, not by the API).
To give an example of this, we also included an image of what an appropriate response should look like for what our expected results were defined as. The response appropriately showcases a 404-error status in this case, but also includes an appropriate error message. Following this, we tested for the viewing of the Blockchain which returned as expected, and subsequently tested similarly the different methods. Our expectations at this point were that we were going to receive the same unimplemented error message as received in the previous tests for the other endpoint. As we expected, our expectations were correct, and we debated whether or not the API should have even implemented that endpoint for methods other than GET and concluded that a proper error message should still be included but that it passes on not returning the Blockchain for that method. We further concluded that middleware validation should be attached as this could further decrease any attacks for users that may eventually be a part of this Blockchain.

Test ID	Actual Result	Pass/Fail	Comments	Type
1	Returns entire Blockchain	Pass		Black Box
2	Returns entire Blockchain	Fail	Developers should include middleware to check for data field, or indicate what is needed	BB
3	Autogenerated error message	Pass on receiving 404 error, Fail on error message	Developers should include an appropriate error message such as the one supplied	BB
4	Returns entire Blockchain	Pass		BB
5	Autogenerated error message	Pass on not allowing the Blockchain to be viewed unless using a GET request, Fail on the error message provided	Developers should include an appropriate error message, however we concluded this was least priority and the test passes on the other methods at least being unavailable Middlewares should	BB

			be more of a priority	

White Box Results

When it came to testing the internal code, we focused on covering the different methods within the code as well as the statements and different logic. We included the testing in the reports section, but for testing the critical components, we included imagery and descriptions. The report showcases that all the code has been tested, but for the testing of the initial block and such, we omitted this.

Here we used Jest to create our test suites and test cases. We asserted our expectations for that function and logged the results.

```
static mineBlock({lastBlock, data}){
    let hash, timestamp;
    const lastHash = lastBlock.hash;
    let { difficulty } = lastBlock;
    let nonce = 0;
    do{
        nonce++;
        timestamp = Date.now();
        difficulty = Block.adjustDifficulty({originalBlock: lastBlock, timestamp});
        hash = cryptoHash(timestamp, lastHash, data, nonce, difficulty);
    } while(hexToBinary(hash).substring(0, difficulty) !== '0'.repeat(difficulty));
    return new this({
        timestamp,
        lastHash,
        data,
        difficulty,
        nonce,
        hash
    });
}
```

```
    describe('mineBlock()', () => {
        const lastBlock = Block.genesis();
        const data = '';
        const minedBlock = Block.mineBlock({lastBlock, data });
        it("return a Block instance", () => {
            expect(minedBlock).toBeInstanceOf(Block);
        });
        it('sets the `lastHash` to be the `hash` of the lastBlock', () => {
            expect(minedBlock.lastHash).toEqual(lastBlock.hash);
            console.log('lastHash', minedBlock.lastHash);
            console.log('hash', lastBlock.hash);
        });
        it('sets the `data`', ()=>{
            expect(minedBlock.data).toEqual(data);
        });
        it('sets the `timestamp` to be the current time', ()=>{
            expect(minedBlock.timestamp).not.toEqual(undefined);
        });

        it('create a SHA-256 `hash ` based on the proper inputs', ()=>{
            expect(minedBlock.hash).toEqual(
                cryptoHash(minedBlock.timestamp,
                    minedBlock.nonce,
                    minedBlock.difficulty,
                    lastBlock.hash,
                    data
                )
            );

        });
        it('sets a `hash` that matches the difficulty criteria', () =>{
            expect(hexToBinary(minedBlock.hash).substring(0, minedBlock.difficulty))
            .toEqual('0'.repeat(minedBlock.difficulty));
        });
        it('adjusts the difficulty', () =>{
            const possibleResults = [lastBlock.difficulty + 1, lastBlock.difficulty - 1];
            expect(possibleResults.includes(minedBlock.difficulty)).toBe(true);
        });
        console.log('minedBlock', minedBlock);
});
```

The mining function associated with the

block class appears as follows.

```
static adjustDifficulty({originalBlock, timestamp}){
    const {difficulty} = originalBlock;
    if(difficulty < 1){return 1;}
    const difference = timestamp - originalBlock.timestamp;
    if(difference > MINE_RATE){
        return difficulty - 1;
    }
    return difficulty + 1;
}
```

To effectively test for this method of obtaining a new mined block, we configured some dummy data, and tested the appropriate fields that the new mined block should have.

We focused on the essentials of requiring the hash be set and that it be matched to the difficulty criteria. We console out our results and example mined block and take notice that it passes as expected.

```
console.log
  minedBlock Block {
    timestamp: 1638823794358,
    lastHash: 'sample-hash',
    hash: '05858e2913af9ad3bf8543a244
    data: '',
    nonce: 4,
    difficulty: 4
  }
```

```
console.log
  changing difficulty to -10

    at block.test.js:104:13

console.log
```

```
mineBlock()
  √ return a Block instance (1 ms)
  √ sets the `lastHash` to be the `hash` of the lastBlock (3 ms)
  √ sets the `data` (1 ms)
  √ sets the `timestamp` to be the current time
  √ create a SHA-256 `hash ` based on the proper inputs
  √ sets a `hash` that matches the difficulty criteria (1 ms)
```

For our difficulty adjustment , we can easily see the source code and make a conclusion that the test will pass based on the code itself. Nonetheless, we include the image to showcase the source code. Here we test for three possible conditions and their respective cases. To be more precise, we test for the case when the difficulty is either going to be increased, decreased and the case where the lower bound is reached. The difficulty adjustment function is used to aid in configuring the Blockchains rate of mining, so we test this to make sure it behaves as expected.

```
describe('adjustDifficulty()', ()=>{
    it('raises the difficulty for a quickly mined block' , () => {
        expect(Block.adjustDifficulty({
            originalBlock: block,
            timestamp: block.timestamp + MINE_RATE - 100
        })).toEqual(block.difficulty + 1);
    });
    it('lowers the difficulty for a slowly mined block' , () => {
        expect(Block.adjustDifficulty({
            originalBlock: block,
            timestamp: block.timestamp + MINE_RATE + 100
        })).toEqual(block.difficulty - 1);
    });

    it('has a lower limit of 1', () => {
        block.difficulty = 0;
        expect(Block.adjustDifficulty({
            originalBlock: block
        })).toEqual(1);
    });
    console.log('block', block);

})
```

We alter the timestamp that is used as a parameter in the adjust functionality , to mimic the scenario of a block that has been mined too quickly or too slowly. The adjust function depends on the mine rate constant that is defined in configuration. What occurs here is there is a defined rate defined in milliseconds in configuration that determines whether the difference

in mined blocks should result in a different difficulty adjustment. Here we can make assertions and also log the new difficulty to see our test in action.

The next part is where we focus on the functionality of the Blockchain itself and preview the Blockchain class and the associated functions. We included the Blockchain class in the appendix, as well as

```
adjustDifficulty()
  √ raises the difficulty for a quickly mined block (1 ms)
  √ lowers the difficulty for a slowly mined block (1 ms)
  √ has a lower limit of 1
```

```
console.log
```

```
it('adds a new block to the chain' , () =>{
    const newData = 'hey test data';
    blockchain.addBlock({data: newData});
    console.log("blockchain after");
    console.log(blockchain.chain);
    expect(blockchain.chain[blockchain.chain.length - 1].data).toEqual(newData);
    //previous hash is the hash of the previous block
    expect(blockchain.chain[blockchain.chain.length - 1].lastHash).toEqual(blockchain.chain[blockchain.chain.length - 2].hash);
});
```

the block class. For the Blockchain, we focus on testing the addBlock function, as well as the validation of the chain itself in which it expects the first block to be the genesis block and the chain to not be altered. When previewing the source code, it's easy to see what the result will be by glancing at the associated logic in the validation. The function itself appears to recreate the chain and test for if any fields have been altered. We will essentially be testing this test. We begin by testing the ability to add a new block to the Blockchain. As expected, we were able to add a block to the Blockchain, and in various data formats. We take note of the requirement that the newly added block should reference the last hash. We also test for this assertion.

```
console.log
  blockchain after

  at Object.<anonymous> (blockchain.test.js:26:17)

console.log
  [
    Block {
      timestamp: 1,
      lastHash: '--',
      hash: 'sample-hash',
      data: [],
      nonce: 0,
      difficulty: 5
    },
    Block {
      timestamp: 1638828000504,
      lastHash: 'sample-hash',
      hash: '03f8f6418a246ba05ae779d05802effe87978d94
      data: 'hey test data',
      nonce: 39,
      difficulty: 4
    }
  ]
```

```
describe('when the new chain is longer and the chain is invalid', () =>
    beforeEach(() => {
        newChain.chain[2].hash = 'some-fake-hash';
        blockchain.replaceChain(newChain.chain);
    });
    it('does NOT replace the chain', () =>{
        expect(blockchain.chain).toEqual(originalChain);
    });
    it('logs an error', () =>{
        expect(errorMock).toHaveBeenCalled();
    });
```

```
        expect(errorMock).toHaveBeenCalled();
    });
})
```

We provide an image to showcase a log of the newly added block. We can also see that this new block references the hash of the last block.

```
describe('a lastHash reference changed' , () =>{
    it('should return false', () =>{
        // tamper data
        blockchain.chain[2].lastHash = 'tampered-hash';
        expect(Blockchain.isValidChain(blockchain.chain)).toBe(false);
    });
});
```

For testing the validity of the chain, we explored three cases, in which we alter the first block on the Blockchain known as the genesis block. In the other two cases, we alter a hash of a block on the Blockchain. To continue testing for validity, we finally test for alteration of a data field in the Blockchain on a random block and log the results. When using Jest, we can mock a new and existing Blockchain and add new blocks to it before each test run. Here we can observe how we utilize Jest to do this, as well as how our test is described in assessing the validity of the chain after genesis block alteration. We assert the validity to be false as expected. Similarly, we conduct a test describing the scenario of tampering the hash on the Blockchain expecting Jest to log for us that it is an invalid chain. We can do the same when it comes to tampering the data on the blockchain as well. We simply assert the expectation that it will be an invalid chain due to the data being tampered.

```
describe('and the chain contains a block with an invalid field',()=>{
    it('should return false', () =>{
        // tamper data
        blockchain.chain[2].data = 'tampered-data';
        expect(Blockchain.isValidChain(blockchain.chain)).toBe(false);

    });
});
        expect(Blockchain.isValidChain(blockchain.chain)).toBe(false);
    });
});
```

```
describe('when the new chain is longer and the chain is valid', ()=>{
    beforeEach(() => {
        blockchain.replaceChain(newChain.chain);
    });
    it('should replace the chain', () =>{
        expect(blockchain.chain).toEqual(newChain.chain);
    });
    it('should log about the chain replacement' , () =>{
        expect(logMock).toHaveBeenCalled();
    });
});
```

If the data were the same, it would not be tampered, and thus would be a valid chain, so we tested on various bits of data. When it comes to testing the replacing of the chain, we focused on four scenarios, where the incoming chain is longer, and

```
PASS ./blockchain.test.js

Test Suites: 3 passed, 3 total
Tests:       29 passed, 29 total
Snapshots:   0 total
Time:        0.958 s, estimated 2 s
Ran all test suites matching /blockchain/i.
```

valid, the incoming chain is not longer, invalid, or valid, and when the incoming chain is valid and longer. When the incoming chain is not longer, it should not replace the chain. However, when the chain IS longer, it depends on whether it is valid or not.

We test for all these cases showcasing our test cases and our test results. Since our testing included the testing of the chain being valid, we were able to mimic an alteration to the chain by altering the hash. This in turn invalidates the Blockchain, resulting in the test passing as it expects the chain to not be replaced. We finally test for the scenario in which the Blockchain does get replaced and the incoming chain is valid.

Test ID	Actual Result	Pass/Fail	Comments	Type
6	Creates correct block	Pass		White Box
7	Raises difficulty	Pass		WB
8	Lowers difficulty	Pass		WB
9	Returns 1 for adjusted difficulty	Pass		WB
10	Creates block	Pass		WB
11	Invalid chain	Pass	Can't alter genesis block	WB
12	Invalid chain	Pass	Can't alter hash	WB
13	Invalid Chain	Pass	Can't alter chain's data	WB
14	Replaces chain	Pass	Replaces chain when valid	WB
15	Does not replace chain	Pass	Does not replace chain	WB
16	Does not replace chain	Pass	Does not replace chain	WB

Conclusion:

In our testing, we came across what we considered to be bugs in the way the API handles its middleware. This is important because when it comes to middleware, there should be checks for the validation of proper data, as well as authentication if needed and proper error handling. We surprisingly uncovered the implementation of proper error handling when it came to the creation of a block on the Blockchain. When observing the internal mechanisms of the Blockchain and block class, we tested for important functionality that is expected of the software. We were able to test effectively for alterations on the Blockchain and the consequences of such actions. We also were able to see how the Blockchain itself uses its functions to validate the chain or add a new block. Our trust in the system should be a bit more solidified now that we understand how the Blockchain validates and assess the logic before it replaces the chain or confirms its validity of not being altered.

Appendix:

Block Class

```
const { GENESIS_DATA, MINE_RATE } = require('./config');
const cryptoHash = require('./crypto-hash');
const hexToBinary = require('hex-to-binary');

class Block{
     constructor({timestamp, lastHash, hash, data, nonce, difficulty}){
         this.timestamp = timestamp;
         this.lastHash = lastHash;
         this.hash = hash;
         this.data = data;
         this.nonce = nonce;
         this.difficulty = difficulty;
     }

    static genesis(){
        return new this(GENESIS_DATA);

    }

    static mineBlock({lastBlock, data}){
        let hash, timestamp;
        const lastHash = lastBlock.hash;
        let { difficulty } = lastBlock;
        let nonce = 0;
        do{
            nonce++;
            timestamp = Date.now();
            difficulty = Block.adjustDifficulty({originalBlock: lastBlock, timestamp});
            hash = cryptoHash(timestamp, lastHash, data, nonce, difficulty);
        } while(hexToBinary(hash).substring(0, difficulty) !== '0'.repeat(difficulty));
        return new this({
            timestamp,
            lastHash,
            data,
            difficulty,
            nonce,
            hash
        });
    }


    static adjustDifficulty({originalBlock, timestamp}){
        const {difficulty} = originalBlock;
        if(difficulty < 1){return 1;}
        const difference = timestamp - originalBlock.timestamp;
        if(difference > MINE_RATE){
            return difficulty - 1;
        }
        return difficulty + 1;
    }
}
```

Blockchain Class

```
const Block = require('./block');
const cryptoHash = require('./crypto-hash');

class Blockchain {
    constructor(){
        this.chain = [Block.genesis()];
    }

    addBlock({data}){
        const newBlock = Block.mineBlock({
            lastBlock: this.chain[this.chain.length - 1],
            data
        });
        this.chain.push(newBlock);
    }
    replaceChain(chain){
        if(chain.length <= this.chain.length){
            console.error("Incoming chain must be longer");
            return;
        }
        if(!Blockchain.isValidChain(chain)){
            console.error("Incoming chain must be valid");
            return;
        }
        console.log("replacing chain with ", chain);
        this.chain = chain;
    }

    static isValidChain(chain){
        if(JSON.stringify(chain[0])
            !== JSON.stringify(Block.genesis())) return false;

        for(let i = 1; i < chain.length; i++){
            const { timestamp, lastHash, hash, data, nonce, difficulty} = chain[i];
            //destructuring our data from the block
            const actualLastHash = chain[i - 1].hash;
            const lastDiffculty = chain[i - 1].difficulty;

            if(lastHash !== actualLastHash){return false;}

            const validatedHash = cryptoHash(timestamp, lastHash, data, nonce, difficulty);
            if(hash !== validatedHash){return false;}

            if(Math.abs(lastDiffculty - difficulty) > 1){return false;}
        }
        return true;

    }
}

module.exports = Blockchain;
```

Code Coverage Report

All files

100% Statements 66/66 **100%** Branches 16/16 **100%** Functions 9/9 **100%** Lines 61/61

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
block.js		100%	27/27	100%	4/4	100%	4/4	100%	26/26
blockchain.js		100%	29/29	100%	12/12	100%	4/4	100%	25/25
config.js		100%	4/4	100%	0/0	100%	0/0	100%	4/4
crypto-hash.js		100%	6/6	100%	0/0	100%	1/1	100%	6/6

Block test (a)

```
const hexToBinary = require('hex-to-binary');
const Block = require('./block');
const { GENESIS_DATA, MINE_RATE } = require('./config');
const cryptoHash = require('./crypto-hash')
describe('Block', () => {
    const timestamp = 2000;
    const lastHash = 'foo-hash';
    const hash = 'bar-hash';
    const data = 'some-data';
    const nonce = 1;
    const difficulty = 1;
    const block = new Block({timestamp, lastHash, hash, data, nonce, difficulty});
    it('should have timestamp, lastHash, hash,', () => {
        expect(block.timestamp).toEqual(timestamp);
        expect(block.lastHash).toEqual(lastHash);
        expect(block.hash).toEqual(hash);
        expect(block.data).toEqual(data);
        expect(block.nonce).toEqual(nonce);
        expect(block.difficulty).toEqual(difficulty);
    });
    describe('genesis()', () => {
        const genesisBlock = Block.genesis();
        it('returns a Block instance' , () => {
            expect(genesisBlock).toBeInstanceOf(Block);
        });
        it('returns the genesis',()=>{
            expect(genesisBlock).toEqual(GENESIS_DATA);
        })
    });
    describe('mineBlock()', () => {
        const lastBlock = Block.genesis();
        const data = '';
        const minedBlock = Block.mineBlock({lastBlock, data });
        it("return a Block instance", () => {
            expect(minedBlock).toBeInstanceOf(Block);
        });
        it('sets the `lastHash` to be the `hash` of the lastBlock', () => {
            expect(minedBlock.lastHash).toEqual(lastBlock.hash);
            console.log('lastHash', minedBlock.lastHash);
            console.log('hash', lastBlock.hash);
        });
        it('sets the `data`', ()=>{
            expect(minedBlock.data).toEqual(data);
        });
        it('sets the `timestamp` to be the current time', ()=>{
            expect(minedBlock.timestamp).not.toEqual(undefined);
        });

        it('create a SHA-256 `hash` based on the proper inputs', ()=>{
```

Block test (b)

```
        it('create a SHA-256 `hash ` based on the proper inputs', ()=>{
            expect(minedBlock.hash).toEqual(
                cryptoHash(minedBlock.timestamp,
                    minedBlock.nonce,
                    minedBlock.difficulty,
                    lastBlock.hash,
                    data
                    )
            );
        });
        it('sets a `hash` that matches the difficulty criteria', () =>{
            expect(hexToBinary(minedBlock.hash).substring(0, minedBlock.difficulty))
            .toEqual('0'.repeat(minedBlock.difficulty));
        });
        it('adjusts the difficulty', () =>{
            const possibleResults = [lastBlock.difficulty + 1, lastBlock.difficulty - 1];
            expect(possibleResults.includes(minedBlock.difficulty)).toBe(true);
        });
        console.log('minedBlock', minedBlock);
});


    describe('adjustDifficulty()', ()=>{
        it('raises the difficulty for a quickly mined block' , () => {
            expect(Block.adjustDifficulty({
                originalBlock: block,
                timestamp: block.timestamp + MINE_RATE - 100
            })).toEqual(block.difficulty + 1);
        });
        it('lowers the difficulty for a slowly mined block' , () => {
            expect(Block.adjustDifficulty({
                originalBlock: block,
                timestamp: block.timestamp + MINE_RATE + 100
            })).toEqual(block.difficulty - 1);
        });

        it('has a lower limit of 1', () => {
            block.difficulty = 0;
            expect(Block.adjustDifficulty({
                originalBlock: block
            })).toEqual(1);
        });
        console.log('block', block);

    })
```

Blockchain test (a)

```
const Blockchain = require('./blockchain');
const Block = require('./block');
const cryptoHash = require('./crypto-hash');

describe('Blockchain', () => {
    let blockchain, newChain, originalChain;

    beforeEach(() => {
        blockchain = new Blockchain();
        newChain = new Blockchain();
        originalChain = blockchain.chain;
    });

    it('should contain a `chain` Array instance', () =>{
        expect(blockchain.chain).toBeInstanceOf(Array);
    });

    it('should start with the genesis block', () =>{
        expect(blockchain.chain[0]).toEqual(Block.genesis());

    });

    it('adds a new block to the chain' , () =>{
        const newData = 'hey test data';
        blockchain.addBlock({data: newData});
        expect(blockchain.chain[blockchain.chain.length - 1].data).toEqual(newData);
        //"last" hash" is the hash of the previous block
        expect(blockchain.chain[blockchain.chain.length - 1].lastHash).toEqual(blockchain.chain[blockchain.chain.length - 2].hash)
    });


    describe('isValidChain()', () =>{
        beforeEach(() => {
            blockchain.addBlock({data: 'value'});
            blockchain.addBlock({data: 'value2'});
            blockchain.addBlock({data: 'value3'});
        });
        describe('when the chain does not start with the genesis block', () =>{
            it('should return false', () =>{
               blockchain.chain[0] = {data: 'data fake'};
               expect(Blockchain.isValidChain(blockchain.chain)).toBe(false);
            });
        });

        describe('when the chain starts with the genesis block', () =>{
                describe('a lastHash reference changed' , () =>{
                    it('should return false', () =>{
                        // tamper data
                        blockchain.chain[2].lastHash = 'tampered-hash';
```

Blockchain test (b)

```
describe('when the chain starts with the genesis block', () =>{
    describe('a lastHash reference changed' , () =>{
        it('should return false', () =>{
            // tamper data
            blockchain.chain[2].lastHash = 'tampered-hash';
            expect(Blockchain.isValidChain(blockchain.chain)).toBe(false);
        });
    });
});

describe('and the chain contains a block with an invalid field',()=>{
    it('should return false', () =>{
        // tamper data
        blockchain.chain[2].data = 'tampered-data';
        expect(Blockchain.isValidChain(blockchain.chain)).toBe(false);

    });
});

describe('and the chain does not contain a any invalid data',()=>{
    it('should return true', () =>{


        expect(Blockchain.isValidChain(blockchain.chain)).toBe(true);
    });
});

describe('and the chain contains a block with a jumped difficulty', () =>{
    it('should return false', () => {
        const lastBlock = blockchain.chain[blockchain.chain.length - 1];
        const lastHash = lastBlock.hash;
        const timestamp = Date.now();
        const nonce = 0;
        const data = [];
        const difficulty = lastBlock.difficulty - 3;

        const hash = cryptoHash(timestamp, lastHash, data, nonce, difficulty);

        const badBlock = new Block({
            timestamp,
            lastHash,
            hash,
```

Blockchain test (c)

```
    describe('and the chain contains a block with a jumped difficulty', () =>{
        it('should return false', () => {
            const lastBlock = blockchain.chain[blockchain.chain.length - 1];
            const lastHash = lastBlock.hash;
            const timestamp = Date.now();
            const nonce = 0;
            const data = [];
            const difficulty = lastBlock.difficulty - 3;

            const hash = cryptoHash(timestamp, lastHash, data, nonce, difficulty);

            const badBlock = new Block({
                timestamp,
                lastHash,
                hash,
                data,
                nonce,
                difficulty

            });
            blockchain.chain.push(badBlock);
            expect(Blockchain.isValidChain(blockchain.chain)).toBe(false);

        });
    });

});
```

Blockchain test (d)

```
describe('replaceChain()', () =>{
    let errorMock, logMock;
    beforeEach(() => {
        errorMock = jest.fn();
        logMock = jest.fn();

        global.console.error = errorMock;
        global.console.log = logMock;
    })

    describe('when the new chain is not longer ', () =>{
        beforeEach(() => {
            newChain.chain[0] = {new: 'chainnnn'};
            blockchain.replaceChain(newChain.chain);
        })
        it('does NOT replace the chain', () =>{
            expect(blockchain.chain).toEqual(originalChain);
        });

        it('logs an error', () =>{
            expect(errorMock).toHaveBeenCalled();
        });
    })

    describe('when the new chain is longer ', () =>{

        beforeEach(() => {
            newChain.addBlock({data: 'new data'});
            newChain.addBlock({data: 'new data1'});
            newChain.addBlock({data: 'new data2'});
        })

        describe('when the new chain is longer and the chain is invalid', () => {
            beforeEach(() => {
                newChain.chain[2].hash = 'some-fake-hash';
                blockchain.replaceChain(newChain.chain);
            });
            it('does NOT replace the chain', () =>{
                expect(blockchain.chain).toEqual(originalChain);
            });
            it('logs an error', () =>{
                expect(errorMock).toHaveBeenCalled();
            });

        });
```

Blockchain test (d)

```
});

describe('when the new chain is longer and the chain is valid', ()=>{
    beforeEach(() => {
        blockchain.replaceChain(newChain.chain);
    });
    it('should replace the chain', () =>{
        expect(blockchain.chain).toEqual(newChain.chain);
    });
    it('should log about the chain replacement' , () =>{
        expect(logMock).toHaveBeenCalled();
    });
});
```

References:

[1] Conway, Luke. "Blockchain Explained." *Investopedia*, Investopedia, 5 Dec. 2021, https://www.investopedia.com/terms/b/blockchain.asp

[2] "Using Matchers · Jest." *Jest Blog RSS*, https://jestjs.io/docs/using-matchers