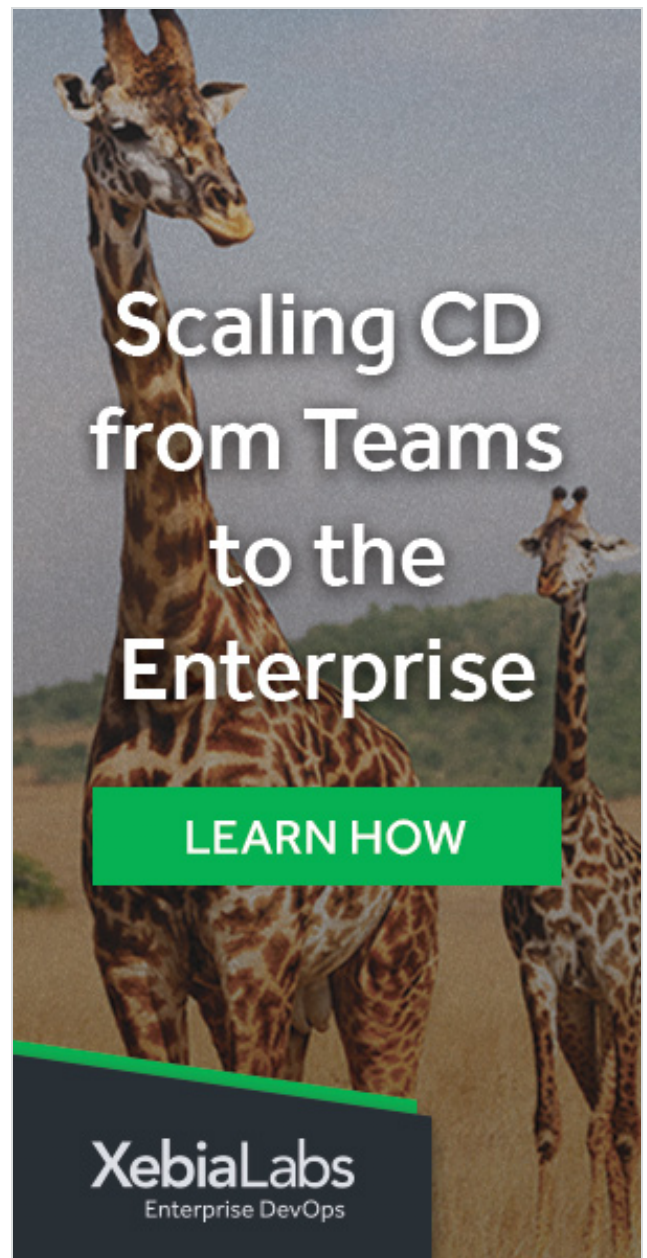




Discover the 6 Most Common Performance Testing Mistakes in the 2018 Guide to Performance


Download Guide ▶



Blockchain Implementation With

Blockchain Implementation With Java Code

Let's take a look at a possible blockchain implementation using Java. We build up from first principles and develop some code to help show how it all fits together.

by David Pitt  MVB · Apr. 12, 18 · Java Zone · Tutorial

Take 60 minutes to understand the Power of the Actor Model with "Designing Reactive Systems: The Role Of Actors In Distributed Architecture". Brought to you in partnership with Lightbend.

Bitcoin is hot — *and what an understatement that is*. While the future of cryptocurrency is somewhat uncertain, blockchain — the technology used to drive Bitcoin — is also very popular.

Blockchain has an almost endless application scope. It also arguably has the potential to disrupt enterprise automation. There is a lot of information available covering what and how blockchain works. We have a free whitepaper that goes into blockchain technology (no registration required).

This article will focus on the blockchain architecture; particularly, demonstrating how the "immutable, append-only" distributed ledger works with simplistic code examples.

As developers, seeing things in code can be much more useful in understanding how it works when compared to simply reading technical articles. At least that's the case for me. So, let's get started!

Blockchain in a Nutshell

Let's first give a quick summary of blockchain. A block contains some header information and a set or block of transactions of any type of data. The chain starts with a first (Genesis) block. As transactions are added/appended, new blocks are created based on how many transactions can be stored within a block.

When a block threshold size is exceeded, then a new block of transactions is created. The new block is linked to the previous block, hence the term blockchain.

Immutability

Blockchains are immutable because an SHA-256 hash is computed for transactions. A block's contents are also hashed which provide a unique identifier. Moreover, the hash from the linked, previous block is

also stored and hashed in the block header.

This is why trying to tamper with a blockchain block is basically impossible, at least with current computing power. Here's a partial Java class definition showing the properties of the block.

```

1  ...
2  public class Block<T> extends Tx<T> {
3  public long timeStamp;
4  private int index;
5  private List<T> transactions = new ArrayList<T>();
6  private String hash;
7  private String previousHash;
8  private String merkleRoot;
9  private String nonce = "0000";
10
11 // caches Transaction SHA256 hashes
12     public Map<String,T> map = new HashMap<String,T>();
13 ...

```

Notice the injected generic type is of type `T`. This allows transaction data to vary. Also, the `previousHash` property will reference the previous block's hash. The `merkleRoot` and `nonce` properties will be described in a bit.

Block Hash

Each block can compute a block hash. This is essentially a hash of all the block's properties concatenated together, including the previous block's hash and a SHA-256 hash computed from that.

Here is the method defined in the `Block.java` class that computes the hash.

```

1  ...
2  public void computeHash() {
3      Gson parser = new Gson(); // probably should cache this instance
4      String serializedData = parser.toJson(transactions);
5      setHash(SHA256.generateHash(timeStamp + index + merkleRoot + serializedData +
6      nonce + previousHash));
7  }
8  ...

```

The block transactions are serialized to a JSON string so it can be appended to the block properties before hashing.

The Chain

The blockchain manages blocks by accepting transactions. When a predetermined threshold has been reached, then a block is created. Here is a `SimpleBlockchain.java` partial implementation:

```

1  ...
2  ...
3  public class SimpleBlockchain<T extends Tx> {
4  public static final int BLOCK_SIZE = 10;
5  public List<Block<T>> chain = new ArrayList<Block<T>>();
6
7  public SimpleBlockchain() {
8  // create genesis block
9  chain.add(newBlock());
10 }
11
12 ...

```

Notice that the chain property holds a list of Blocks typed with a `Tx` type. Also, the `no arg` constructor creates an initial "genesis" block when the chain is created. Here is the source for the `newBlock()` method.

```

1  ...
2  public Block<T> newBlock() {
3  int count = chain.size();
4  String previousHash = "root";
5
6  if (count > 0)
7  previousHash = blockChainHash();
8
9  Block<T> block = new Block<T>();
10
11 block.setTimeStamp(System.currentTimeMillis());
12 block.setIndex(count);
13 block.setPreviousHash(previousHash);
14 return block;
15 }
16 ...

```

This new block method will create a new block instance, seed appropriate values, and assign the previous block's hash (which will be the hash of the head of the chain). It will then return the block.

Blocks can be validated before being added to the chain by comparing the new block's previous hash to the last block (head) of the chain to make sure they match. Here's a `SimpleBlockchain.java` method depicting this.

```

1  ....
2  public void addAndValidateBlock(Block<T> block) {
3
4  // compare previous block hash, add if valid
5  Block<T> current = block;
6  for (int i = chain.size() - 1; i >= 0; i--) {
7  Block<T> b = chain.get(i);
8  if (b.getHash().equals(current.getPreviousHash())) {

```

```

8      if (b.gethash().equals(current.getPrevioushash())) {
9          current = b;
10     } else {
11
12     throw new RuntimeException("Block Invalid");
13     }
14
15     }
16
17     this.chain.add(block);
18     }
19     ...

```

The entire blockchain is validated by the looping-over of the chain to ensure a block's hash still matches the previous block's hash.

Here is the `SimpleBlockChain.java validate()` method implementation.

```

1     ...
2     public boolean validate() {
3
4     String previousHash = null;
5     for (Block<T> block : chain) {
6     String currentHash = block.getHash();
7     if (!currentHash.equals(previousHash)) {
8     return false;
9     }
10
11     previousHash = currentHash;
12
13     }
14
15     return true;
16
17     }
18     ...

```

You can see that trying to fudge transaction data or any other property in any way is very difficult. And, as the chain grows, it continues to get very, very, very difficult, essentially impossible — that is, until quantum computers are available!

Adding Transactions

Another significant technical point of blockchain technology is that it is distributed. The fact that they are append-only helps in duplicating the blockchain across nodes participating in the blockchain network. Nodes typically communicate in a peer-to-peer fashion, as is the case with Bitcoin, but it does not have to be this way. Other blockchain implementations use a decentralized approach, like using APIs via HTTP. However, that is a topic for another article.

...as the first step, that is a topic for another article.

Transactions can represent just about anything. A transaction could contain code to execute (i.e Smart Contract) or store and append information about some kind of business transaction.

Smart contract: Computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract.

In the case of Bitcoin, a transaction contains an amount from an owner's account and amount(s) to other accounts (e.g. transferring Bitcoin amounts between accounts). The transaction also includes public keys and account IDs within it, so transferring is done securely. But that's Bitcoin-specific.

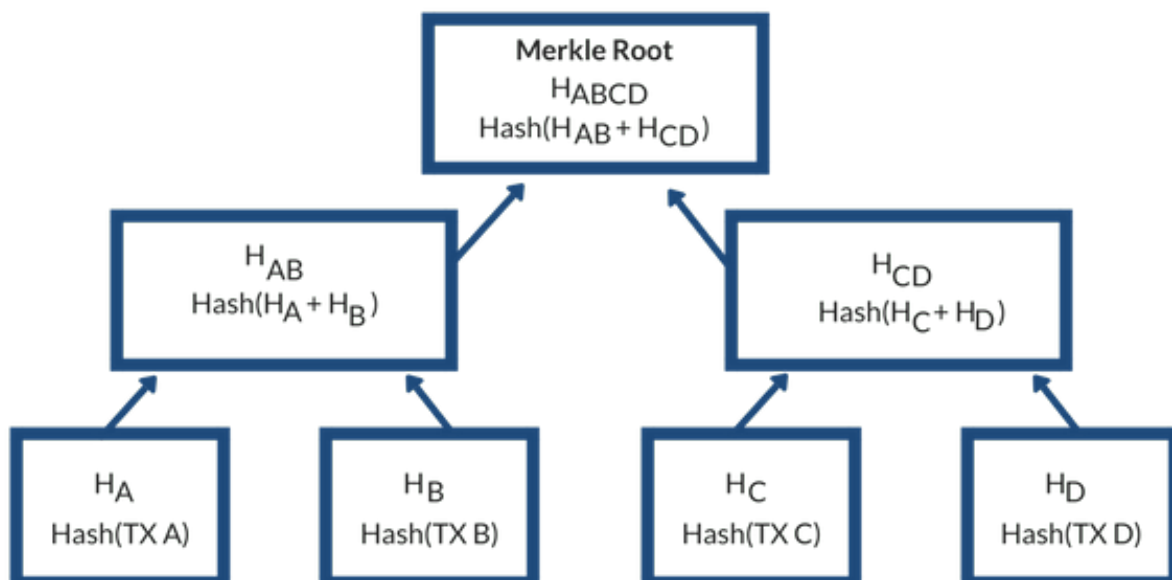
Transactions are added to a network and pooled; they are not in a block or the chain itself.

This is where a blockchain **consensus mechanism** comes into play. There are a number of proven consensus algorithm and patterns beyond the scope of this article.

Mining is a consensus mechanism that Bitcoin blockchains use. That is the type of consensus discussed further down this article. The consensus mechanism gathers transactions, builds a block with them, and then adds the block to the chain. The chain then validates the new block of transactions before adding to the chain.

Merkle Trees

Transactions are hashed and added to the block. A Merkle Tree data structure is created to compute a Merkle Root hash. Each block will store the root of the Merkle tree, which is a balanced binary tree of hashes where interior nodes are hashes of the two child hashes, all the way up to the root hash, which is the Merkle Root.



This tree is used to validate the block transactions. If a single bit of information is changed in any transaction, the Merkle Root will be invalid. Also, they can help with transmitting blocks in a distributed fashion, since the structure allows only a single branch of transaction hashes required to add and validate the entire block of transactions.

Here's the method in the `Block.java` class that creates a Merkle Tree out of the transaction list.

```
1  ...
2  public List<String> merkleTree() {
3  ArrayList<String> tree = new ArrayList<>();
4  // Start by adding all the hashes of the transactions as leaves of the
5  // tree.
6  for (T t : transactions) {
7  tree.add(t.hash());
8  }
9  int levelOffset = 0; // Offset in the list where the currently processed
10 // level starts.
11 // Step through each level, stopping when we reach the root (levelSize
12 // == 1).
13   for (int levelSize = transactions.size(); levelSize > 1; levelSize = (levelSize + 1
14 ) / 2) {
15 // For each pair of nodes on that level:
16 for (int left = 0; left < levelSize; left += 2) {
17 // The right hand node can be the same as the left hand, in the
18 // case where we don't have enough
19 // transactions.
20 int right = Math.min(left + 1, levelSize - 1);
21 String tleft = tree.get(levelOffset + left);
22 String tright = tree.get(levelOffset + right);
23 tree.add(SHA256.generateHash(tleft + tright));
24 }
25 // Move to the next level.
26 levelOffset += levelSize;
27 }
28 return tree;
29 }
30 ...
```

This method is used to compute a Merkle Tree root for the block. The companion project has a Merkle Tree unit test that attempts to add a transaction to a block and verify that the Merkle Roots have changed. Here is the source code for the unit test.

```
1  ...
2  @Test
3  public void merkleTreeTest() {
4
```

```

5 // create chain, add transaction
6
7 SimpleBlockchain<Transaction> chain1 = new SimpleBlockchain<Transaction>();
8
9 chain1.add(new Transaction("A")).add(new Transaction("B")).add(new Transaction("C"))
10 ).add(new Transaction("D"));
11
12 // get a block in chain
13 Block<Transaction> block = chain1.getHead();
14
15 System.out.println("Merkle Hash tree :" + block.merkleTree());
16
17 // get a transaction from block
18 Transaction tx = block.getTransactions().get(0);
19
20 // see if block transactions are valid, they should be
21 block.transasctionsValid();
22 assertTrue(block.transasctionsValid());
23
24 // mutate the data of a transaction
25 tx.setValue("Z");
26
27 // block should no longer be valid, blocks MerkleRoot does not match computed merkl
28 e tree of transactions
29 assertFalse(block.transasctionsValid());
30
31 }
32 ...

```

This unit test emulates validating transactions, then changing a transaction in a block outside of the consensus mechanism, e.g. if someone tries to change transaction data.

Remember, blockchains are append-only, and as the blockchain data structure is shared between nodes, block data structure (including the Merkle Root) are hashed and connected to other blocks. All nodes can validate new blocks and existing blocks can be easily proved as valid. So, a miner trying to add a bogus block or a node attempting to adjust older transactions are effectively not possible before the sun grows to a supernova and gives all a really nice tan.

Mining Proof of Work

The process of combining transactions in into a block, then submitting it for validation by members of the chain, is referred to as "mining" in the Bitcoin world.

More generally, in blockchain speak, this is called consensus. There are different types of proven distributed consensus algorithms. Which mechanism to use is based upon whether you have a public or

permissioned blockchain. Our white paper describes this more in depth, but this article is focusing on the blockchain mechanics, so this example we will apply a proof-of-work consensus mechanism.

So, mining nodes will listen for transactions being executed by the blockchain and will perform a simple mathematical puzzle. This puzzle produces block hash with a predetermined set of leading zeros using a nonce value that is changed on every iteration until the leading zero hash is found.

The example Java project has a `Miner.java` class with a `proofOfWork(Block block)` method implementation, as shown below.

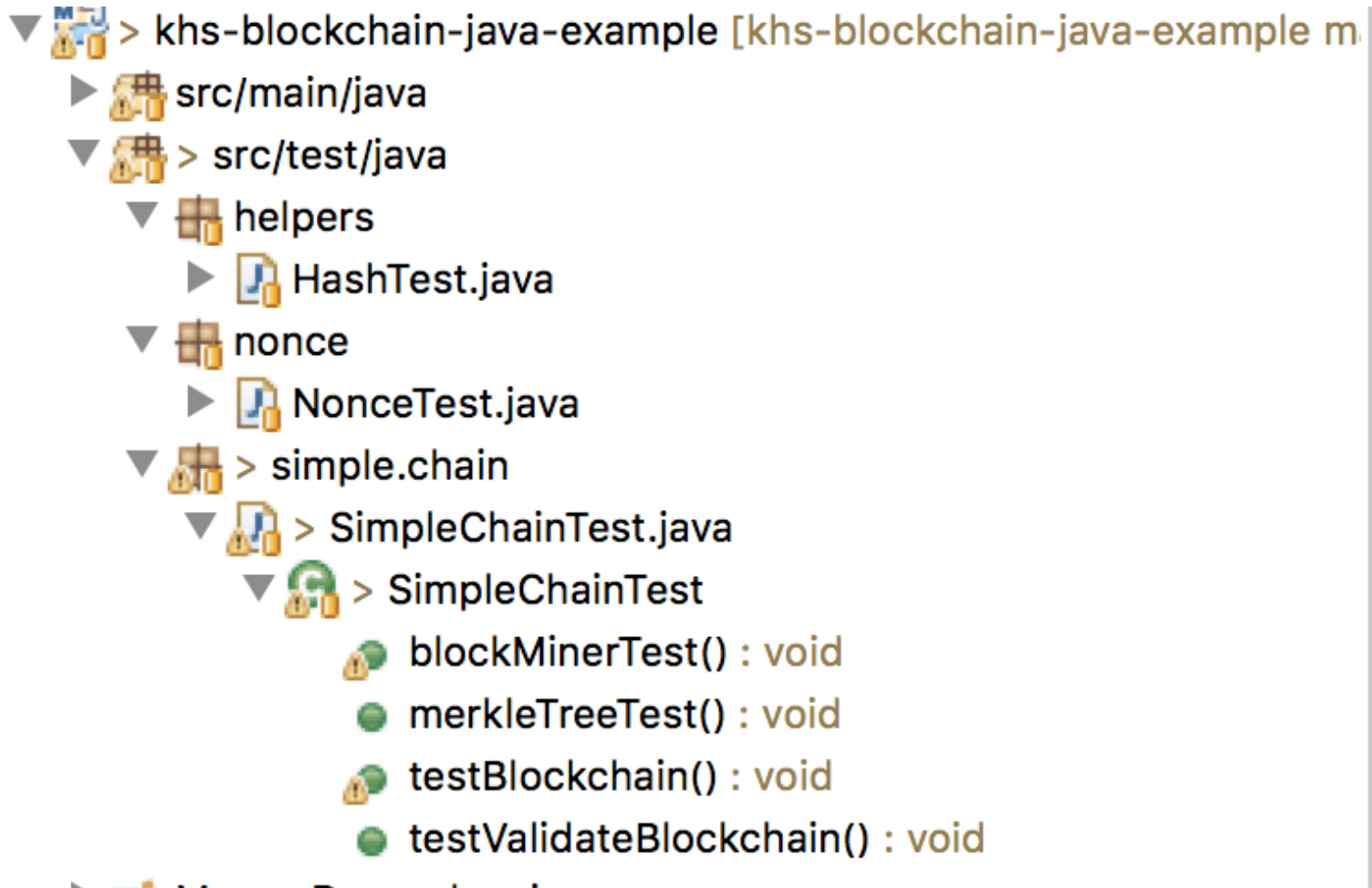
```
1  private String proofOfWork(Block block) {
2
3  String nonceKey = block.getNonce();
4  long nonce = 0;
5  boolean nonceFound = false;
6  String nonceHash = "";
7
8  Gson parser = new Gson();
9  String serializedData = parser.toJson(transactionPool);
10 String message = block.getTimestamp() + block.getIndex() + block.getMerkleRoot() +
11 serializedData
12 + block.getPreviousHash();
13 while (!nonceFound) {
14
15 nonceHash = SHA256.generateHash(message + nonce);
16 nonceFound = nonceHash.substring(0, nonceKey.length()).equals(nonceKey);
17 nonce++;
18
19 }
20
21 return nonceHash;
22
23 }
```

Again, this is simplified, but the miner implementation will perform a proof-of-work hash for the block once a certain number of transactions have been received. The algorithm simply loops and creates an SHA-256 hash of the block until the leading number hash is produced.

This can take a lot of time, which is why specific GPU microprocessors have been implemented to perform and solve this problem as fast as possible.

Unit Tests

You can see all these concepts pulled together with the Java example project's JUnit tests available on [GitHub](#).



Give this a run. It will let you check out how this simple blockchain works.

Also, if you are a C#'er reading, this (we won't tell anyone), we also have these same examples written in C#. Here is the link to the example C# blockchain implementation.

Final Thoughts

Hopefully, this post has provided you enough interest and insight to keep researching blockchain technology.

All of the examples introduced in this article are used in our in-depth blockchain white paper (no registration required to read). These same examples are in more detail in the white paper.

Also, if you want to see a full blockchain implementation in Java, here's a link to the open-source BitcoinJ project. You'll see these concepts in action in a real production implementation.

If so, next recommended learning steps are to check out a more production-based open-source blockchain framework. A good example is HyperLedger Fabric. That will be the subject of my next article — stay tuned!

Learn how the Actor model provides a simple but powerful way to design and implement reactive applications that can distribute work across clusters of cores and servers. Brought to you in partnership with Lightbend.

Like This Article? Read More From DZone



**Hyperledger's Brian Behlendorf:
Blockchain for Business**



The Bitcoin Protocol: How It Works




Why Do We Need Blockchain?



**Free DZone Refcard
Getting Started With Kotlin**

Topics: [JAVA](#) , [BLOCKCHAIN](#) , [BITCOIN](#) , [DISTRIBUTED LEDGER](#) , [TUTORIAL](#)

Published at DZone with permission of David Pitt , DZone MVB. [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

SUBSCRIBE

Get the best of Java in your inbox.

Stay updated with DZone's bi-weekly Java Newsletter. [SEE AN EXAMPLE](#)

Java Partner Resources

[Migrating to Microservice Databases](#)

[Red Hat Developer Program](#)



[Single-Page App \(SPA\) Security with Spring Boot and OAuth](#)

[Okta](#)



[Deep insight into your code with IntelliJ IDEA.](#)

[JetBrains](#)



[Advanced Linux Commands \[Cheat Sheet\]](#)

Red Hat Developer Program

