Blockchain

Tags

Introduction

Blockchain is a distributed database that maintains a continuously growing list of records called blocks. Each block contains a timestamp and a link to the previous block. This allows the database to be efficiently updated without the need for a central authority. Because of its decentralized nature and cryptographic security, blockchain is used to facilitate secure and transparent transactions. It is the technology behind cryptocurrencies like Bitcoin.

Complete courses

- https://www.blocktrain.info/course

The history of blockchain

The history of blockchain technology dates back to the creation of Bitcoin in 2009. Bitcoin was the first decentralized cryptocurrency, and it was created by an anonymous individual or group known as Satoshi Nakamoto. The key innovation of Bitcoin was the use of a decentralized, distributed ledger to record and validate transactions, which eliminated the need for a central authority to maintain and update the ledger. This ledger, known as the blockchain, was secured using cryptographic techniques, which ensured that it was tamper-resistant and transparent.

Since the creation of Bitcoin, blockchain technology has evolved and expanded to include many different applications beyond cryptocurrency. In 2013, Vitalik Buterin developed the Ethereum platform, which introduced the concept of smart contracts and expanded the capabilities of blockchain technology. Today, there are many different blockchain platforms and thousands of decentralized applications being built on top of them.

The history of blockchain technology is still unfolding, and it is likely that the technology will continue to evolve and expand in the coming years. It has the potential to disrupt many industries and enable new forms of economic activity.

Cryptography

Cryptography is a key technology that underpins the security of blockchain systems. It is used to secure the transactions and data on the blockchain, ensuring that they cannot be tampered with or modified without being detected.

In a blockchain, cryptography is used in several different ways, including the following:

1. Hashing: Hashing is the process of generating a fixed-size, unique output from an input of any size. In a blockchain, hashing is used to generate a unique "fingerprint" for each block of transactions, known as a block hash. This allows the blockchain to efficiently store and verify the integrity of the transactions.
2. Digital signatures: Digital signatures are a cryptographic technique that allows the sender of a message to prove that they are the true owner of the message. In a blockchain, digital signatures are used to sign transactions, proving that they are authorized by the owner of the associated cryptocurrency.

3. Public-key cryptography: Public-key cryptography is a cryptographic technique that uses a pair of keys, one public and one private, to secure communications. In a blockchain, public-key cryptography is used to generate addresses for users and to enable them to securely receive and send transactions.

These are just a few examples of the ways that cryptography is used in blockchain technology. Cryptography is a complex and constantly evolving field, and it is essential for the security and functionality of blockchain systems.

Here is an example of code that demonstrates some of the cryptographic techniques used in blockchain technology:

```
// define a function to compute the hash of a block
function hash(block) {
  // compute the SHA-256 hash of the block data
  return sha256(block.data);
}

// define a function to generate a digital signature for a message
function sign(message, privateKey) {
  // use the private key to sign the message using the ECDSA algorithm
  return ecdsa.sign(message, privateKey);
}

// define a function to verify a digital signature
function verify(message, signature, publicKey) {
  // use the public key to verify the signature using the ECDSA algorithm
  return ecdsa.verify(message, signature, publicKey);
}

// define a function to generate a pair of public and private keys
function generateKeyPair() {
  // use the secp256k1 curve to generate a pair of keys
  return ecdsa.generateKeyPair(curve: "secp256k1");
}
```

This is a very simple example of code that demonstrates some of the cryptographic techniques used in blockchain technology. It is not intended to be used as production code.

Explore the blockchain

To explore different blockchain data, you can use a blockchain explorer. A blockchain explorer is a web-based tool that allows you to view and search the contents of a

blockchain. It provides detailed information about blocks, transactions, and addresses on the blockchain. To use a blockchain explorer, you simply need to enter a block number, transaction hash, or address into the search bar, and the explorer will retrieve and display the relevant information. Some popular blockchain explorers for Bitcoin include Blockchain.com and Blockstream.info. These tools can be accessed through a web browser, and you do not need any special software or technical knowledge to use them.

Some examples of blockchain explorers for the Bitcoin blockchain include Blockchain.com and Blockstream.info. Both of these explorers allow you to view and search the contents of the Bitcoin blockchain, including blocks, transactions, and addresses. Other examples of blockchain explorers include Etherscan for the Ethereum blockchain, Binance Chain Explorer for the Binance Chain, and Zchain for the Zcash blockchain. These tools are all available through web browsers and provide similar functionality for exploring the contents of their respective blockchains.

Ethereum introduction

Ethereum is a decentralized, open-source blockchain platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference. These applications are run on a custom-built blockchain, which is a decentralized database that runs on a network of computers. Ethereum is the second-largest blockchain platform by market capitalization, after Bitcoin. It was developed by Vitalik Buterin in 2013 and has since become a popular platform for creating and running decentralized applications.

Smart contracts

Smart contracts are self-executing contracts with the terms of the agreement between buyer and seller being directly written into lines of code. The code and the agreements contained therein exist across a distributed, decentralized blockchain network. Smart contracts allow for the automation of digital contracts and have the potential to revolutionize the way that transactions are executed and recorded, enabling new forms of economic activity and improving the efficiency and security of existing systems.

Here is an example of a simple smart contract written in Solidity, a programming language for writing smart contracts on the Ethereum blockchain:

```
pragma solidity ^0.8.0;

// define the contract
contract SimpleContract {
  // define a state variable to store a value
  uint256 value;

  // define a function to set the value of the state variable
  function setValue(uint256 newValue) public {
    value = newValue;
  }

  // define a function to get the value of the state variable
  function getValue() public view returns (uint256) {
    return value;
  }
}
```

This is a very simple example of a smart contract. It defines a contract called `SimpleContract` that has two functions: `setValue` and `getValue`. The `setValue` function allows the contract to be updated by setting the value of the `value` state variable. The `getValue` function allows the contract to be queried by returning the value of the `value` state variable. This contract is not complete and is not intended to be used in production. It is intended only to illustrate the basic structure of a Solidity smart contract.

L1 blockchains

L1 blockchains, also known as "layer 1" blockchains, are the foundational layer of a blockchain platform. They are the underlying infrastructure that supports the platform and enables it to function. L1 blockchains typically consist of a network of nodes that maintain a shared, decentralized ledger of transactions and a consensus mechanism to ensure the integrity and security of the ledger. L1 blockchains are often the first layer of a multi-layer blockchain platform, with additional layers built on top to provide additional features and functionality. Examples of L1 blockchains include the Bitcoin and Ethereum networks.

As I mentioned earlier, some examples of L1 blockchains include the Bitcoin and Ethereum networks. Both of these networks are decentralized, open-source platforms that run on a network of nodes and use a consensus mechanism to maintain a shared ledger of transactions. Other examples of L1 blockchains include Litecoin, which is based on the Bitcoin network and provides faster transaction processing times, and

Ripple, which is a payment network for financial institutions. These are just a few examples of L1 blockchains; there are many others that are in use today.

Building an L1 blockchain requires a significant amount of technical expertise and resources. It typically involves the following steps:

1. Determine the goals and requirements for your blockchain, including the types of transactions it will support, the consensus mechanism it will use, and any other features or functionality it will provide.
2. Design the architecture for your blockchain, including the data structure for the blockchain ledger and the network architecture for the nodes that will maintain it.
3. Implement the core components of your blockchain, including the consensus mechanism, the ledger, and the network architecture.
4. Test your blockchain to ensure that it functions correctly and is secure against potential attacks.
5. Launch your blockchain and recruit a network of nodes to support it.

This is a high-level overview of the process for building an L1 blockchain. There are many technical details and challenges involved in each step, and it is recommended that you have expertise in cryptography, distributed systems, and software engineering to successfully build an L1 blockchain.

Here is an example of building an L1 blockchain to support a simple cryptocurrency:

1. Determine the goals and requirements for the blockchain. In this case, the blockchain will support transactions of a fictional cryptocurrency called "CoinX." It will use a proof-of-work consensus mechanism, and it will have a limited supply of 10 million coins.
2. Design the architecture for the blockchain. The blockchain ledger will consist of blocks of transactions, each of which will have a timestamp, a link to the previous block, and a list of transactions. The network architecture will consist of nodes that will communicate with each other using a gossip protocol to propagate new transactions and blocks.

3. Implement the core components of the blockchain. This involves writing the code for the consensus mechanism, the ledger, and the network architecture. The consensus mechanism will require miners to solve cryptographic puzzles to create new blocks, and the ledger will use a data structure called a Merkle tree to efficiently store and verify transactions.
4. Test the blockchain to ensure that it functions correctly and is secure. This involves running simulations and stress tests to verify that the consensus mechanism works as intended, the ledger is properly maintained, and the network can withstand various attacks.
5. Launch the blockchain and recruit a network of nodes. This involves deploying the blockchain code to a network of computers, which will run the code and participate in the consensus process. The network will need to reach a consensus on the initial state of the ledger, and then it will be ready to process transactions.

This is just one possible example of building an L1 blockchain. The specific steps and details will vary depending on the goals and requirements of the blockchain. It is a complex process that requires a deep understanding of cryptography, distributed systems, and software engineering.

L2 blockchain

An L2 blockchain, or "layer 2 blockchain," is a blockchain that is built on top of another blockchain, known as the "layer 1" or "base" blockchain. L2 blockchains are designed to improve the scalability, performance, and security of the underlying blockchain by offloading some of the workload to the L2 blockchain. This can help to reduce transaction fees, increase transaction throughput, and improve the user experience. Some examples of L2 blockchains include the Lightning Network on top of the Bitcoin blockchain, and the Plasma framework on top of the Ethereum blockchain.

Sharding

Sharding is a technique used to improve the scalability and performance of a blockchain. In a sharded blockchain, the network is divided into multiple "shards," or partitions, where each shard processes a subset of the transactions and maintains its own portion of the ledger. This allows the network to process multiple transactions in

parallel, which can increase the overall transaction throughput and reduce transaction fees.

Here is an example of the steps to build a blockchain with sharding:

1. Determine the number of shards and the shard size. The number of shards should be determined based on the desired level of scalability and performance, while the shard size should be determined based on the desired level of decentralization and security.
2. Implement the sharding mechanism. This involves designing and implementing the algorithms and data structures for dividing the network into shards, assigning transactions to shards, and maintaining the consistency and integrity of the ledger across all shards.
3. Implement the consensus mechanism. In a sharded blockchain, the consensus mechanism must be designed to ensure that the transactions within each shard are processed and validated in a decentralized and secure manner, and that the ledgers across all shards remain consistent.
4. Implement the transaction processing and validation logic. This involves designing and implementing the algorithms and data structures for processing and validating transactions within each shard, as well as the mechanisms for communicating and coordinating with other shards.
5. Implement the APIs and interfaces for interacting with the blockchain. This includes designing and implementing the APIs and interfaces for submitting transactions, querying the ledger, and accessing the data and functionality of the blockchain.

This is a high-level overview of the steps to build a blockchain with sharding. The specific implementation will depend on the design and requirements of the blockchain.

Here is an example of the code for a sharded blockchain in pseudocode:

```
// define the blockchain class
class Blockchain {
  // define the number of shards
  numShards = 10;

  // define the array for the shards
  Shard[] shards = new Shard[numShards];

  // define the function to initialize the shards
```

```
  function initialize() {
    // initialize each shard with its shard ID
    for (int i = 0; i < numShards; i++) {
      shards[i] = new Shard(i);
    }
  }

  // define the function to process a transaction
  function processTransaction(Transaction tx) {
    // get the shard ID for the transaction
    int shardId = getShardId(tx);

    // process the transaction in the appropriate shard
    shards[shardId].processTransaction(tx);
  }

  // define the function to get the shard ID for a transaction
  function getShardId(Transaction tx) {
    // compute the shard ID based on the transaction data
    // (this is a simplified example and the specific implementation
    // will depend on the design and requirements of the blockchain)
    return tx.sender % numShards;
  }
}
```

Designing the architecture of the blockchain

When designing the architecture for a blockchain, there are several key components to consider, including the data structure for the ledger, the network architecture for the nodes, and the consensus mechanism.

The data structure for the ledger will determine how transactions are stored and organized within the blockchain. In a simple design, the ledger can consist of a series of blocks, where each block contains a timestamp, a link to the previous block, and a list of transactions. More advanced designs may use data structures such as Merkle trees or directed acyclic graphs to improve the efficiency and security of the ledger.

The network architecture for the nodes will determine how the nodes communicate with each other and maintain the ledger. In a simple design, the nodes can use a gossip protocol to propagate new transactions and blocks to each other. More advanced designs may use different communication protocols or incorporate mechanisms such as sharding to improve scalability and performance.

The consensus mechanism will determine how the nodes reach agreement on the state of the ledger and prevent conflicting transactions from being included in the blockchain. There are many different consensus mechanisms, including proof of work, proof of stake, and practical Byzantine fault tolerance. The choice of consensus mechanism will depend on the goals and requirements of the blockchain.

These are just a few examples of the key components to consider when designing the architecture for a blockchain. The specific design will depend on the goals and requirements of the blockchain. It is a complex process that requires a deep understanding of distributed systems and cryptography.

Here is an example of code for the design of a simple blockchain in pseudocode:

```
// define the data structure for the ledger
// the ledger will consist of a series of blocks,
// where each block contains a timestamp, a link to the previous block,
// and a list of transactions
class Block {
  timestamp;
  previousBlockHash;
  transactions;
}

// define the network architecture for the nodes
// the nodes will communicate with each other using a gossip protocol
// to propagate new transactions and blocks
class Node {
  // method to broadcast a message to the network
  broadcast(message) {
    for each otherNode in nodes:
      send(message, otherNode);
  }

  // method to receive a message from the network
  receive(message) {
    if (message.type == "transaction") {
      // if the message is a transaction, add it to the list of valid transactions
      transactions.add(message.transaction);
    } else if (message.type == "block") {
      // if the message is a block, verify it and add it to the local copy of the ledger
      if (verify(message.block)) {
        addToLedger(message.block);
      }
    }
  }
}
```

```
// define the consensus mechanism
// the nodes will use proof of work to reach consensus on the next block of transactions
function proofOfWork(block) {
  // compute a cryptographic hash of the block
  blockHash = hash(block);

  // iterate until a valid block hash is found
  while (true) {
    // add a nonce to the block and compute the new block hash
    block.nonce = random();
    blockHash = hash(block);

    // if the block hash meets the difficulty target, return the block
    if (blockHash < TARGET) {
      return block;
    }
  }
}

// define a function to select the next block of transactions
function selectNextBlock() {
  // create a candidate block containing a subset of the valid transactions
  candidateBlock = new Block(timestamp, previousBlockHash, transactions);

  // use proof of work to reach consensus on the candidate block
  nextBlock = proofOfWork(candidateBlock);

  // broadcast the new block to the network
  broadcast(nextBlock);
}
```

This is just one possible example of code for the design of a blockchain. The specific design and implementation will depend on the goals and requirements of the blockchain. This example is not complete and is not intended to be used as production code. It is intended only to illustrate the key components of a blockchain design.

Consensus mechanism

A consensus mechanism is a algorithm or protocol used by nodes in a distributed system to reach agreement on the state of the system. In a blockchain, the consensus mechanism is used to determine which transactions will be included in the next block of the ledger and to prevent conflicting transactions from being included.

Here is an example of a simple consensus mechanism written in pseudocode:

```
// define a list of valid transactions
transactions = [...];

// define a list of nodes in the network
nodes = [...];

// define a function to select the next block of transactions
function selectNextBlock() {
  // select a random node to propose the next block
  proposingNode = nodes[random(0, nodes.length)];

  // the proposing node creates a candidate block of transactions
  // by selecting a subset of valid transactions
  candidateBlock = transactions[random(0, transactions.length)];

  // the proposing node broadcasts the candidate block to the network
  broadcast(candidateBlock);

  // each node in the network verifies the candidate block
  for each node in nodes:
    if (verify(candidateBlock)) {
      // if the candidate block is valid, the node adds it to its local copy of the ledger
      addToLedger(candidateBlock);
    }
  }
}
```

This is a very simple example of a consensus mechanism, and it is not secure against various attacks. In a real blockchain, the consensus mechanism would be much more complex and secure.

Private blockchain

Here is an example of code for a private blockchain in pseudocode:

```
// define the data structure for the ledger
// the ledger will consist of a series of blocks,
// where each block contains a timestamp, a link to the previous block,
// and a list of transactions
class Block {
  timestamp;
  previousBlockHash;
  transactions;
}

// define the network architecture for the nodes
// the nodes will communicate using a secure, private network
```

```
// and will authenticate each other using a shared secret key
class Node {
  // method to broadcast a message to the network
  broadcast(message) {
    for each otherNode in nodes:
      send(message, otherNode, secretKey);
  }

  // method to receive a message from the network
  receive(message) {
    if (authenticate(message, secretKey)) {
      if (message.type == "transaction") {
        // if the message is a transaction, add it to the list of valid transactions
        transactions.add(message.transaction);
      } else if (message.type == "block") {
        // if the message is a block, verify it and add it to the local copy of the ledger
        if (verify(message.block)) {
          addToLedger(message.block);
        }
      }
    }
  }
}

function pbft(block) {
  // each node prepares a request to add the block to the ledger
  request = prepareRequest(block);

  // each node broadcasts its request to the other nodes
  broadcast(request);

  // each node waits for a quorum of nodes to send their requests
  quorum = (nodes.length / 2) + 1;
  receivedRequests = waitFor(quorum);

  // if a quorum of requests have been received, each node prepares a commit message
  if (receivedRequests.length >= quorum) {
    commit = prepareCommit();
    broadcast(commit);
  }

  // each node waits for a quorum of commit messages
  quorum = (nodes.length / 2) + 1;
  receivedCommits = waitFor(quorum);

  // if a quorum of commits have been received, each node adds the block to the ledger
  if (receivedCommits.length >= quorum) {
    addToLedger(block);
  }
}
```

Build a blockchain

The ledger is a data structure that stores the blocks and transactions in a blockchain. It is typically implemented as an array or linked list, where each element in the array or list represents a block. The ledger is updated as new blocks are added to the blockchain, and it is used to track the state of the blockchain and to verify the integrity of the blocks and transactions.

Here is an example of the ledger data structure in pseudocode:

```
// define the ledger data structure
class Ledger {
  // the ledger is a list of blocks
  ledger = [];
}
```

The specific functions that are needed in a blockchain class will depend on the design and implementation of the blockchain. However, here are some common functions that are often included in a blockchain class:

- `addBlock()`: This function adds a new block to the blockchain. It typically includes logic for verifying the new block, linking it to the previous block, and updating the ledger with the new block.
- `getLatestBlock()`: This function returns the latest block in the blockchain.
- `getBlock()`: This function returns a specific block in the blockchain, given its index or hash.
- `getTransaction()`: This function returns a specific transaction in the blockchain, given its hash.
- `getBalance()`: This function returns the balance of a specific address in the blockchain.
- `verify()`: This function verifies the integrity of a block or transaction in the blockchain. It typically includes logic for computing and comparing cryptographic hashes.

These are just some examples of the functions that are often included in a blockchain class. The specific functions and their implementation will depend on the design and requirements of the blockchain.

Here is an example of the `addBlock()` function in pseudocode:

```
function addBlock(block) {
  // verify the new block
  if (verify(block)) {
    // get the latest block in the blockchain
    latestBlock = getLatestBlock();

    // link the new block to the previous block
    block.previousBlockHash = latestBlock.hash;

    // compute the hash of the new block
    block.hash = hash(block);

    // add the new block to the ledger
    ledger.add(block);
  }
}
```

Here is an example of the `getLatestBlock()` function in pseudocode:

```
function getLatestBlock() {
  // get the index of the latest block in the ledger
  latestBlockIndex = ledger.length - 1;

  // return the latest block in the ledger
  return ledger[latestBlockIndex];
}
```

This is a very simple example of the `getLatestBlock()` function. It retrieves the index of the latest block in the ledger, and then returns the block at that index. This function is not complete and is not intended to be used as production code. It is intended only to illustrate the key steps for getting the latest block in a blockchain. The specific implementation will depend on the design and requirements of the blockchain.

Here is an example of the complete `getBlock()` function in pseudocode:

```
function getBlock(blockHash) {
  // iterate over the blocks in the ledger
  for each block in ledger:
    // if the block hash matches the given hash, return the block
    if (block.hash == blockHash) {
      return block;
    }
  }

  // if no matching block was found, return null
  return null;
}
```

This function iterates over the blocks in the ledger and returns the first block that has a hash that matches the given block hash. If no matching block is found, it returns `null`. This function is not complete and is not intended to be used as production code. It is intended only to illustrate the key steps for getting a specific block in a blockchain. The specific implementation will depend on the design and requirements of the blockchain.

Here is an example of the `getTransaction()` function in pseudocode:

```
function getTransaction(transactionHash) {
  // iterate over the blocks in the ledger
  for each block in ledger:
    // iterate over the transactions in the block
    for each transaction in block.transactions:
      // if the transaction hash matches the given hash, return the transaction
      if (transaction.hash == transactionHash) {
        return transaction;
      }
    }
  }

  // if no matching transaction was found, return null
  return null;
}
```

This function iterates over the blocks and transactions in the ledger, and returns the first transaction that has a hash that matches the given transaction hash. If no matching transaction is found, it returns `null`. This function is not complete and is not intended to be used as production code. It is intended only to illustrate the key steps for getting a

specific transaction in a blockchain. The specific implementation will depend on the design and requirements of the blockchain.

Here is an example of the `getBalance()` function in pseudocode:

```
function getBalance(address) {
  // initialize the balance to 0
  balance = 0;

  // iterate over the blocks in the ledger
  for each block in ledger:
    // iterate over the transactions in the block
    for each transaction in block.transactions:
      // if the transaction involves the given address, update the balance
      if (transaction.from == address) {
        balance -= transaction.amount;
      }
      if (transaction.to == address) {
        balance += transaction.amount;
      }
    }
  }

  // return the balance
  return balance;
}
```

This function iterates over the blocks and transactions in the ledger, and updates the balance for the given address by adding the amounts of any transactions where the address is the sender (`from`) or receiver (`to`). It returns the final balance. This function is not complete and is not intended to be used as production code. It is intended only to illustrate the key steps for getting the balance of an address in a blockchain. The specific implementation will depend on the design and requirements of the blockchain.

Here is an example of the complete `verify()` function in pseudocode:

```
function verify(blockOrTransaction) {
  // compute the hash of the block or transaction
  computedHash = hash(blockOrTransaction);

  // compare the computed hash to the stored hash
  if (computedHash == blockOrTransaction.hash) {
```

```
    return true;
  } else {
    return false;
  }
}
```

This function computes the hash of a block or transaction, and then compares the computed hash to the stored hash. If the computed hash matches the stored hash, it returns `true` to indicate that the block or transaction is valid. If the computed hash does not match the stored hash, it returns `false` to indicate that the block or transaction is invalid. This function is not complete and is not intended to be used as production code. It is intended only to illustrate the key steps for verifying a block or transaction in a blockchain. The specific implementation will depend on the design and requirements of the blockchain.

Election smart contracts

Here is an example of an election smart contract in the Solidity programming language:

```
pragma solidity ^0.5.0;

// define the contract
contract Election {
  // define the mapping for the candidates
  mapping(uint => string) public candidates;

  // define the array for the votes
  uint[] public votes;

  // define the constructor
  constructor() public {
    // initialize the candidates
    candidates[1] = "Alice";
    candidates[2] = "Bob";
    candidates[3] = "Charlie";
  }

  // define the function to cast a vote
  function castVote(uint candidateId) public {
    // add the vote to the array of votes
    votes.push(candidateId);
  }

  // define the function to get the result
  function getResult() public view returns (string[] memory) {
    // initialize the array for the result
```

```
    string[] memory result = new string[](3);

    // iterate over the candidates
    for (uint i = 1; i <= 3; i++) {
      // initialize the vote count to 0
      uint voteCount = 0;

      // iterate over the votes
      for (uint j = 0; j < votes.length; j++) {
        // if the vote matches the candidate, increment the vote count
        if (votes[j] == i) {
          voteCount++;
        }
      }

      // set the result for the candidate
      result[i - 1] = candidates[i] + ": " + voteCount;
    }

    // return the result
    return result;
  }
}
```

NFT marketplace contract

Here is an example of a smart contract in the Solidity programming language that implements a marketplace for non-fungible tokens (NFTs):

```
pragma solidity ^0.5.0;

// import the ERC-721 interface
import "https://github.com/OpenZeppelin/openzeppelin-solidity/contracts/token/ERC721/ERC72
1.sol";

// define the contract
contract NFTMarketplace is ERC721 {
  // define the mapping for the token metadata
  mapping(uint256 => string) public tokenMetadata;

  // define the mapping for the token prices
  mapping(uint256 => uint) public tokenPrices;

  // define the mapping for the token owners
  mapping(uint256 => address) public tokenOwners;

  // define the event for a token purchase
```

```
  event TokenPurchase(
    uint256 indexed tokenId,
    address indexed buyer,
    uint price
  );

  // define the function to mint a new token
  function mint(string memory metadata, uint price) public {
    // mint the new token and get its token ID
    uint256 tokenId = _mint(msg.sender);

    // set the metadata and price for the token
    tokenMetadata[tokenId] = metadata;
    tokenPrices[tokenId] = price;
  }

  // define the function to buy a token
  function buyToken(uint256 tokenId) public payable {
    // check that the token is not already owned
    require(
      tokenOwners[tokenId] == address(0),
      "Token is already owned."
    );

    // get the price of the token
    uint price = tokenPrices[tokenId];

    // check that the buyer has enough funds
    require(
      msg.value >= price,
      "Insufficient funds to buy token."
    );

    // transfer the token from the contract to the buyer
    _transfer(address(0), msg.sender, tokenId);

    // update the token owner
    tokenOwners[tokenId] = msg.sender;

    // send the funds to the contract owner
    msg.sender.transfer(price);

    // emit the token purchase event
    emit TokenPurchase(tokenId, msg.sender, price);
  }
}
```

This contract implements the basic functionality for a marketplace for NFTs

Complete courses

- https://www.blocktrain.info/course