Blockchain
Without the Fog

From your first transaction
to your own service on RpcNode

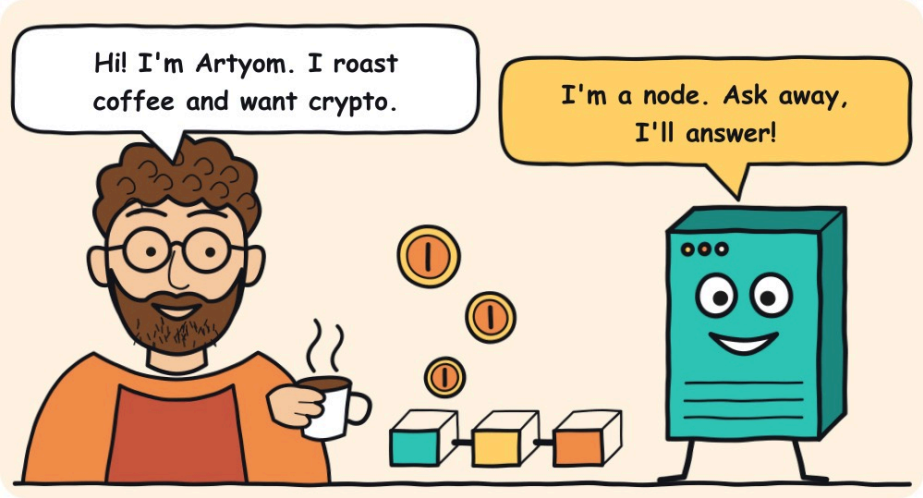


In plain words · with pictures and examples

In plain words

Blockchain Without the Fog

From your first transaction to your own service on RpcNode



This book is for you if you

- have heard the word “blockchain” but don’t know how it works;
- want to accept cryptocurrency on your website;
- write programs and have never talked to a node.

What you will need

- curiosity — no prior knowledge required;
- a computer with a terminal, to repeat the examples;
- the first requests work with no sign-up and no key.

How to read a chapter

The question · Digging in · Try it · The trap · Think it over

Author: rpcnode.dev · 2026

Contents

Part 1. Why you need a provider

1. I want to accept crypto 4
2. What doing it all yourself costs 9
3. Provider or your own node 15

Part 2. Blockchain from scratch

4. Money without a middleman 21
5. Blocks and the chain 26
6. Keys, addresses and signatures 31
7. A transaction's journey 36
8. How a network reaches agreement 41
9. Coins, tokens and smart contracts 47

Part 3. Nodes and RPC

10. Kinds of nodes 53
11. How to talk to a node 58
12. Your own node from the inside 64
13. How to choose a provider 70

Part 4. How networks work

14. Bitcoin and its relatives 75
15. Ethereum: accounts, gas and events 80
16. Ethereum's relatives and L2s 85
17. TRON: energy instead of gas 90
18. Solana: everything is accounts 96
19. TON: letters instead of calls 102
20. Other approaches 108
21. Which network to choose 115

Part 5. RpcNode

22. Your first five minutes in RpcNode 122
23. What happens to a request 128
24. History and archive data 134
25. Plans, call prices and limits 140
26. Address events 146
27. Payments and transaction tracking 152

Part 6. Practice

28. Building the shop 158
29. When everything goes wrong 165
30. Security and launch checklist 172

Part 7. What's next

31. DeFi: finance without a middleman 178

At the end of the book

Conclusion 185
Appendix A — Glossary 187
Appendix B — Networks on one page 189
Appendix C — Request cheat sheet 191

Chapter 1. I want to accept crypto: what do I need?

It all started with one customer

Artyom has a small online shop: he roasts coffee and mails it out. Cards, bank transfers, cash on delivery — it all works already. Then an email arrives:

> "Hello! I'd like to order two kilos. Can I pay in cryptocurrency?"

Artyom doesn't mind. He installs a wallet on his phone, gets an address and sends it to the customer. A few minutes later the money shows up on the screen.

By the tenth order, checking the wallet by hand gets old. He wants the website itself to know that an order has been paid.



Digging in: what a node is

To find out anything about a blockchain, you have to ask someone who stores it. That someone is called a **node**.

Picture a shared notebook that records who sent how much to whom. The notebook doesn't sit in a single safe. Many participants around the world have copies of it, and they keep checking them against each other. That is a blockchain, and we will take it apart in detail in Chapter 5.

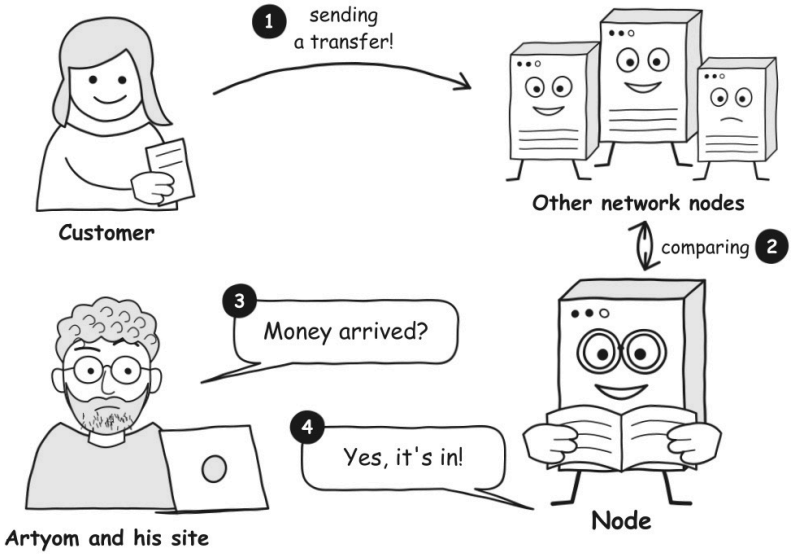
A node is a computer running the network's software. It does four things:

1. Keeps its own copy of the records.
2. Receives new records from other nodes and checks them against the network's rules.
3. Passes on the transfers people send to it.

4. Answers programs' questions: "what's the balance?", "did the transfer go through?".

Nodes come in different kinds: some don't keep the whole history. More on that in Chapter 10.

Artyom's website doesn't look into the blockchain by itself. It asks a node a question, and the node answers.



Try it: your first question to a node

Programs talk to a node by simple rules: send a request, get a response. One such set of rules is called **JSON-RPC**. Here is a question to an Ethereum node: "how many coins are on this address?"

```
curl https://NODE_ADDRESS \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance","params":["WALLET_ADDRESS","late
st"]}'
```

Here `NODE_ADDRESS` is where we are knocking, and `WALLET_ADDRESS` is whose balance we are asking about. The word `latest` means "as of the most recent moment".

The node replies:

```
{"jsonrpc":"2.0","id":1,"result":"0xde0b6b3a7640000"}
```

It looks like a jumble of characters, but it is a number written in hexadecimal: 1,000,000,000,000,000,000. The node counts in the tiniest fractions of a coin, called **wei**. One ETH is exactly that many of them. So there is 1 ETH on the address.

So the whole trick is a question and an answer. What remains is to decide whose `NODE_ADDRESS` it will be.

Two paths: your own node or a provider

You can run a node yourself. Or you can connect to someone else's, which they keep running for you. Such a service is called a **provider**. It hands you a ready-made `NODE_ADDRESS`. RpcNode, which this book is about, is one of these services.

Question	Your own node	Provider
When can you start	After installation and syncing: from hours to weeks, depending on the network	After signing up, within minutes
What you pay for	Server, disks, traffic and a specialist's time	The service plan
Who updates and fixes it	You	The provider
Several networks	A separate node for each	Usually through one account
What you risk	The node fell behind or broke while you were asleep	You depend on someone else's service

Both paths have their place. Your own node makes sense if you need special privacy, or your load is very heavy, or you take part in running the network itself.

> **Artyom:** "I roast coffee, I don't babysit servers. But how much would doing it myself actually cost?"

The short answer: for several networks with backup servers and full history, the bill runs into tens of thousands of dollars. A single small node can be run far more cheaply. The detailed math is in Chapter 2.

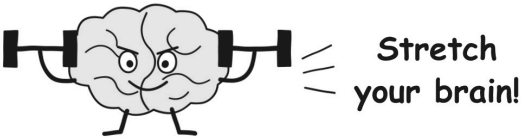
The trap: "I'll give the provider my wallet key"

Never. To learn a balance or see a payment, a node only needs the address. The private key and the secret phrase control the money: whoever knows them can take everything. An honest provider has no use for them. We will cover keys in Chapter 6.



And a second caveat. The request above shows only the network's native coin, here ETH. If a customer pays with a token, such as USDT, this question won't see it. Tokens are asked about differently — that's Chapter 9.

Think it over



1. Artyom checks for a payment, but his node is an hour behind the network. What will he see, and what will he write to the customer?
2. The shop gets three orders a day. What if it were three thousand? How would that change the choice between your own node and a provider?

Hint for the first: a lagging node faithfully shows how things were an hour ago.

Key points

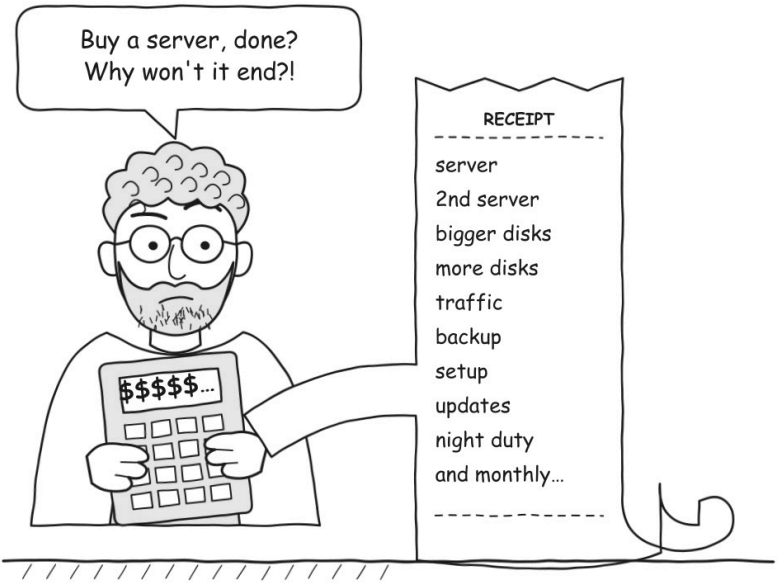
- A program learns about the blockchain by asking a node questions.
- You can run a node yourself or get one from a provider.
- A node needs the address, but not your private key.

Next: Chapter 2. Artyom picks up a calculator and works out what his own servers would cost.

Chapter 2. What does it cost to do it all yourself?

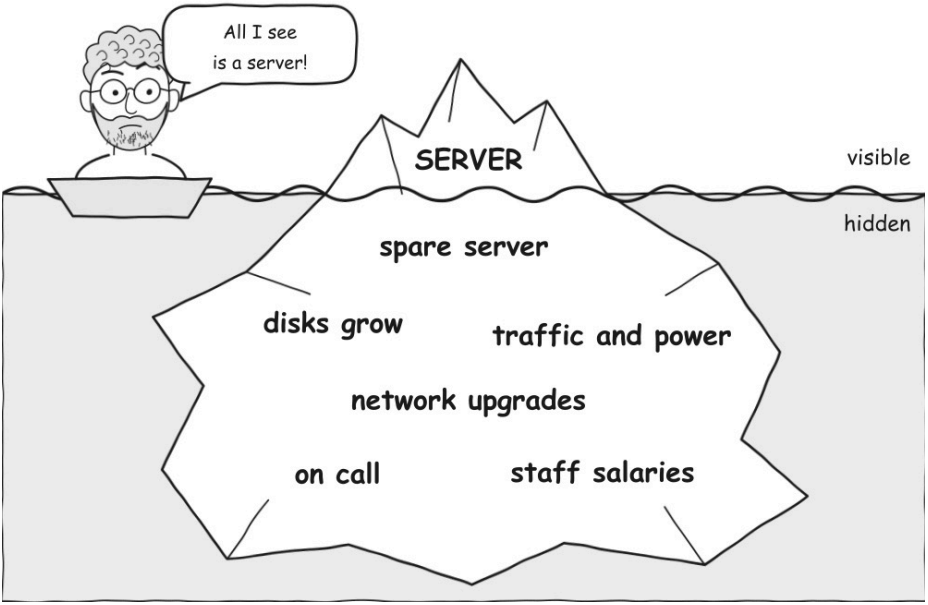
Artyom picks up a calculator

Artyom's reasoning is simple: "I'll buy a server, put a node on it, and payments will take care of themselves. I'll pay once." He opens the calculator and starts writing things down.



Digging in: what makes up the bill

The server is only the tip. The rest isn't visible at first.



1. **The server.** Processor, memory and fast NVMe disks.
2. **A spare server.** One breaks — the other answers while the first is being fixed.
3. **Disks keep growing.** A blockchain only ever adds new data. In the year to November 2026, the Bitcoin chain grew from about 700 to 786 GB.
4. **Traffic.** A node exchanges data constantly. A TON liteserver needs about 16 TB a month at peak.
5. **Updates.** The network changes its rules, and the old software can no longer keep up.
6. **People.** Someone has to set all this up, watch it and wake up at night.

How much hardware each network asks for

These are the requirements for an ordinary node from the official documentation, not for taking part in the network's voting.

Network	Processor	Memory	Disk
Bitcoin	1 GHz or more	2 GB	750 GB recommended, while the chain itself is already about 786 GB
Ethereum	4+ cores	32 GB	4 TB NVMe
BNB Smart Chain	16 cores	64 GB	SSD of 3 TB or more; a full database snapshot is already about 7.4 TB, a pruned one about 1.9 TB
TRON	16 cores	32 GB	SSD of 3.5–4 TB or more
Solana (RPC node)	16 cores or more	512 GB or more	several NVMe drives: accounts 1 TB+, ledger 1 TB+, snapshots 500 GB+

Noticed the oddities? For Bitcoin, the recommended disk is smaller than the chain itself. For BNB the minimum is 3 TB, while a full snapshot is more than twice that. Documentation doesn't always keep pace with the network, so people buy disks with room to spare.

And if you need the whole history?

An ordinary node knows the present well. To ask "what was the balance a year ago?", you need an **archive** node; see Chapter 10. It is noticeably heavier. A TON archive needs 16–20 TB. For Ethereum the size depends greatly on the software: from 2 to 14 TB and more. Our estimate leaves archives out: with them it would cost more.

Counting the money

You can buy your own server, but we will price renting one. This is an estimate based on hosting companies' public price lists as of November 27, 2026, excluding taxes and setup fees. Prices change.

Server class	What's inside	About per month
Medium	8 cores, 64 GB, 2×2 TB NVMe	$180
Large	16 cores, 128 GB, 4–8 TB NVMe	$460–560
Very large	512 GB of memory or more	$1,160–2,350

Now take five networks. We pick prices near the low end and put two servers on each network: a working one and a spare.

Network	Class	One server, per month	With a spare, per month
Bitcoin	medium	$180	$360
Ethereum	medium	$180	$360
BNB Smart Chain	large	$500	$1,000
TRON	large	$500	$1,000
Solana	very large	$1,200	$2,400
Total		**$2,560**	**$5,120**

Artyom: "$5,120 a month is $61,440 a year. And that's just rent?!"

Yes. Take the top prices from the first table and you get about $92,000 a year. That is where "tens of thousands of dollars" comes from.

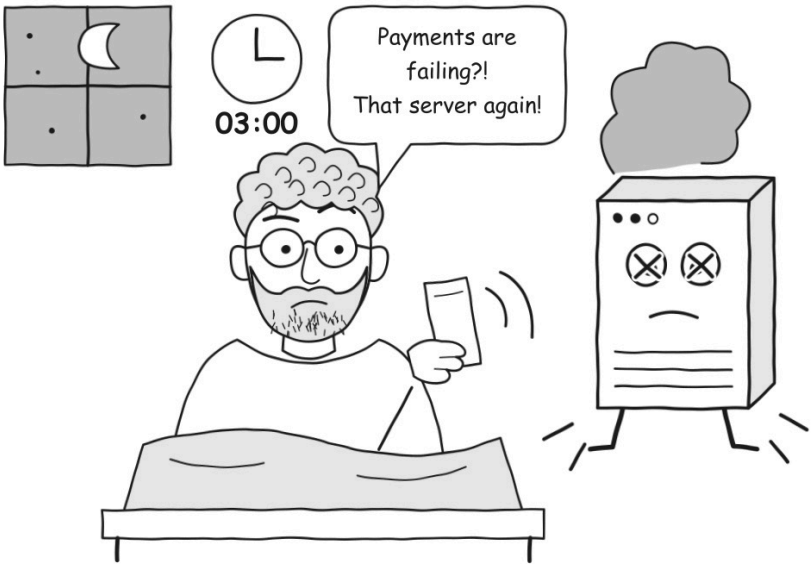
Keeping a single network is far cheaper. Bitcoin isn't demanding: even a server more modest than the medium one will do.

And who is going to look after it?

Servers don't maintain themselves. According to Habr Career, a mid-level DevOps engineer in Russia earns roughly 160,000–290,000 rubles a month. One person cannot be on call around the clock, with no weekends or vacation. Even two already come to about 3.8–7 million rubles a year.

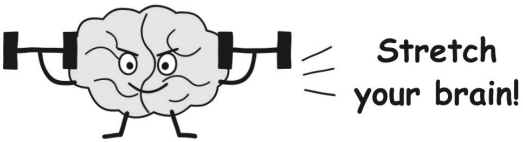
The trap: "I'll buy one server and I'm done"

One server is a single point of failure. The disk filled up, the software froze, the network upgraded — and the website no longer sees payments. The customer has sent the money, yet the order says "unpaid".



And one more thing people often forget. Read the hosting company's terms before you pay: some forbid cryptocurrency-related software and may shut your server off.

Think it over



1. Artyom only needs Bitcoin, with a spare medium-class server. How much is that a year?
2. A sixth network on large $500 servers, also with a spare, is added to the five. How much does the yearly bill grow?
3. Customers pay in crypto three times a day. Are they worth this expense?

Answers: 1) 360 × 12 = $4,320. 2) 1,000 × 12 = $12,000. 3) That's your call, and Chapter 3 will help.

Key points

- You pay not once but every month.
- Five networks with spares come to more than $60,000 a year in rent alone, without archives or people.
- Disks grow, networks upgrade, someone has to be on call.

Next: Chapter 3. What a provider takes off your hands, and when you really do need your own node.

Where the numbers come from

All pages were accessed on November 27, 2026.

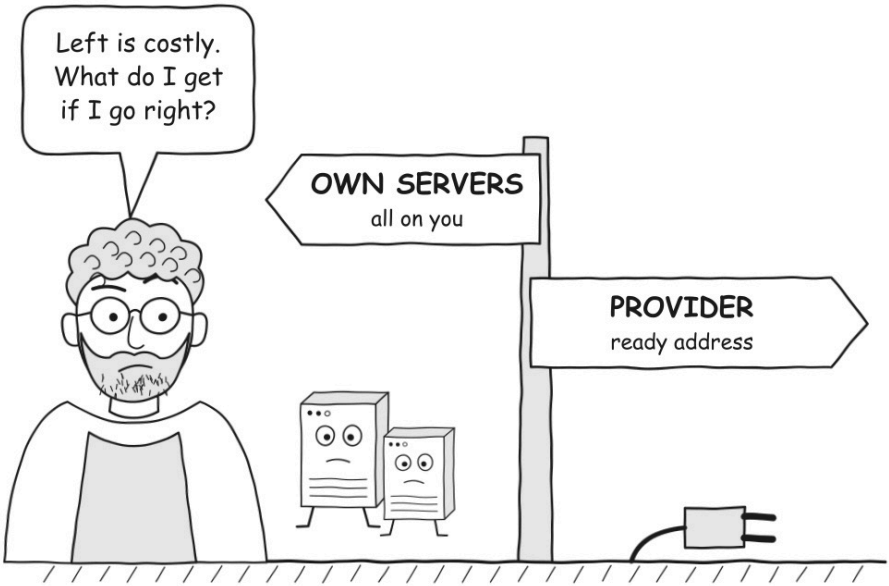
- Bitcoin, requirements: bitcoin.org/en/bitcoin-core/features/requirements
- Bitcoin, chain size: api.blockchain.info/charts/blocks-size
- Ethereum: ethereum.org/developers/docs/nodes-and-clients/run-a-node
- BNB Smart Chain: docs.bnbchain.org/bnb-smart-chain/developers/node_operators/node_best_practices; github.com/bnb-chain/bsc-snapshots
- TRON: tronprotocol.github.io/documentation-en/using_javatron/installing_javatron; developers.tron.network/docs/deploy-the-fullnode-or-supernode
- Solana: docs.anza.xyz/operations/requirements
- TON: docs.ton.org/nodes/cpp/run-liteserver; docs.ton.org/nodes/cpp/run-archive-liteserver
- Salaries: career.habr.com/professions/devops (data for 2025–2026)

Server prices are the author's rounded estimate based on several hosting companies' public price lists.

Chapter 3. What a provider gives you, and when you do need your own node

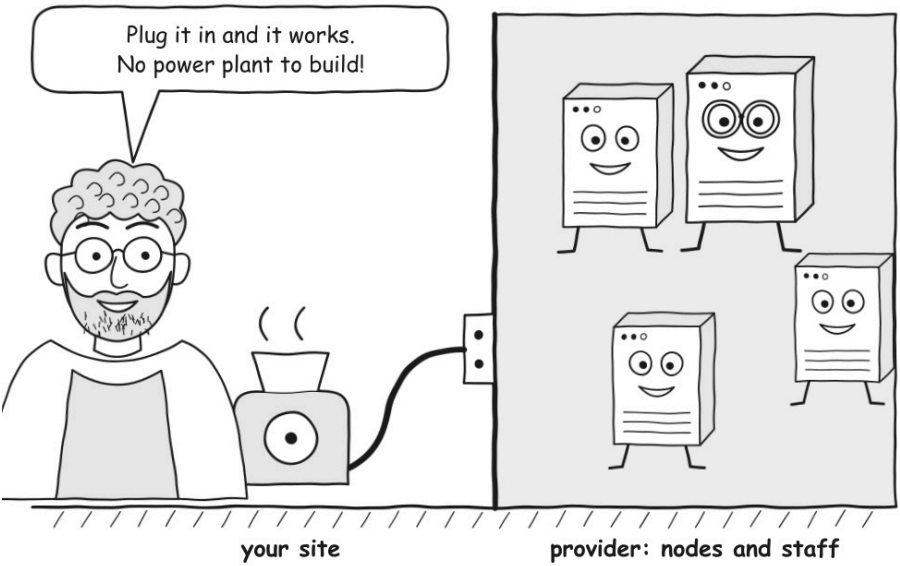
Artyom at a fork in the road

After the math in Chapter 2, the calculator lies face down. Five networks with spares mean more than $60,000 a year in rent alone. Coffee doesn't bring in that much.



Digging in: what a provider takes on

You get electricity from a wall socket; you don't build a power station for the sake of your kettle. It's the same with nodes: a **provider** runs them on its side and gives you an address you can send questions to.



Usually five chores are on the provider's side:

1. **Hardware and disks.** Buying, renting, expanding when space runs out.
2. **Syncing.** The node must not fall behind the network.
3. **Updates.** The network changes its rules — the software gets changed in time.
4. **Replacing what breaks.** Requests are routed to a healthy node.
5. **On-call duty.** Someone other than Artyom wakes up at night.

Providers differ, and so do their promises. Check exactly which items on this list you are getting before you pay. We will look at how to choose in Chapter 13.

What stays your responsibility

- **Your wallet keys.** Only you have them.
- **The endpoint address.** That is the name for the address the provider gave you. It often has your personal access key built into it. Guard it like a password: other people's requests will eat up your quota.
- **Your code.** It has to survive an error, retry a request and never count one payment twice.

Three words from any plan

Word	What it means	Ask yourself
Requests per second (RPS)	How often you may ask	How many customers will show up at once?
Quota, or credits	How much you can ask in total over a period. Different questions cost different amounts	Will it last to the end of the month?
Archive access	Whether you can ask about the distant past	Do I need old history?

Try it: a question to a real node

In Chapter 1 there was `NODE_ADDRESS` where an address should be. Now let's put a real one there. RpcNode has free public nodes that need neither sign-up nor key. Let's ask for the number of the latest block on the Ethereum network:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
```

Here is what the node answered while this chapter was being written:

```
{"jsonrpc":"2.0","id":1,"result":"0x1946393"}
```

`0x1946393` is 26,502,035. Your number will be bigger: the network doesn't stand still. Repeat the request a minute later and see for yourself.

Public nodes are shared by everyone, so they come with limits: as of November 27, 2026, that is 5 requests per second from one address, and the most "expensive" methods are closed. Enough to get acquainted. A shop needs its own endpoint.

What it costs

As an example, here are RpcNode's first two plans as of November 27, 2026. Check the website for current ones.

	Free	Starter
Price per month	$0	$19
Credits	15 million	50 million
Requests per second	15	40
Endpoints	3	5
Networks	Ethereum and Solana	all
Archive access	no	yes

RpcNode's nodes are located in three geographies: Frankfurt, the USA and Japan.

> **Artyom:** "$19 against $5,120 a month? What's the catch?"

These are different things. Your own server is all yours. On a plan you get a share of common capacity, with limits. For a small shop a share is enough, and there's no reason to pay for a whole "power station".

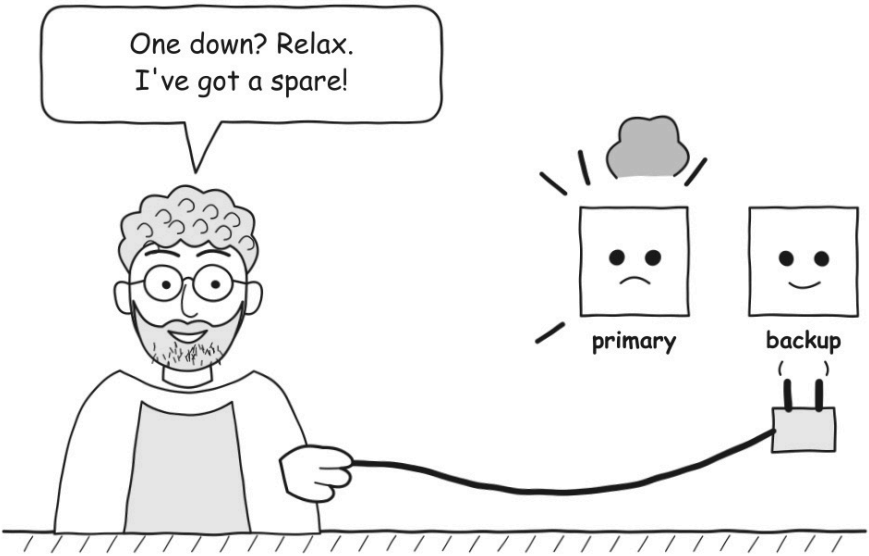
When you really do need your own node

- **Privacy.** The provider sees your requests: which addresses you check and where you ask from.
- **Enormous load.** There are so many requests that a plan costs more than your own servers.
- **Taking part in the network.** You are a validator or you develop the network itself.
- **Special settings.** You need a method or mode the provider doesn't offer.
- **Legal or client requirements.** The data must be processed only on your side.

There is also a mixed path: your own node for the main network, a provider for the rest and as a safety net.

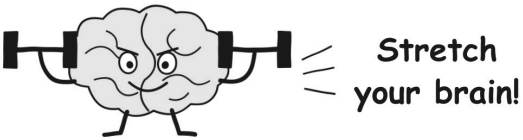
The trap: "a provider never goes down"

Everything goes down: servers, network links, and providers' services too. If payments matter to you, decide in advance what your website does when there is no answer.



- Don't tell a customer "unpaid" when you simply haven't managed to ask.
- Retry the request later, not a hundred times in a row.
- Keep a backup address: a second provider or your own node.

Think it over



1. Artyom's website checks for payment once a second for each of 20 pending orders, one request per check. Will that fit into 15 requests per second?
2. What should a customer see if the node doesn't answer for ten minutes?
3. Which of them needs their own node: a coffee shop or a team developing the network itself?

Hint for the first: 20 is more than 15. Check less often or ask more cleverly — how, in Chapter 26.

Key points

- A provider takes on the hardware, syncing, updates and on-call duty.
- The keys, the endpoint address and how your website behaves during outages remain yours.
- You need your own node for privacy, heavy load, taking part in the network or special requirements.

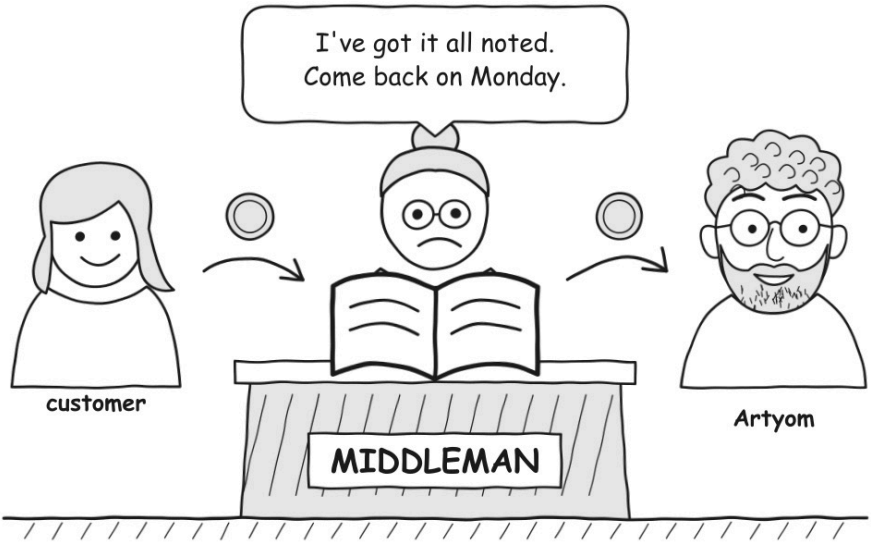
Part one is over. Artyom has chosen a provider and got an answer from a real node. But what is this "block" number 26,502,035 that it named? **Next:** Part 2, Chapter 4. Money and records without a middleman.

Chapter 4. Money and records without a middleman

Where is your money kept?

Artyom received a transfer to his card. Not a single banknote travelled anywhere. So what happened? Someone lowered the number next to the customer's name and raised the number next to Artyom's.

Cashless money is a record. And whoever keeps the records is called a middleman: a bank or a payment service.



Digging in: why a middleman is needed and why he is inconvenient

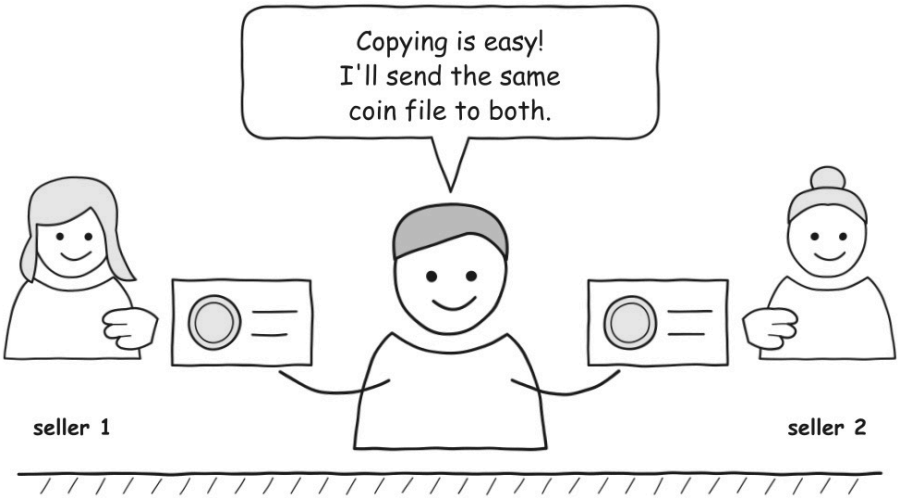
A middleman does a lot of useful things. Checks that there is enough money. Keeps it from being spent twice. Can settle a dispute and reverse a payment.

But all the records are in his hands. He can stop a transfer, take a fee, block an account or be closed for the weekend. This is especially noticeable when the customer lives in another country.

> **Artyom:** "Then let the customer send me the money as a file. Like a photo. No middlemen at all!"

Why you can't just send a file

You can send a photo a hundred times and still have it. Money can't work like that: one coin would pay two people. This is called a **double spend**.

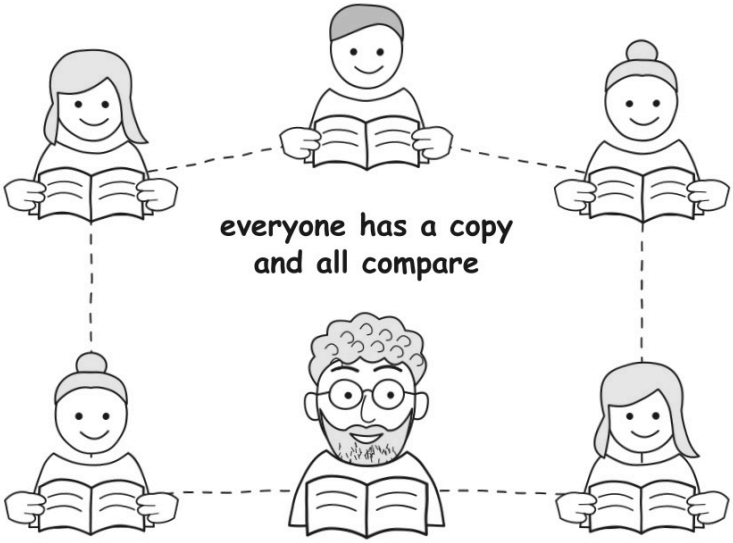


For a long time there was only one solution: someone in charge keeps the list and says whose coin it is now. In other words, a middleman again.

The idea: everyone keeps the records

In 2008 an author under the pseudonym Satoshi Nakamoto described another way, and in January 2009 the Bitcoin network started running. The idea goes like this:

1. The list of transfers isn't locked away with one party. **Many have a copy.**
2. Participants **check every new transfer against shared rules**: is the money there, and has it already been spent.
3. Participants **agree** on which records are correct, with nobody in charge.



Such a shared list, kept jointly, is called a **blockchain**. We will see how records form a chain in Chapter 5, and how participants agree in Chapter 8.

Try it: the shared notebook game

No code needed. Get two friends and let each have their own sheet of paper. At the start everyone writes down: Anya has 10 coins, Boris 5, Vera 0. There is one rule: **you can't give more than you have**.

No.	Who wants to send	Accept?	Anya	Boris	Vera
start			10	5	0
1	Anya → Vera 4				
2	Vera → Boris 1				
3	Boris → Anya 7				
4	Anya → Boris 6				

Fill in the empty cells, then compare the sheets. If they all match, you have just reached agreement with nobody in charge. The answer is at the end of the chapter.

What a blockchain gives you and what it doesn't

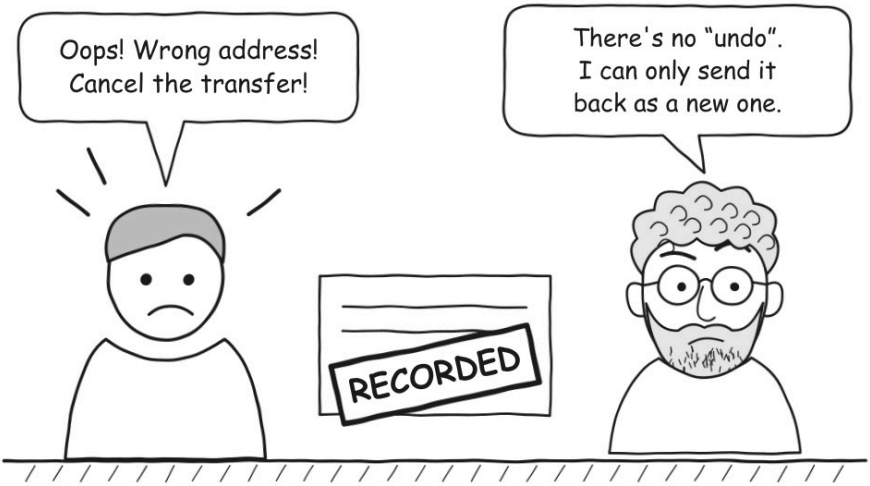
Gives	Doesn't give
Direct transfers, any day and hour	An "undo" button
One account accessible from different countries	A stable coin price: the rate can swing sharply
Records anyone can verify	Secrecy: every address's transfers are visible to all
No need to trust anyone to keep the list	A support desk that will recover a lost key

Not everything swings: some coins try to stay close to one dollar, and they are called **stablecoins**. But behind the most widespread ones stands an issuing company, and it can freeze the money on an address. Details are in Chapter 9.

Middlemen haven't vanished: exchanges, wallets, providers. But they differ. An exchange may hold your coins itself. The provider from part one doesn't hold money: it only passes your questions on to a node.

The trap: "if I don't like it, I'll undo it"

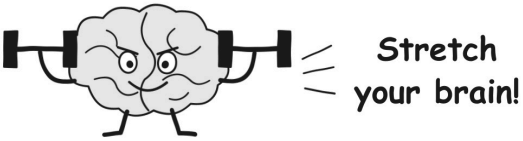
A confirmed transfer cannot be undone. Get the address wrong and the money is gone. Only the recipient can return it, if they want to and if the address belongs to anyone at all.



That is why people check the address twice, and before sending a large sum they first try a small part of it.

A little story. In May 2010, someone paid 10,000 bitcoins for two pizzas. That transfer is still written in the shared notebook.

Think it over



1. After the game Vera has 3 coins. At the same moment she writes to Anya "I give you 3" and to Boris "I give you 3". What should the participants do?
2. And if one participant accepted the first transfer and another the second? Who is right?
3. Which suits Artyom better: being able to reverse a payment or not? And his customer?

Hints: 1) only one can be accepted. 2) This is the hardest question, and Chapter 8 is devoted to it.

Key points

- Cashless money is records, and someone has to keep them.
- A blockchain is a shared list that many keep by common rules.
- Freedom from a middleman also means responsibility: there is nobody to undo a transfer or recover a key.

Answer to the game: transfer 1 accepted (6, 5, 4); 2 accepted (6, 6, 3); 3 rejected, Boris has only 6 (6, 6, 3); 4 accepted (0, 12, 3). There are still 15 coins in total.

Next: Chapter 5. What the "block" the node named is, and why an old record can't be quietly rewritten.

Chapter 5. Blocks and the chain: why the past can't be quietly rewritten

That very number

In Chapter 3 a node answered Artyom: the latest block is number 26,502,035. What is a block? Why does it have a number? And what stops someone from secretly correcting an entry in the notebook?

Digging in: a block is a page of the notebook

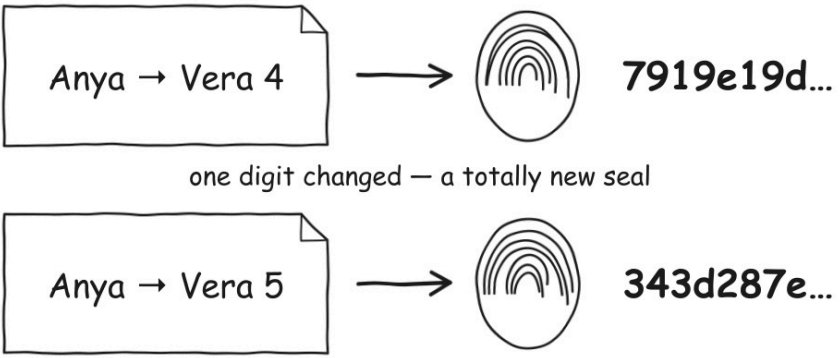
Transfers aren't written down one at a time. They are gathered into batches, like lines on a page. Such a page is called a **block**. On it you will find:

1. **Transfers** and other operations, called transactions.
2. The page's **number** in order.
3. The **time** it was made.
4. **The seal of the previous page.** That's the whole secret.

The seal, or hash

For any text you can compute a short string of digits and letters. It is a **hash**; we will call it a seal. It is like a fingerprint:

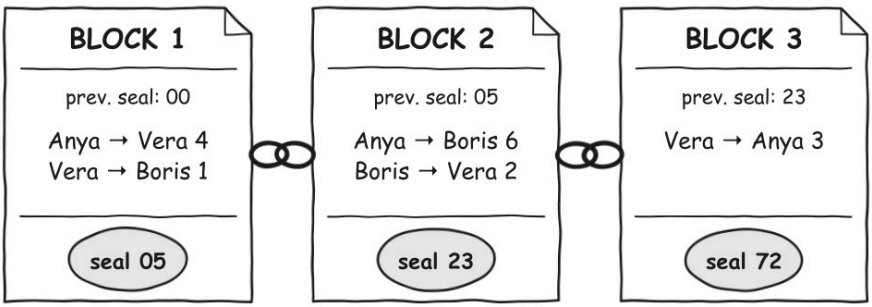
- the same text always gives the same seal;
- change even one letter and the seal is completely different;
- you can't rebuild the text from the seal;
- finding another text with the same seal is impossible in practice.



These are the beginnings of real SHA-256 hashes for the two lines in the picture. Bitcoin uses SHA-256; other networks compute seals in other ways too.

The chain

Every page records the seal of the one before. A page gets its own seal from everything on it, **including that previous seal**.

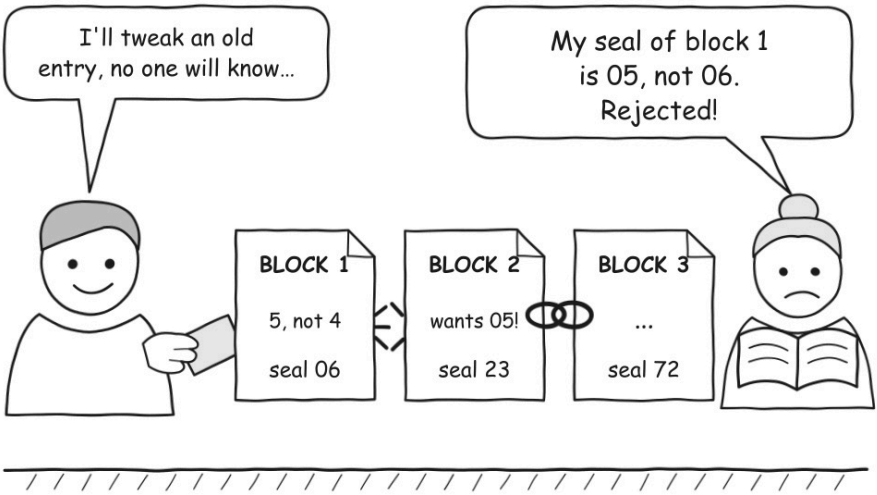


That is how pages link into a **chain of blocks** — a blockchain. That is the whole story behind the name.

The seals in the picture are toy ones, two digits long, so you can work them out in your head.

What happens if you tweak the past

A cheater changes a 4 to a 5 in the first block. The block's seal changes. But the second block has the old one written down. The chain breaks, and it shows at once.



He can rewrite all the following seals too. But other participants have their own copies, and his version won't match them. To get his version accepted, he would have to overpower the rest of the network. That is very expensive. We will see exactly how the network decides whose chain is correct in Chapter 8.

Take a look at a real chain

These are three blocks of the Ethereum network from November 26, 2026. The seals are long, so only the beginning and the end are shown.

Block number	Its seal	Previous seal	Transactions
26,502,033	`0xbe58…d764`	`0x2844…c876`	229
26,502,034	`0x7bb7…b7a4`	`0xbe58…d764`	229
26,502,035	`0xeab2…c904`	`0x7bb7…b7a4`	515

Run your finger along it: the seal in one row repeats in the next. Exactly 12 seconds passed between each of these blocks. In Bitcoin a new block appears less often, on average about once every 10 minutes.

The very first block has no previous one. The first Bitcoin block was created on January 3, 2009.

Try it: your own seal

Let's invent a toy rule:

Seal = (previous seal × 3 + all the amounts on the page), then take the last two digits of the result.

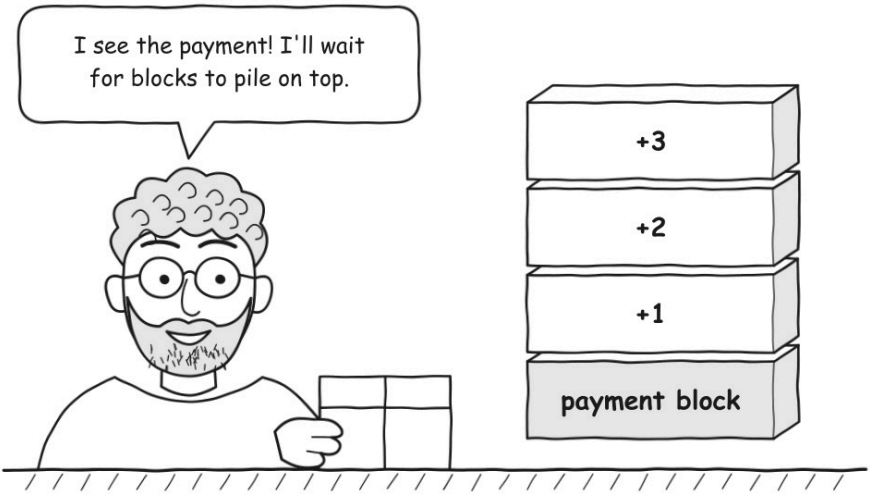
Block	Transfer amounts	Seal	If the 4 in block 1 becomes a 5
1	4 and 1		
2	6 and 2		
3	3		

For the first block the previous seal is 0. Fill in both columns. How many seals changed?

The trap: "it's in a block, so it's there forever"

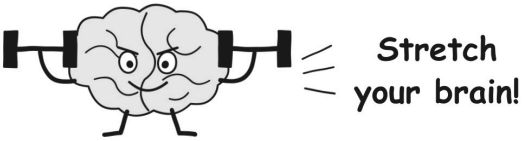
The freshest blocks are sometimes replaced by others: parts of the network haven't agreed yet. The more blocks pile on top, the harder it is to remove your entry.

This is counted in **confirmations**: the block with the payment itself is the first, and each following block adds one more.



How long to wait depends on the network and on the amount. Less for a cup of coffee, longer for a large consignment. Details are in Chapter 8.

Think it over



1. Fool our seal: replace the amounts 4 and 1 in block 1 with others so that the seal doesn't change. What does that say about our rule?
2. Why is it harder to change a block from three years ago than yesterday's?
3. If a block appears every 12 seconds, how many blocks will there be in an hour?

Answers: 1) for example, 3 and 2: the sum is the same. A real hash would notice that. 2) You would have to redo everything that piled on top. 3) 300.

Key points

- A block is a page with transactions, a number, a time and the seal of the previous page.
- Changing an old entry changes the seal and breaks the chain.
- A fresh block is trusted less: people wait for confirmations.

Answer to the game: seals 05, 23, 72. After the replacement — 06, 26, 81: all three changed.

Next: Chapter 6. How does the network know that it was the owner of the money who sent a transfer? Keys, addresses and signatures.

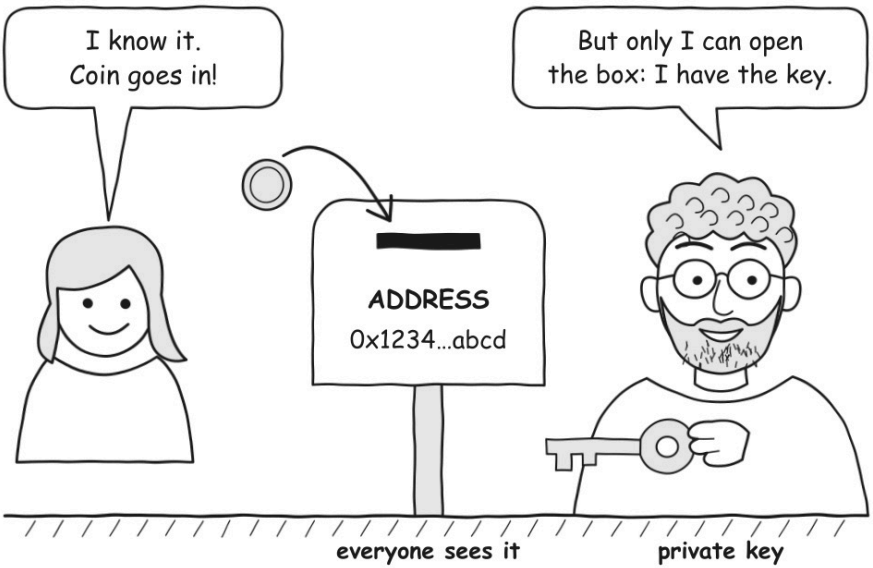
Chapter 6. Keys, addresses and signatures

How does the network know this money is yours?

There are no passports in the shared notebook. Nobody asks your name or checks your photo. So what stops anyone from writing "Artyom gives me everything"?

Digging in: a mailbox and a key

Picture a mailbox on the street. Everyone knows the box number, and anyone can drop a coin through the slot. But only the one who has the key can open the box and take it.



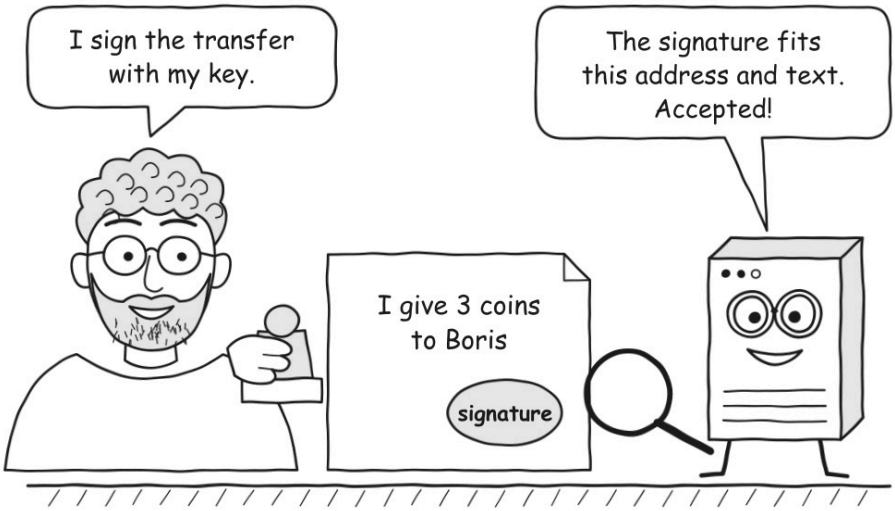
What	What it's like	Whom to show
Address	The box number	Anyone, to receive money
Public key	The specimen a signature is checked against	Fine to show everyone
Private key	The key to the box	**Nobody**

They are linked in one direction. The public key is derived from the private key, and the address from the public key. Computing backwards is impossible in practice. In some networks the address simply is the public key, written down in a special way.

A **wallet** is a program or device that stores keys. There are no coins inside it: they remain entries in the shared notebook.

The signature

To give away coins, the owner draws up a transfer and **signs** it with the private key. The key itself isn't sent anywhere. The transfer and the signature go to the network.



A node checks two things: that the signature was made with the key of this very address, and that the text hasn't been changed since signing. A signature on one transfer won't fit another. And the network won't carry out the very same transfer a second time.

Try it: a toy signature

All of it can be done on a calculator. The private key is the number **7**, the public key is the number **3**. Only the first is secret.

> The rule: multiply by the chosen number and after every step keep only the remainder after dividing by 33.

Signing the message "4". We take 4 as many times as the private key says, that is, 7:

How many fours	1	2	3	4	5	6	7
Remainder	4	16	31	25	1	4	**16**

Each step is the previous remainder multiplied by 4. For example, the third: 16 × 4 = 64, and the remainder after dividing by 33 is 31. After the seventh step we have the signature: **16**.

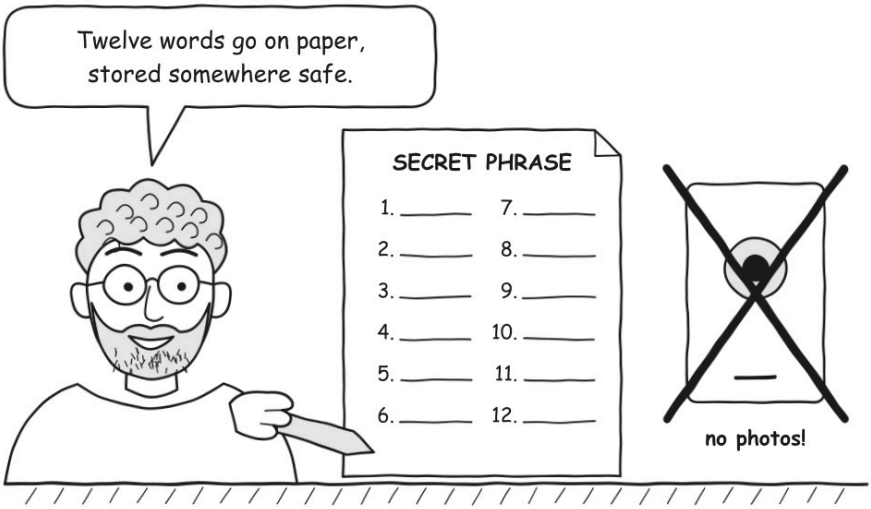
Verifying. We take the signature 16 as many times as the public key says, that is, 3: 16, then 16 × 16 = 256, remainder 25, then 25 × 16 = 400, remainder **4**. We got our message back. The signature is genuine!

Now someone has changed the message to "5", leaving the signature 16. The check still gives 4, not 5. The forgery shows.

This is only the idea. A schoolkid could crack numbers this small. In real networks the keys are enormous numbers, and the math is different.

The secret phrase

A private key is awkward to copy out: it is a long string. So many wallets give you a **secret phrase** of 12 or 24 words. The keys and all the addresses can be restored from it.



- Whoever knows the phrase owns the money.
- Lose both the phrase and the device and there is nobody to restore them.
- Don't photograph it, don't keep it in notes or chats.
- Don't type it into websites or dictate it to "support". Real support never asks for it.

You can change a password, but not a private key. If it has leaked, all you can do is urgently move the money to a new address.

What this means for a shop

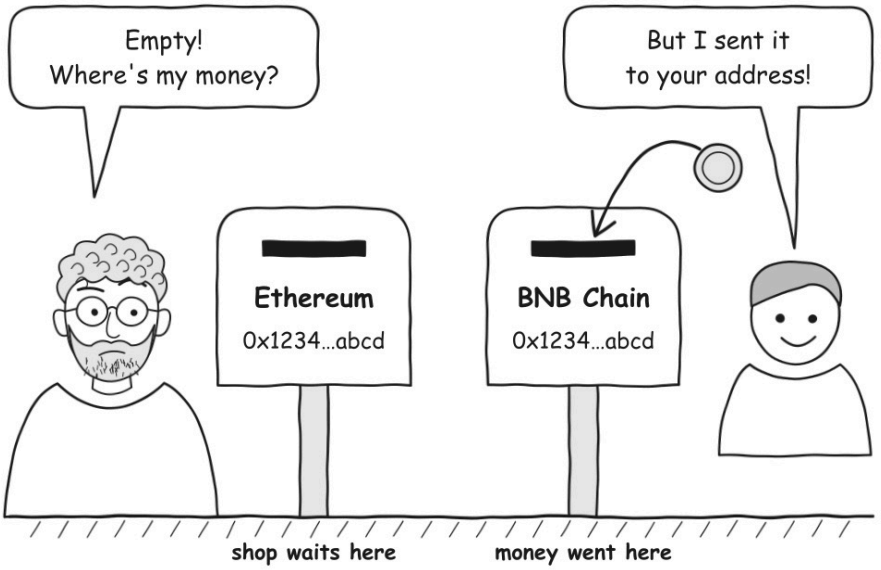
To **receive** money and see payments, Artyom's website needs only the address. The private key isn't needed on the shop's server, and it shouldn't be kept there: if the website is hacked, the money goes too. It is handy to give every order its own address; more on that in Chapter 28.

The trap: "same address, so it'll get there"

Addresses in different networks look different:

Network	Address starts with
Bitcoin	`1`, `3` or `bc1`
Ethereum and similar networks	`0x`
TRON	`T`
TON	usually `EQ` or `UQ`
Solana	no special prefix

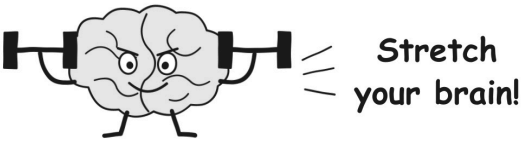
But on Ethereum, BNB Smart Chain, Polygon and many other networks an address looks **the same**. Their notebooks, though, are different. If a customer sends on the wrong network, the shop won't see the payment.



That is why an invoice states not only the address and the amount but also the **network**.

And one more scammers' trick: they send a tiny transfer from an address that resembles one you know. They are counting on you to copy it from your history. Compare the whole address, not just the first and last characters.

Think it over



1. Sign the message "2" with key 7 and verify the signature with key 3.
2. Artyom printed the shop's address on his business card. Is that dangerous?
3. "Wallet support" asks for your secret phrase to return a stuck transfer. What do you do?

Answers: 1) the signature is 29, and verification gives 2 again. 2) Money can't be stolen that way, but everyone will see what arrives at this address. 3) Tell them nothing: they are scammers.

Key points

- Show the address to everyone; the private key and the secret phrase to no one.
- A signature proves that the owner of the key authorized the transfer and that it hasn't been changed.
- The network matters as much as the address.

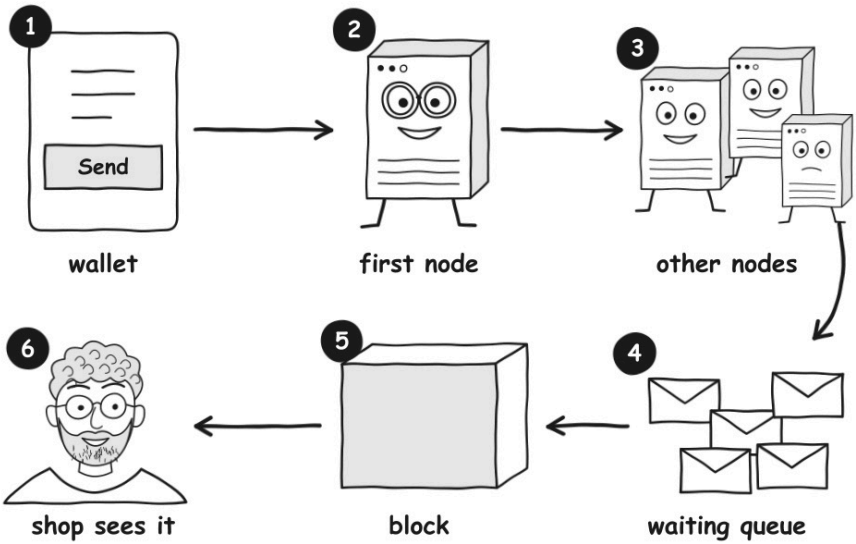
Next: Chapter 7. The transfer is signed. What happens to it between the "Send" button and being written into a block?

Chapter 7. A transaction's journey: from the "Send" button to a block

"I sent it!" — "I've got nothing"

The customer pressed the button and writes: "Sent." Artyom looks — no money. Is someone cheating? Most likely no one is. The transfer is still on its way.

Digging in: six stops



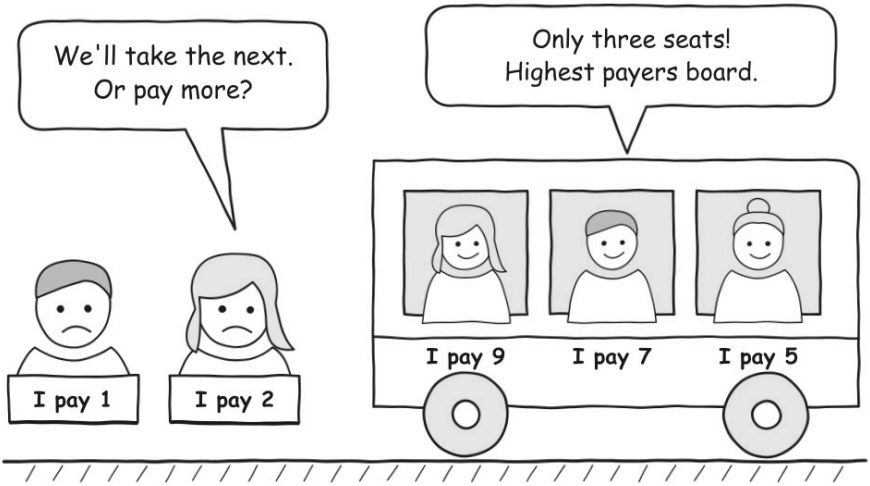
1. The **wallet** draws up the transfer and signs it with the key.
2. The **first node** receives the transfer and checks: is the signature valid, is there enough money.
3. It passes it on to **other nodes**, and they pass it further.
4. The transfer waits in a **queue**. It is called the mempool.
5. Whoever's turn it is to assemble a page puts the transfer into a **block**.
6. The **shop** sees it and waits for confirmations.

Names and details differ between networks: some, for example, have no common queue, and a transfer goes straight to whoever is assembling the block.

Every transfer has its own "seal" — the **transaction hash**. It is like a parcel's tracking number: with it you can find out where the transfer is now.

The fee: seats on a bus

Not everything fits into a block. Picture a bus with three seats and five people at the stop.



To get into a block, the sender pays a network **fee**. The more people want in, the pricier the seat. Whoever offered too little waits for the next bus, and sometimes it never comes.

- The sender pays the fee, and the shop doesn't get it.
- It depends not on the amount transferred but on the transfer's "weight" and how busy the network is.
- Each network counts that "weight" its own way: that's part four of the book.

Look at a real transfer

This is an ordinary transfer on the Ethereum network, block 26,502,085. A node was asked about it by transaction hash:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,
       "method":"eth_getTransactionReceipt",
       "params":["TRANSFER_HASH"]}'
```

The response is long. Here are the main points, converted into ordinary numbers:

Field	Value	What it means
`status`	`0x1`	Executed successfully
`blockNumber`	26,502,085	Which block it is in
`gasUsed`	21,000	The “weight” of a simple transfer
fee	0.000043 ETH	“Weight” × price per unit
amount transferred	0.000041 ETH	Taken from the transfer itself

Noticed? This time **more was paid for sending than was sent**. The fee doesn’t care how much money is inside.

And this number changes all the time. Ask for yourself:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_gasPrice","params":[]}'
```

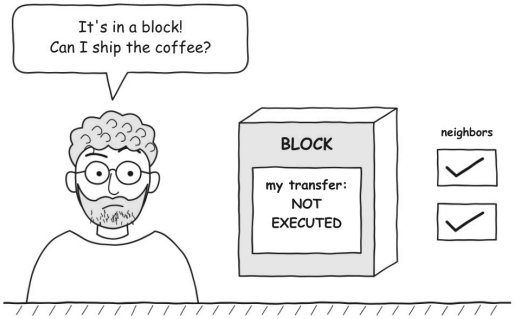
The node will name the current price of one unit of “weight” in wei. And on RpcNode’s public Bitcoin node that same day, almost 83,000 transfers were waiting in the queue.

What can happen to a transfer

State	What it means	What the shop should do
Pending	Still in the queue	Wait, don’t ship the goods
In a block	Recorded, but with few confirmations	Check it succeeded and wait
Confirmed	Enough blocks have piled on top	Ship the order
Failed	In a block, but the transfer didn’t happen	Don’t count it as payment
Dropped from the queue or replaced	Never made it into a block	Ask the customer to check their wallet

The trap: “it’s in a block, so it’s paid”

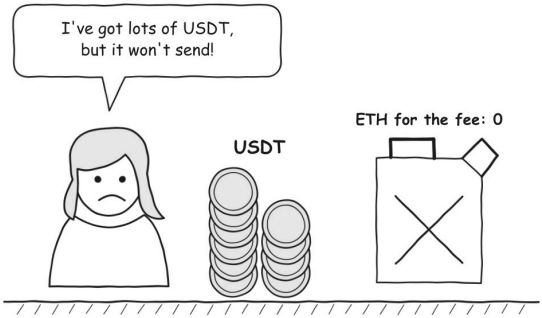
In Ethereum and similar networks a transfer can get into a block and still fail. For example, the customer didn’t have enough tokens. The fee is charged anyway, and the money doesn’t reach you.



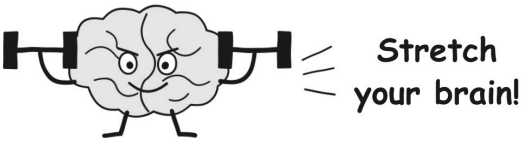
So you look not only at the block but also at the result: in the example above it is the `status` field. And you also check the recipient, amount, coin and network.

A second trap: the empty tank

The fee is charged in the network's native coin. In Ethereum that is ETH, even if you are sending USDT. In TRON you pay in TRX and the network's special resources.



Think it over



1. A block holds three transfers. The queue has offered 4, 9, 1, 7 and 2 for a seat. Who rides in the first block?
2. A customer sent a transaction hash and a screenshot saying “success”. Is that enough to ship the coffee?
3. A customer has 100 USDT on Ethereum and not a drop of ETH. What should they do?

Answers: 1) 9, 7 and 4. 2) No: a screenshot can be faked, check the hash on your side. 3) First get a little ETH for the fee.

Key points

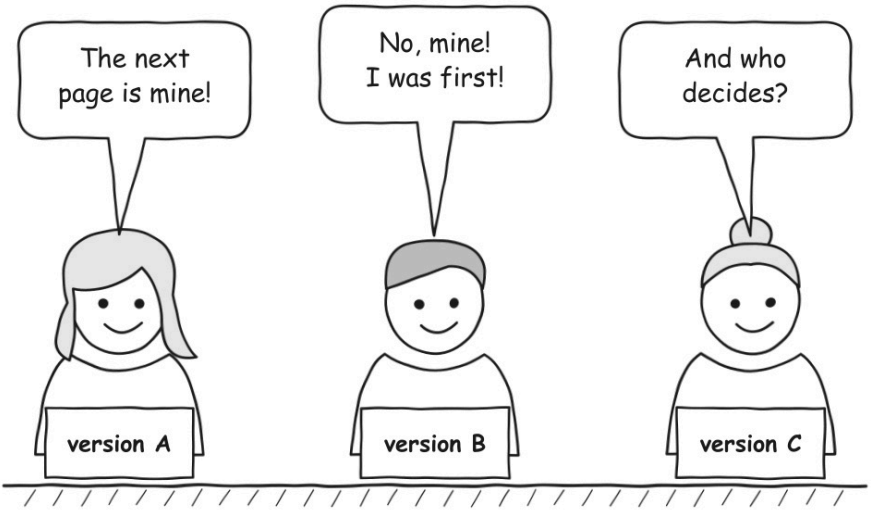
- Between “sent” and “received” there is a journey: node, queue, block, confirmations.
- The fee buys a place in a block and doesn’t depend on the amount.
- A payment is a successful transfer of the right coin on the right network with enough confirmations.

Next: Chapter 8. Who assembles the blocks, and how does everyone agree that this particular chain is the right one?

Chapter 8. How a network reaches agreement

Who writes the next page?

Nobody is in charge of the shared notebook. Thousands of participants have copies, and each can propose their own next page. Which one counts as the real one?



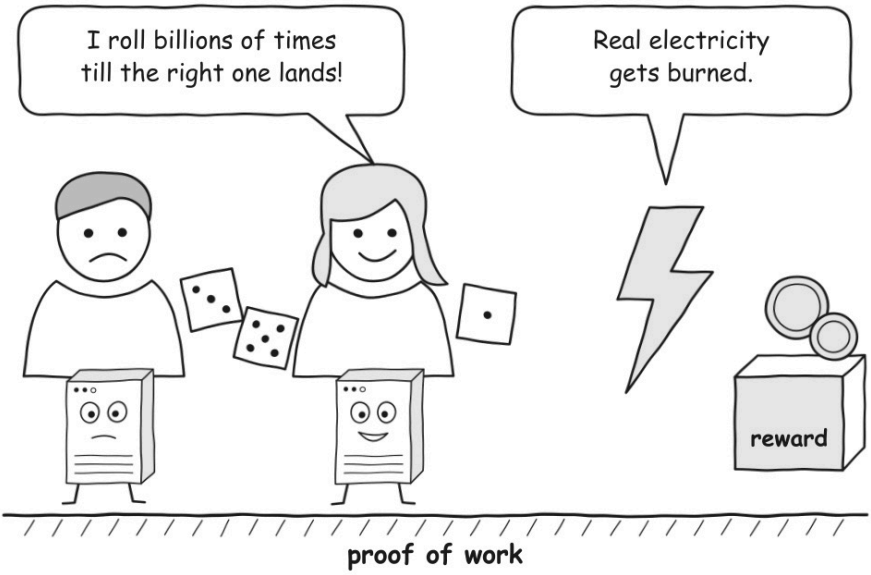
Digging in: consensus

The rules by which participants arrive at a common answer are called **consensus**. Their job is to make cheating more expensive than playing fair.

You can't simply count votes: a cheater would start a million programs and get a million votes. So a vote has to cost something. There are two main ways.

Method 1: proof of work

It is called Proof of Work, and the process itself is called **mining**. Participants race to solve a puzzle that can't be cracked by cleverness, only by trying one option after another. It is like rolling dice until a rare combination comes up.

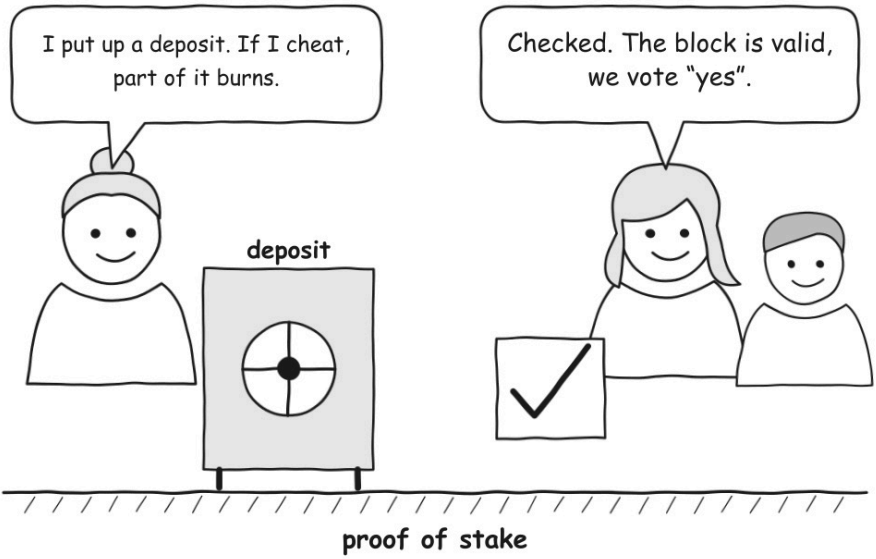


The winner adds a block and gets a reward: new coins and fees. Checking the answer is easy; finding it is expensive. That is how Bitcoin works:

- a block appears on average once every 10 minutes;
- every 2016 blocks the difficulty of the puzzle is adjusted to keep that pace;
- a block currently brings 3.125 BTC in new coins, and roughly every four years the reward is halved.

Method 2: proof of stake

It is called Proof of Stake, or **staking**. Instead of electricity, a participant puts up coins as a deposit. Such a participant is called a **validator**. Validators take turns proposing blocks, and the others check and vote.

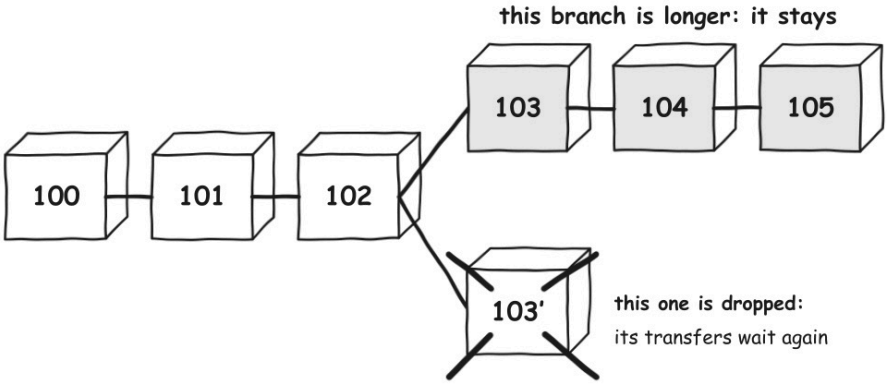


For a serious violation part of the deposit is burned; this is called slashing. Ethereum switched to this method on September 15, 2022, and the network's electricity use fell by about 99.95%. To become a validator yourself you need 32 ETH.

There are variations too. In TRON, blocks are written by 27 elected representatives chosen by coin holders. BNB Smart Chain also runs with a limited circle of validators.

When there are two pages at once

The network is large, and news doesn't travel instantly. Sometimes two blocks with the same number appear, and the chain forks for a while. Then one branch wins and the other is thrown away. This is called a **reorganization**.



A transfer from a discarded block doesn't vanish without a trace: usually it goes back into the queue and ends up in another block. But it may not. That is what the confirmations from Chapter 5 are for.

In the picture the longer branch wins. More precisely, Bitcoin picks the one with more work put into it, and staking networks pick the one backed by more votes of stake.

Finality: when it can no longer be undone

Finality is the moment after which a block won't be replaced. In some networks it is declared explicitly; in others confidence simply grows with every block.

Network	Roughly when a record is considered final
Bitcoin	No explicit mark. For large sums 6 confirmations are advised, about an hour
Ethereum	Usually 13–15 minutes, sometimes longer
TRON	Usually about a minute
BNB Smart Chain	About a second or faster
Solana	After the 2026 upgrade, designed for fractions of a second
TON	About a second

Data as of November 27, 2026, from the networks' documentation. These numbers change with upgrades.

Try it: ask for three kinds of "fresh"

Ethereum nodes can be asked for more than just the newest block. The word `latest` from Chapter 1 has two relatives: `safe` (already reliable) and `finalized` (final).

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,
       "method":"eth_getBlockByNumber",
       "params":["finalized",false]}'
```

Replace `finalized` with `safe` and `latest` and compare the numbers. While this chapter was being written, the node answered like this:

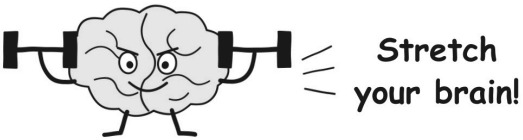
What was asked	Block number	Lag behind the newest
`latest`	26,502,104	—
`safe`	26,502,070	34 blocks, about 7 minutes
`finalized`	26,502,038	66 blocks, about 13 minutes

The trap: "it's the same in every network"

"I wait for one confirmation and that's it" is a dangerous rule. One confirmation means very different reliability in different networks. Decide by network and by amount:

- the pricier the order, the longer you wait;
- for large sums, wait for finality where it exists;
- don't hand over something you can't take back on the strength of a transfer from a fresh block.

Think it over



1. Why can't the notebook be run on "one program, one vote"?
2. Bitcoin blocks come once every 10 minutes. How long on average will 6 confirmations take? And 3?
3. One customer takes a cup of coffee to go, another takes 200 kg of beans. With whom should Artyom wait for finality?

Answers: 1) programs cost nothing; you can run as many as you like. 2) About an hour and about half an hour. 3) With the one taking 200 kg.

Key points

- Consensus is the set of rules by which participants with nobody in charge choose one chain.
- A vote costs either work (mining) or a deposit (staking).
- Fresh blocks can be replaced; finality comes later, and differently in each network.

Next: Chapter 9. Coins, tokens and smart contracts: why USDT is not the same thing as ETH.

Chapter 9. Coins, tokens and smart contracts

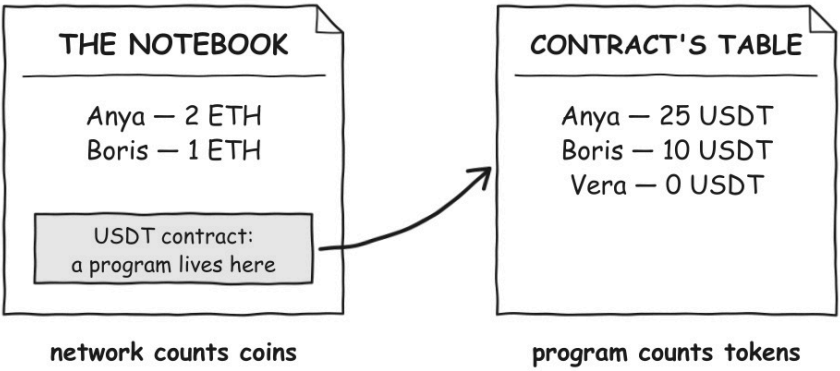
"I paid, and you've got nothing"

A customer sent 25 USDT on Ethereum. Artyom asks a node, as in Chapter 1, but the balance hasn't changed. Is the money gone? No. He simply needed to ask differently.

Digging in: coin and token

A **native coin** is one the network itself keeps track of: ETH in Ethereum, TRX in TRON, BNB in BNB Smart Chain. Fees are paid in it.

A **token** is a set of records kept by a program living inside the network. USDT is a token.

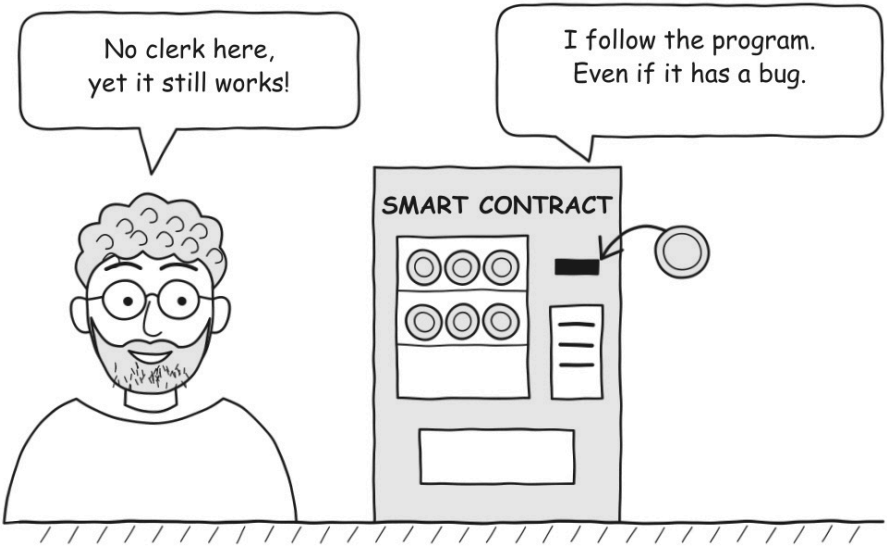


	Native coin	Token
Who keeps the records	the network itself	a smart contract
Example in Ethereum	ETH	USDT, USDC
What fees are paid in	the coin itself	the network's native coin
How to ask for a balance	`eth_getBalance`	`eth_call` to the contract

Tokens are built differently in different networks. Here we look at Ethereum and the networks like it.

A smart contract: a machine with no clerk

A **smart contract** is a program written into the blockchain. It is like a vending machine: you put something in, press a button and get a result by rules set in advance. Every node runs it the same way.



- It isn't an agreement with signatures, it is literally a program.
- Usually it can't be changed, but some authors keep that ability for themselves.
- The program may have an "owner" with special rights.
- A bug in the program can cost money, and there is nobody to roll it back.

Try it: a token balance

The `eth_call` method means "ask the contract without changing anything". The network charges no fee for such a question: nothing is written into the notebook. Let's ask the USDT contract how much USDT sits on its own address:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[
  {"to":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "data":"0x70a08231000000000000000000000000dac17f958d2ee523a2206206994597c13d831ec7"},
  "latest"]}'
```

`to` is which contract we are asking. And `data` is glued together from three pieces:

Piece	What it means
`0x70a08231`	which question: "what's the balance?" (`balanceOf`)
24 zeros	padding to the required length
40 characters	whose balance: the address without `0x`

While this chapter was being written, the node answered like this:

```
{"jsonrpc":"2.0","id":1,"result":"0x000000000000000000000000000000000000000000000000
014d0a9aa6c5"}
```

After the zeros comes `14d0a9aa6c5` , which is 1,430,402,016,965. USDT on Ethereum has six decimal places, so the contract's address holds **1,430,402.016965 USDT**.

How did it get there? By all appearances, mostly from transfers sent to the wrong place: someone pasted the contract address instead of the recipient's. The sender has no ordinary way to get it back.

Where the decimal point goes

A contract stores only whole numbers. How many digits to split off is stated by a separate property, `decimals` .

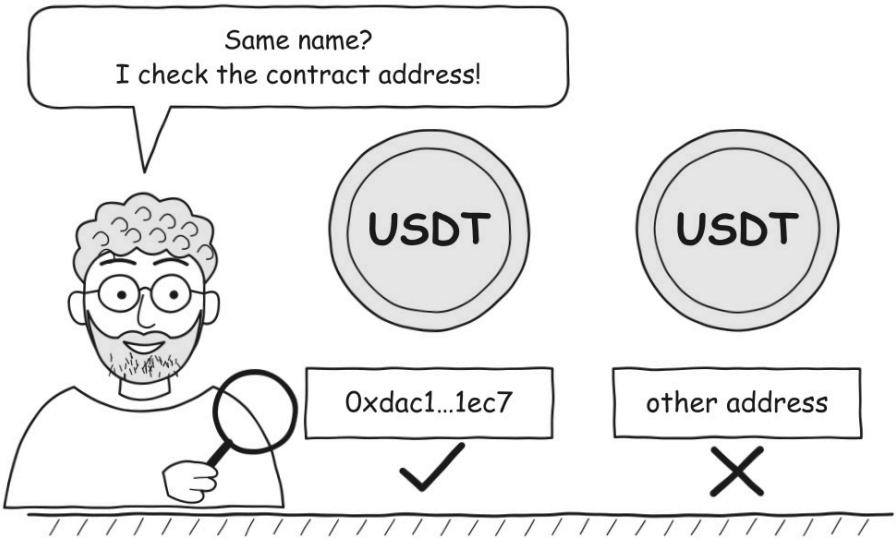
What	Decimal places
ETH, the native coin	18
USDT on Ethereum	6
the "Tether USD" (USDT) token on BNB Smart Chain	18

Same name, three times as many digits. Mix them up and you are off by a factor of a trillion. Besides, the address of this BNB Smart Chain token is not among the USDT contracts on Tether's website: the same name doesn't mean the same issue.

Checked on November 27, 2026 with requests to public nodes and against the page tether.to/en/supported-protocols.

The trap: a fake token

Anyone can call their token "USDT". A name proves nothing. Real USDT is recognized by a pair: **the network and the contract address**.



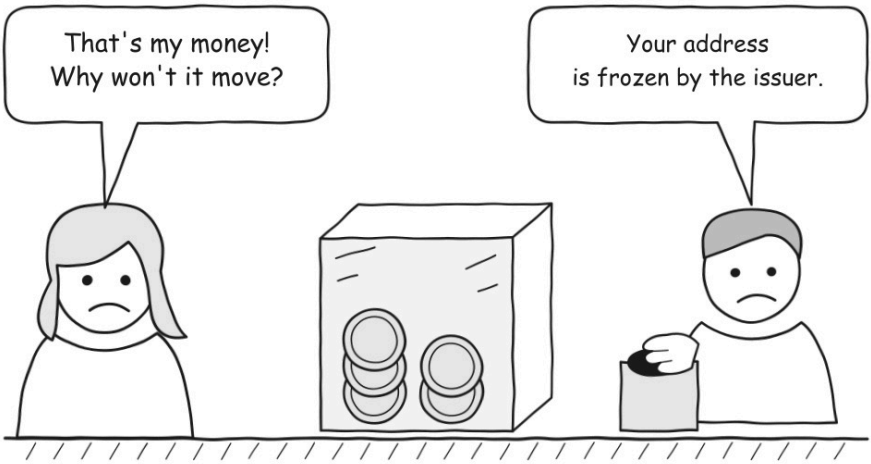
So a shop checks everything at once:

- the network;
- the token contract address;
- the recipient;
- the amount with the right number of decimals;
- success and confirmations.

Stablecoins: convenient, but someone is in charge

A **stablecoin** tries to stay close to one dollar. For USDT that is the issuing company's promise, not a law of nature.

The issuer also has power: Tether's terms of service state outright that it has the right to freeze tokens (version of February 26, 2026, tether.to/en/legal).

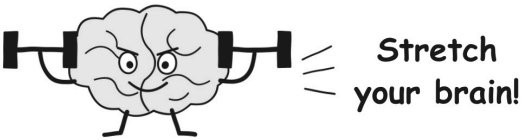


A second trap: "allow spending"

A website may ask you to sign an **allowance** (`approve`): "this contract may spend my tokens". If you grant a scammer one with no amount limit, they will take the tokens later, without a new signature. They don't need to know your secret phrase.

> **The rule:** if you don't understand what you are signing, don't sign.

Think it over



1. For USDT on Ethereum a node returned the number 25,000,000. How much USDT is that?
2. A customer sent a token named USDT, but with a different contract address. Is the order paid?
3. Artyom has 100 USDT on Ethereum and 0 ETH. Can he send it to his supplier?

Answers: 1) 25. 2) No. 3) No: there is nothing to pay the fee with, see Chapter 7.

Key points

- The network keeps track of a coin; a smart contract keeps track of a token.
- A token is recognized by network and contract address, not by name.
- An amount is read with its decimal places taken into account.

Part two is over. Artyom knows what a block, a key, a fee and a token are. But a node is still a black box to him. **Next:** Part 3, Chapter 10. What kinds of nodes there are.

Chapter 10. What kinds of nodes there are

"And how much was there back then?"

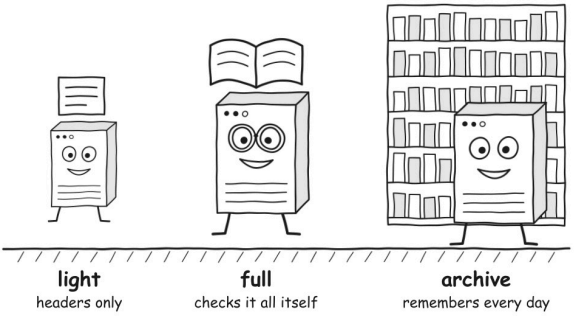
Artyom is reconciling his books and wants to know how much was on the shop's address several months ago. The node answers beautifully about today, but refuses to talk about the past. Doesn't a blockchain keep everything?

Digging in: records and balances

Two different things are hiding here:

- **History** is the records themselves: blocks and transfers.
- **State** is who owns what at a particular moment: balances and contract data.

It is like at a bank: there is a list of operations, but the balance for every minute of last year isn't stored separately. It would have to be recalculated. Nodes differ in how much of all this they remember.

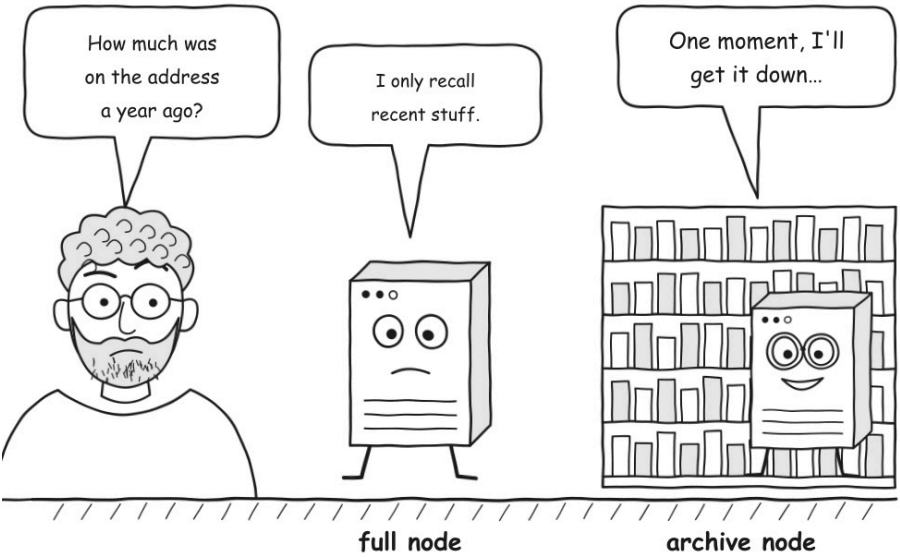


Three main kinds

A **full node** checks blocks against the network's rules and keeps the recent state. In Ethereum that is usually about the last 128 blocks, or roughly 25 minutes. The exact depth depends on the software and settings.

An **archive node** checks everything from the very beginning and doesn't delete what it has downloaded. It remembers the state at any block, but needs far more disk.

A **light node** downloads only block headers and asks others for the details. A good idea, but as of November 27, 2026 Ethereum has no light node ready for everyday use.



The pruned node: checked it and threw it away

Bitcoin has a disk-saving mode. A **pruned node** checks everything and later deletes old blocks. It remains a fully fledged full node; it just can't hand over what it has erased. According to the Bitcoin Core developers, 10 GB is enough for it.

Ethereum has gained a similar option too: since July 2025, node software can skip storing the contents of the oldest blocks — those from before the switch to staking in September 2022. It is a setting, and it isn't turned on everywhere. So even a "full" node may not remember the really old records.

Try it: ask about the past

Let's ask for the balance of the same address twice. First, as of now:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance",
  "params":["0xdac17f958d2ee523a2206206994597c13d831ec7","latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"result":"0x2a"}
```

`0x2a` is 42 wei, mere crumbs. Now replace `"latest"` with `"0xe4e1c0"`, that is, block number 15,000,000. The public node will answer like this:

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,
 "message":"archive data is not available on the public endpoint,
 use a personal endpoint"}}
```

In plain words: old data isn't served on the public endpoint; you need a personal one. It isn't a breakdown, just the boundary of a free service. RpcNode offers archive access on paid plans; Free doesn't have it (Chapter 3).

How much it weighs

Node	Disk, roughly
Bitcoin, pruned	10 GB or more
Bitcoin, full	about 900 GB
Ethereum, full	about 2 TB or more
Ethereum, archive	from 2 to 14 TB and more, depending on the software

Ethereum data is from the ethereum.org page on running nodes, as of November 27, 2026.

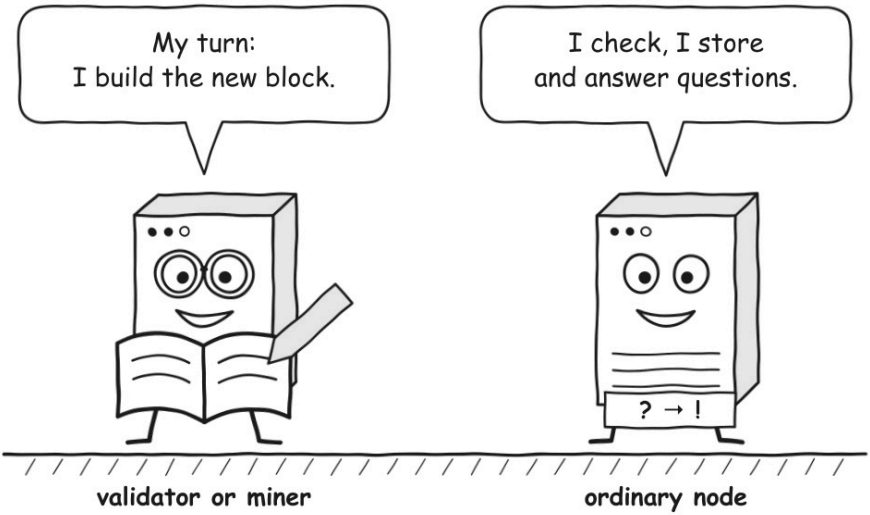
You can check the second row yourself: a Bitcoin node honestly reports its size.

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getblockchaininfo","params":[]}'
```

Find two lines in the long response. While this chapter was being written, they were: `"size_on_disk":896027659764` (about 896 GB including service files) and `"pruned":false` — the node isn't pruned.

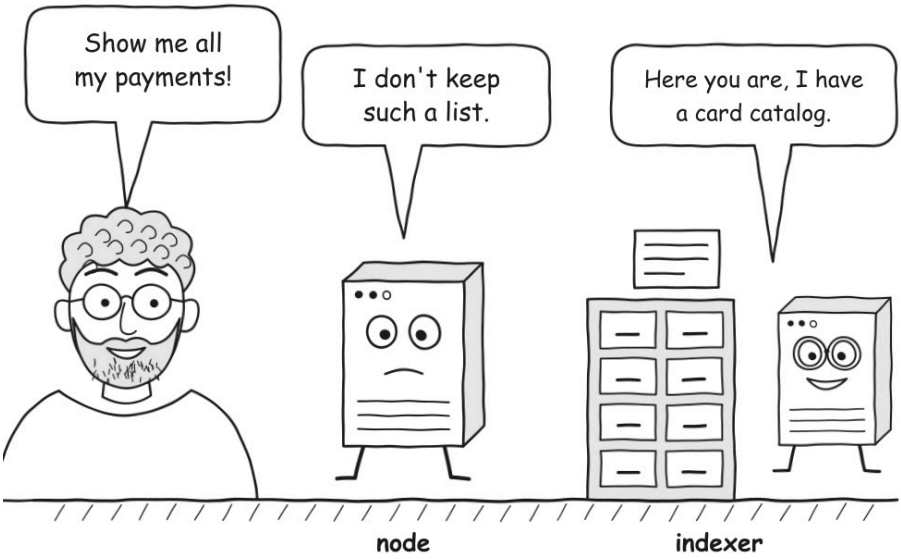
And who writes the blocks?

That is a separate role. A **validator** or **miner** takes part in creating blocks (Chapter 8). You don't need to be one to check payments: a node that reads, checks and answers is enough.



The trap: "I'll ask the node for all my payments"

The standard set of questions to an Ethereum node has nothing like "show all transfers of this address". Even an archive node can't do that. The `eth_getTransactionCount` method will only name the number of transactions **sent** from an address.

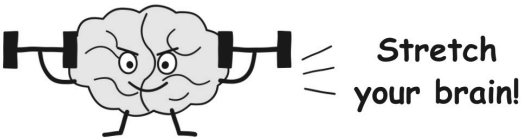


For such lists you need an **indexer**: a program that reads blocks and files the records into a card catalog. Blockchain explorer websites work this way too. A shop finds it easier to record its own payments from the start and watch for new ones; see Chapter 26.

What your task needs

Task	What fits
Check a recent payment	an ordinary full node
Find a balance on a past date	an archive node
Get a list of all transfers of an address	an indexer
Take part in creating blocks	a validator or miner

Think it over



1. Artyom wants to chart the shop's balance over a year, one point per day. Which node does he need?
2. A pruned Bitcoin node has deleted old blocks. Can it be trusted to check new ones?
3. What does it take to show a customer all their past orders with payments: an archive node or the shop's own database?

Answers: 1) an archive node. 2) Yes, it checked everything before deleting. 3) The shop's own database: a node keeps no such lists.

Key points

- History is the records; state is the balances at a particular moment.
- A full node remembers recent state, an archive node any state.
- A node doesn't give you the list of an address's transfers; for that you need an indexer.

Next: Chapter 11. We have already asked a node a dozen questions. Time to sort out what language we use: JSON-RPC, REST, gRPC and WebSocket.

Chapter 11. How to talk to a node

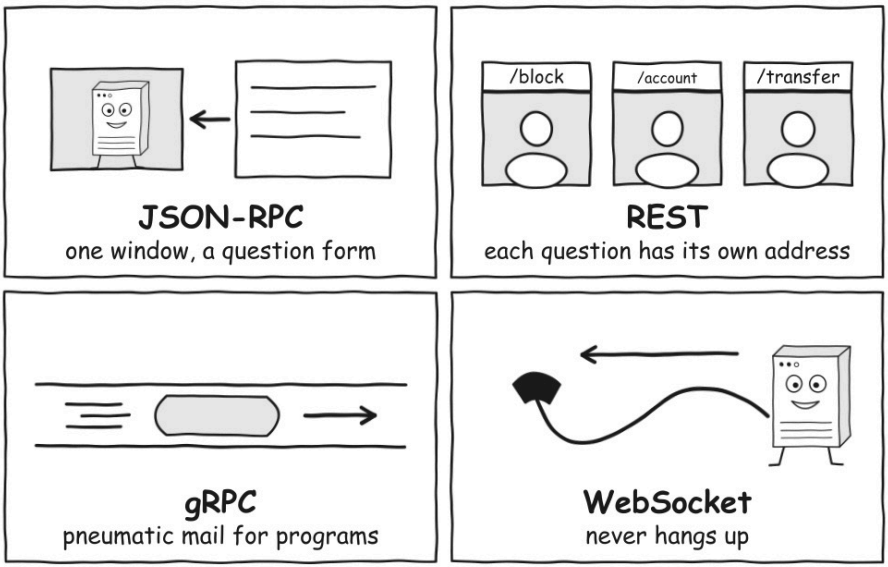
What are all those letters in the requests?

Artyom has copied commands from this book a dozen times by now, and they worked. Time to understand what is written in them: what RPC is, why there are so many brackets, and why requests look different in different networks' documentation.

Digging in: what API and RPC are

An **API** is a service window through which one program addresses another by rules known in advance. **RPC** (remote procedure call) is a way to ask a program on another computer to do something as if it were right next to you. A node accepts such requests.

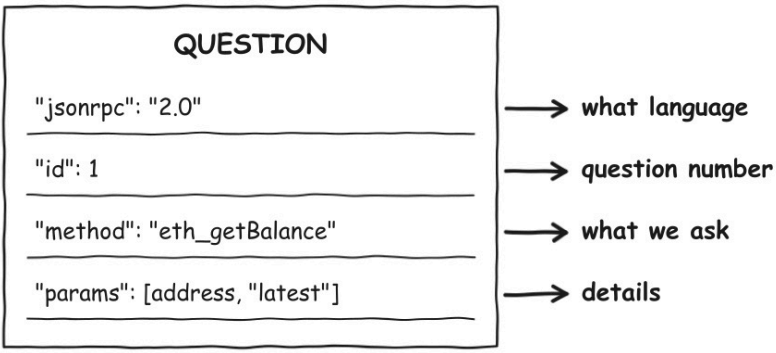
There are several ways of talking:



Way	What it looks like	Where you will meet it
JSON-RPC	One address; the request names a method and parameters	Ethereum and similar, Solana, Bitcoin
REST	Each question has its own path in the address	TRON, TON, Aptos
gRPC	Compact binary messages following a strict schema	Exchange between programs, for example in Sui
WebSocket	A connection that stays open	When the node should report news by itself

WebSocket isn't a language but a **communication line**: in Ethereum the same JSON-RPC messages travel over it. Some networks offer several ways at once.

JSON-RPC, bone by bone



JSON is a simple way to write down data: curly braces, names in quotes, and values. Here is the command from Chapter 3, taken apart line by line:

Piece of the command	What it means
`curl`	a program that sends a request from the terminal
`https://…`	the endpoint address
`-H '…'`	a label saying "JSON inside"
`-d '{…}'`	the question itself

In the response you look at `result` (what came out) and `error` (what didn't work). Some nodes send only one of these fields, others both, and then the empty one is `null`. The `id` in the response is the same as in the question: that is how they are matched.

Several questions at once

Questions can be put into a list in square brackets. This is called a **batch**:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '[{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]},
       {"jsonrpc":"2.0","id":2,"method":"eth_chainId","params":[]}]'
```

```
[{"jsonrpc":"2.0","id":1,"result":"0x1946420"},
 {"jsonrpc":"2.0","id":2,"result":"0x1"}]
```

The first answer is block number 26,502,176, the second is the network number: for Ethereum it is 1. By the JSON-RPC rules, answers in a batch may come in any order, so look at the `id`. And remember: a batch saves trips, not quota. In RpcNode a batch of ten questions counts as ten requests, and it is accepted or rejected as a whole.

And here is REST

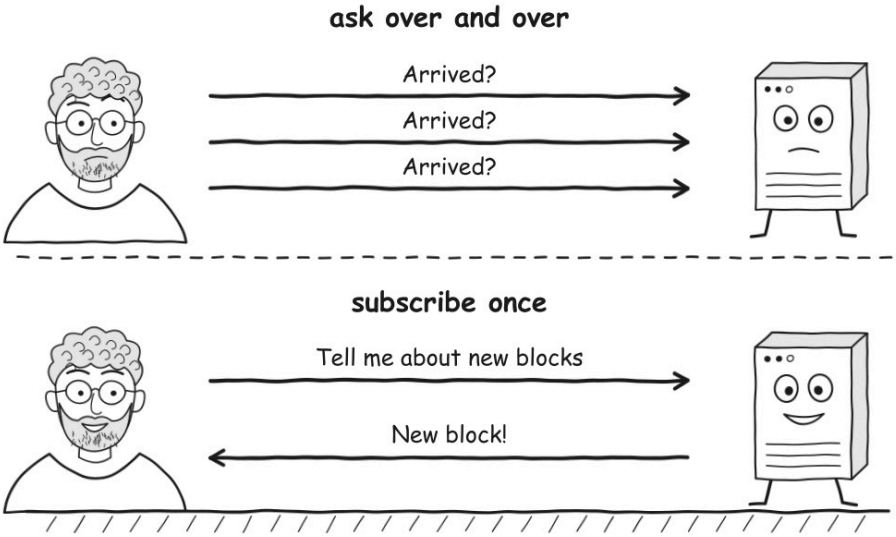
Here the question is built into the address itself. Let's ask the TRON network for the latest block:

```
curl https://public-tron-mainnet.rpcnode.dev/wallet/getnowblock
```

A long response about a block with transfers will arrive. While this chapter was being written, it was block number 88,351,889, with 194 transactions in it. A different question means a different path after the address.

WebSocket: don't ask, listen

Asking "has it arrived?" every second is wasteful. Over an open line you can subscribe once, and the node will send the news by itself.



RpcNode's public nodes don't answer over WebSocket; you need a personal endpoint. Its address starts with `wss://`. Here is a sketch in JavaScript:

```
const ws = new WebSocket('wss://YOUR_ENDPOINT')

ws.onopen = () => ws.send(JSON.stringify({
  jsonrpc: '2.0', id: 1,
  method: 'eth_subscribe', params: ['newHeads'],
}))

ws.onmessage = (event) => console.log(event.data)
ws.onclose = () => console.log('connection dropped')
```

Lines break. A program must be able to reconnect, subscribe again and check what it missed. Subscriptions use credits too.

gRPC

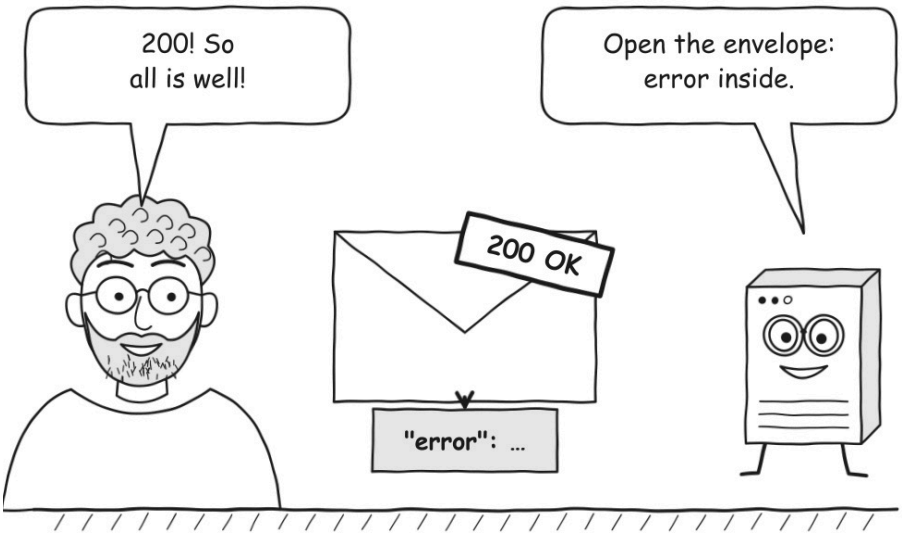
This is a way for programs that care about speed and a strict format. The messages can't be read by eye, and their fields are described in advance in a separate file. A beginner's shop usually doesn't need it. RpcNode doesn't offer gRPC on every network: see the endpoint's list of transports.

The trap: "200 means all is well"

200 is a delivery code: the envelope arrived. But there may be an error inside. Let's ask for the balance of the address "hello":

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance","params":["hello","latest"]}'
```

The response code is 200, and inside is an `error` with number `-32602` : invalid parameters.



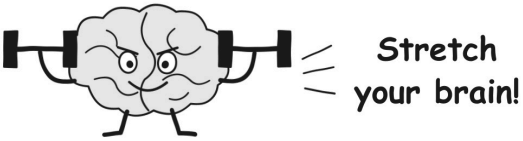
So a program always checks both the code and the `error` field.

Common errors

Code	What happened	What to do
`-32700`	The request text can't be parsed	Check the brackets and quotes
`-32600`	The request breaks the rules	Check the fields
`-32601`	Method not found or unavailable	Check the name and your plan
`-32602`	Invalid parameters	Check the values
`-32005` (HTTP 429)	RpcNode: too often	Wait and ask less often
`-32006` (HTTP 402)	RpcNode: not enough credits	Check your usage and plan

The first four codes are from the JSON-RPC 2.0 standard; the last two are RpcNode codes.

Think it over



1. In the first request of the batch, write `eth_blockNumbr` instead of `eth_blockNumber`. Which code will come back? Will the second, correct question be answered?
2. What is better for a "latest block" display that must update by itself: polling every second or a subscription?
3. A program got HTTP 200 and immediately marked the order as paid. What is wrong?

Answers: 1) `-32601`, and one for everything: RpcNode will reject the whole batch, and the second question goes unanswered too. 2) A subscription. 3) Neither the `error` field nor the transfer itself was checked (Chapter 7).

Key points

- JSON-RPC names the method in the request, REST in the address.
- WebSocket keeps the line open so the node can report news itself.
- Success isn't code 200 but a response without `error` and with what you expected.

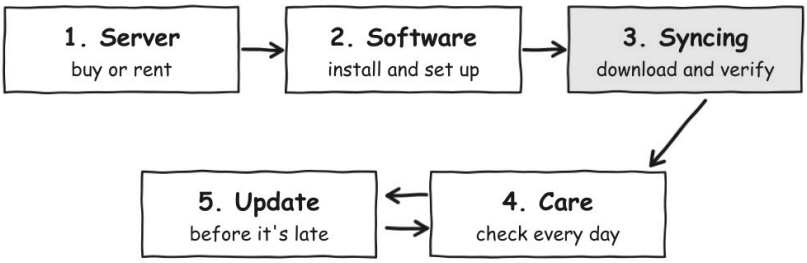
Next: Chapter 12. Let's look on the other side of the window: how your own node lives, what syncing is and why it breaks.

Chapter 12. Your own node from the inside: setup, syncing and breakdowns

"It's just one evening's work!"

A programmer friend assures Artyom: "Download it, start it, and you have your own node." Starting it really doesn't take long. What takes long is what begins afterwards.

Digging in: the life of a node

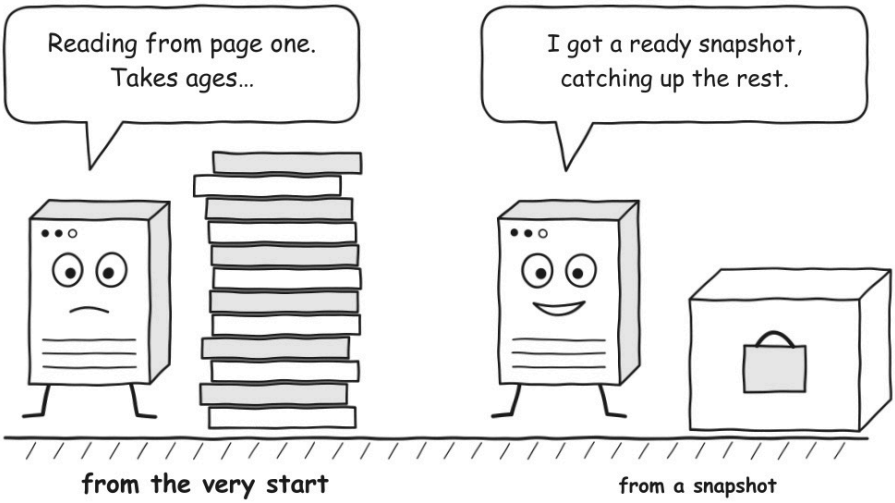


steps 4 and 5 repeat while the node lives

1. **Server.** We looked at the requirements in Chapter 2.
2. **Software.** Each network has its own, often with several to choose from. In Ethereum you need two at once: an **execution client** (carries out transfers and contracts) and a **consensus client** (tracks which chain is correct).
3. **Syncing.** The node catches up with the network.
4. **Supervision.** Has it fallen behind, is there enough space.
5. **Updates.** The network changes its rules — the software is changed in advance.

Syncing: catching up with the network

A new node knows nothing. Until it has caught up with the others, asking it about a balance is pointless: it will answer from the past.



- **From the very beginning.** The node checks every block itself. Reliable and slow. Bitcoin's initial download is about 740 GB.
- **From a snapshot.** You take a ready copy of the database and catch up on the rest. Faster, but you are trusting whoever made the snapshot.
- **In-between ways.** Ethereum software usually downloads the recent state and checks it against block headers.

In time this takes from several hours to many days, depending on the network, method, disk and connection.

Try it: a two-line configuration

This is what the beginning of a `bitcoin.conf` settings file might look like:

```
server=1
prune=5000
```

Line	What it means
`server=1`	accept RPC questions; unless otherwise allowed, only from this same computer
`prune=5000`	a pruned node (Chapter 10): delete old blocks, keep about 5 GB of them; service data takes space on top

This isn't a complete guide. Before installing, read your software's documentation.

How to tell the node has caught up

You can ask a node directly. In Ethereum:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_syncing","params":[]}'
```

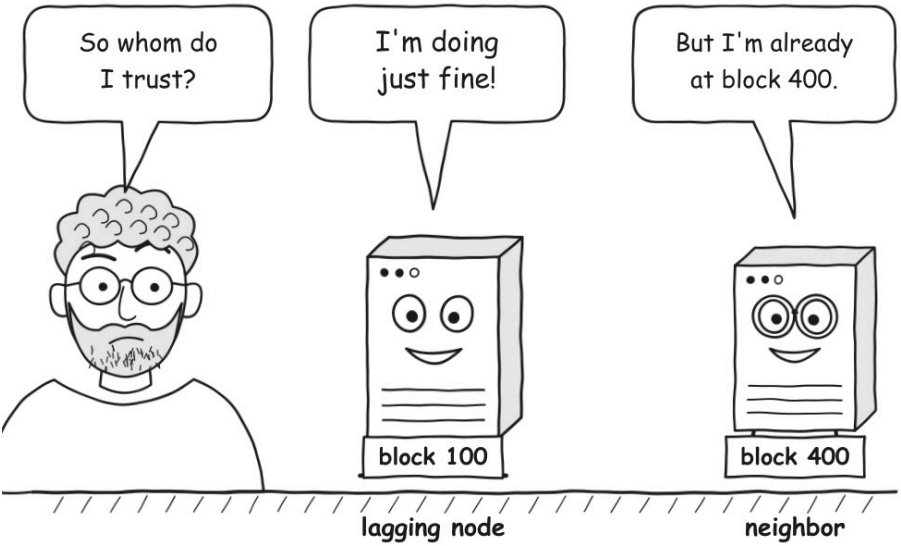
```
{"jsonrpc":"2.0","id":1,"result":false}
```

`false` means "I'm not catching up right now". While a node is catching up, numbers come back instead: where it is and where it is heading. For Bitcoin you look at the `getblockchaininfo` response from Chapter 10:

Field	Value when checked	What it means
`blocks`	977,715	how many blocks are verified
`headers`	977,715	how many blocks it knows about
`initialblockdownload`	`false`	the initial download is behind it

The trap: "it told me itself that all is well"

A node judges by what it knows. If it has lost contact with the others, it may have no idea it has fallen behind.



So you check two things: **how fresh the last block is** and **how it compares with another source**. Freshness is easy to find out. Ask for the time of the last block:

```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockByNumber",
  "params":["latest",false]}' | grep -o '"timestamp":"0x[0-9a-f]*"'
```

```
"timestamp":"0x6b08aacb"
```

These are seconds elapsed since 1970. Subtract them from the current time (put in your own number):

```
echo $(( $(date +%s) - 0x6b08aacb ))
```

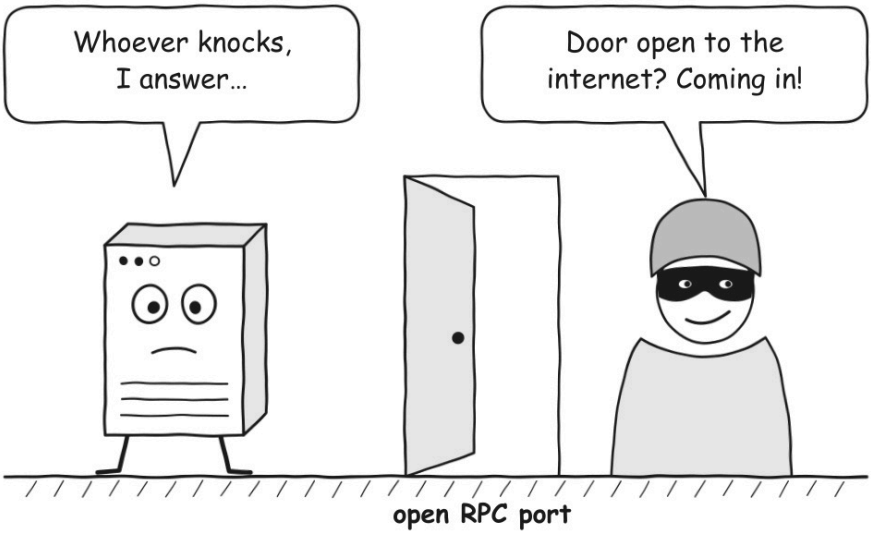
You get the block's age in seconds. While this chapter was being written it was 7 seconds: for Ethereum, with a block every 12 seconds, that is normal. But if your node shows thousands, it has fallen behind. Remember the question from Chapter 1 about a node an hour behind?

What breaks

What you see	Common cause	What to do
The node stopped	The disk ran out of space	Watch the space and expand in advance
Always lagging	Weak disk or connection, few peers	Check the hardware and network
"Stuck" after a network upgrade	An old software version	Update before the announced date
Won't start after a power failure	The database is corrupted	Restore from a copy or sync again
Slow for no visible reason	Other people's requests are loading it	Close access to outsiders

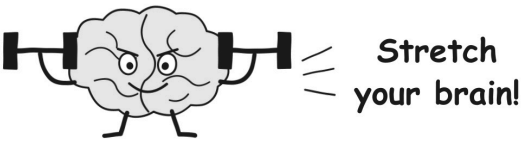
A second trap: the open door

Bitcoin Core and Geth by default accept RPC questions only from their own computer. That is for good reason. Open the port to the whole internet and you let in anyone at all, and if there is a wallet on the node, you are risking money.



- Allow access only from your own addresses (a firewall).
- Put a key check in front of the node.
- Don't keep a wallet with money on a node that serves your website.

Think it over



1. `eth_syncing` answers `false`, but the last block is 40 minutes old. Is the node healthy?
2. A 2000 GB disk has 1900 GB used, and 5 GB is added per day. In how many days will everything stop?
3. The website is on another server. Can you open the RPC port to everyone "so the website can ask"?

Answers: 1) No. 2) In 20 days at the latest, so start fixing it today. 3) No: allow only the website's address.

Key points

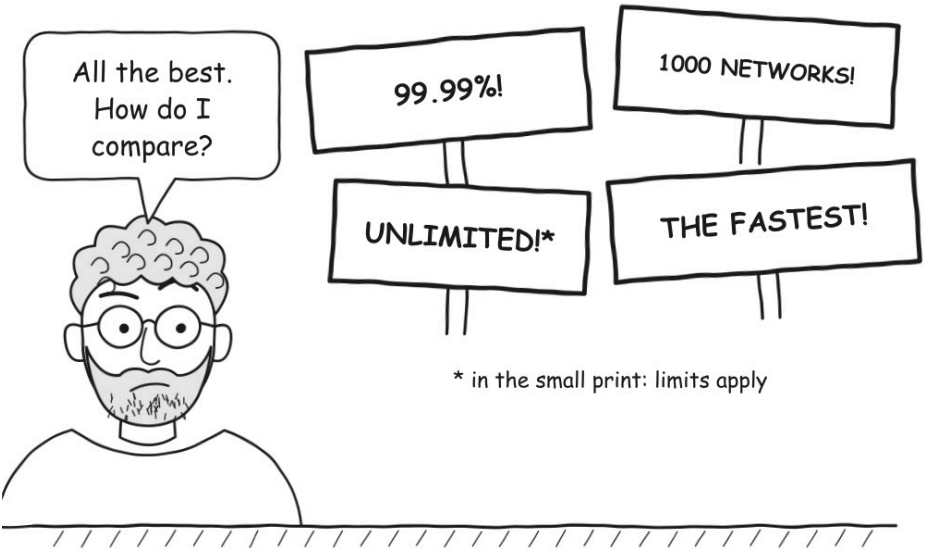
- Starting a node is the beginning of the work, not the end.
- Health is checked by block freshness and by comparing with another source.
- An RPC port is never opened to the whole internet.

Next: Chapter 13. If all this isn't your favorite pastime: how to choose a provider and what to ask them.

Chapter 13. How to choose a provider

Everyone promises the best

Artyom has decided: running his own node isn't for him. He opens providers' websites, and it is exclamation marks all the way.



Digging in: start with yourself

Comparing providers without knowing your own needs is pointless. First answer these:

- Which networks do you need, and do you need test ones?
- How many requests per second and per month?
- Do you need old history (an archive)?
- Do you need to learn about events right away (WebSocket, notifications)?

Count your requests

Artyom has 20 orders awaiting payment, and the website checks each one every 10 seconds:

> 20 orders × 6 times a minute = 120 requests a minute, that is 2 a second.
> Over 30 days: 5,184,000 requests.

Then multiply by the method's price in credits: different methods have different prices. Now think about whether you need to ask that often. Once every 30 seconds means a third as many requests.

A list of questions



Ask	Warning sign
Are **my networks** there? And the methods I need in them?	A long list of networks with no list of methods
Limits: requests per second and in total; what happens if you exceed them?	"Unlimited" with no numbers; vague extra charges
Archive, WebSocket, gRPC: on which plan and for which networks?	"We have everything", with no details
Node failure: will they switch you to another? Where are the servers?	No answer
Availability: what is promised, how is it measured, what happens if it is broken?	A percentage with no period and no terms
Protection: can you restrict addresses and websites? What is logged about requests?	The key works for anyone
Support: where do you write and in what language will they reply?	Only a faceless form
Leaving: are the methods standard? Can you pay month by month?	A long mandatory term

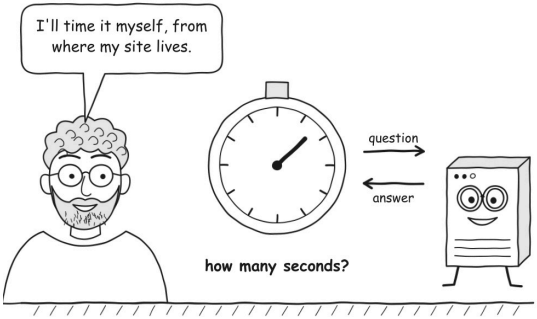
What a percentage hides

“99% availability” sounds almost like “always”. Let’s work out how much downtime that allows over 30 days:

Promised	May be down for
99%	7 hours 12 minutes
99.9%	about 43 minutes
99.99%	about 4 minutes

And ask **what exactly** counts as working: the provider’s website, or your endpoint on your network.

Try it: measure the speed yourself



Other people’s measurements are taken from another place. Run five requests from the server where your website will live:

```
for i in 1 2 3 4 5; do
  curl -s -o /dev/null -w '%{time_total} s\n' \
    https://public-ethereum-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
  sleep 1
done
```

You will see five numbers — response times in seconds. Don’t look only at the best one: a customer will get the worst one too. Here the connection is opened anew each time; a program with a persistent connection is usually faster. And compare not only speed but also block freshness (Chapter 12).

Ask us these questions too

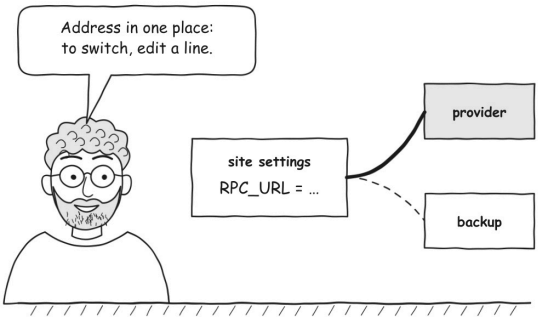
This book is about RpcNode, so let’s check it as well. Data as of November 27, 2026.

Question	RpcNode
Networks	25 listed on the website
Try for free	the Free plan and public nodes without a key
Limits	Free: 15 requests per second and 15 million credits; Starter: 40 and 50 million
If exceeded	no overage charges: requests are rejected (429 or 402, Chapter 11)
Archive	on paid plans
WebSocket and gRPC	through a personal endpoint, not for all networks
Where the servers are	Frankfurt, the USA, Japan
Protection	an endpoint can be restricted by IP addresses and websites
Privacy	the provider sees your requests (Chapter 3)
Availability and support	see the current terms at rpcnode.dev

The trap: choosing by the biggest number

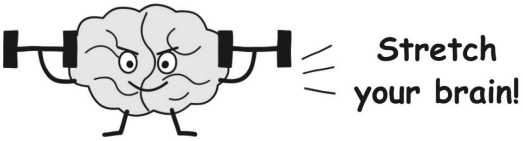
A thousand networks won't help if your two work badly. The lowest price won't help if the method you need is expensive or unavailable. Take a free or the cheapest plan and run your own real requests for a few days.

Don't get too attached



- Keep the endpoint address in a single setting, not in the code.
- Where possible, use the network's standard methods.
- Set up a fallback and switch to it once to test it.

Think it over



1. 20 orders, checked once every 30 seconds. How many requests in 30 days?
2. 99.9% was promised, and the endpoint was down for two hours in a month. Was the promise kept?
3. The endpoint address is written into 14 places in the code. Why is that bad?

Answers: 1) 1,728,000. 2) No: about 43 minutes is allowed. 3) When changing it, it is easy to miss a place, and a key leaks more easily from code.

Key points

- Work out your needs first, then compare.
- Test promises with your own measurements and your own requests.
- Leave yourself a way out.

Part three is over. Artyom knows what a node is, how to talk to it and who to get it from. But the networks are all different! **Next:** Part 4, Chapter 14. Bitcoin and its relatives.

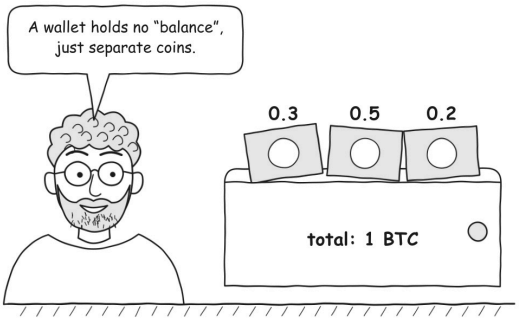
Chapter 14. Bitcoin and its relatives

"So where is its balance?"

A customer wants to pay in bitcoin. Artyom is experienced by now: he'll ask a node for the address balance, as in Ethereum. But a Bitcoin node has no such simple question. Why?

Digging in: not an account but coins

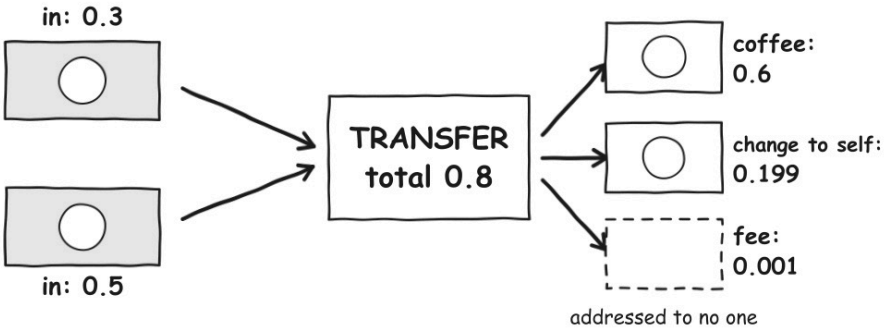
In Ethereum an address has a balance, like a bank account. In Bitcoin it doesn't. There are separate "pieces of money" there, like banknotes of different values.



Each such piece is called a **UTXO** (unspent transaction output). A balance is the sum of all your unspent pieces. A wallet or an indexer adds it up (Chapter 10).

A transfer: inputs, outputs and change

You can't tear off part of a banknote. You have to hand it over whole and get change. It is the same here: a transfer spends old pieces completely (**inputs**) and creates new ones (**outputs**).



- One output goes to the recipient.
- The second is **change** back to yourself.
- Whatever isn't written into any output goes to the miner. That is the **fee**: inputs minus outputs.

> 0.8 − 0.6 − 0.199 = 0.001 BTC in fees

The fee is priced not by the amount but by the size of the transfer: in **satoshis per virtual byte**. A satoshi is one hundred-millionth of a bitcoin. The more pieces go in, the bigger the transfer and the more it costs.

The very first transfer between people

On January 12, 2009, in block 170, a single 50 BTC piece turned into two: 10 BTC and 40 BTC. The recipient of the ten, programmer Hal Finney, told the story himself: Satoshi sent it to him as a test. The change, 40 BTC, went back to the sender. There was no fee.

There are no names in the blockchain: Finney's part is known from his own account (bitcointalk.org, 2013).

You can still request this transfer:

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getrawtransaction","params":
  ["f4184fc596403b9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16",true]}'
```

In the response look for `vin` (inputs, one here) and `vout` (outputs, two here: `10.0` and `40.0`).

Try it: has the coin been spent?

The `gettxout` method asks: "is this piece still intact?". A piece is named by the transfer's number and the output's number, counting from zero. Let's ask about those very 10 BTC:

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"gettxout","params":
  ["f4184fc596403b9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16",0]}'
```

```
{"jsonrpc":"2.0","result":null,"id":1}
```

`null` means there is no such unspent piece: it was spent long ago. And now the reward for the very first block after the genesis one:

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"gettxout","params":
  ["0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098",0]}'
```

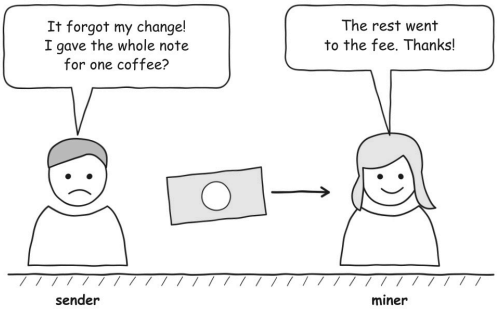
Here you get `"value":50.00000000` and almost a million confirmations. While this chapter was being written, nobody had touched those 50 BTC.

Three more facts about Bitcoin

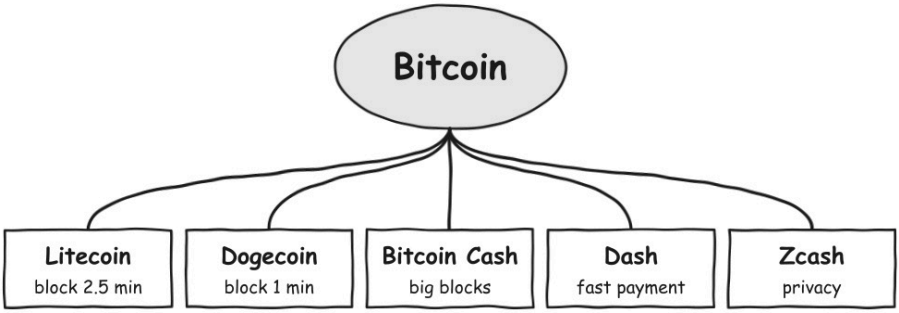
- There will never be more than **21 million** coins.
- A miner's reward can't be spent until 100 blocks have piled on top.
- For large sums people wait for 6 confirmations, about an hour (Chapter 8).

The trap: forgotten change

An ordinary wallet adds change by itself. But if a program builds a transfer by hand and the change output is forgotten, the whole remainder becomes the fee.



The relatives



built alike, but each has its own network and coins

Network	Block, roughly	What sets it apart	Address starts with
Bitcoin	10 min	the eldest in the family	`1`, `3`, `bc1`
Litecoin	2.5 min	a different mining algorithm, Scrypt	`L`, `M`, `ltc1`
Dogecoin	1 min	no upper limit on issuance; can be mined together with Litecoin	`D`
Bitcoin Cash	10 min	much larger blocks	`bitcoincash:q…`
Dash	2.5 min	fast payments (InstantSend) and protection of blocks against replacement (ChainLocks)	`X`
Zcash	75 s	transfers can be hidden	transparent `t1`, shielded `zs`, `u1`

They are built alike, so the questions are almost the same too. Let's ask five networks for their chain height in one go:

```
for net in bitcoin litecoin doge bch dash; do
  echo -n "$net: "
  curl -s https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"getblockcount","params":[]}'
  echo; sleep 1
done
```

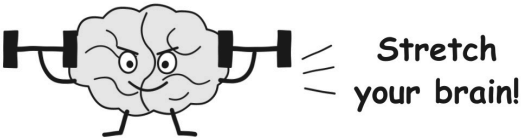
```
bitcoin: {"jsonrpc":"2.0","result":977717,"id":1}
litecoin: {"result":3219941,"error":null,"id":1}
doge: {"result":6474162,"error":null,"id":1}
bch: {"result":979125,"error":null,"id":1}
dash: {"result":2578936,"error":null,"id":1}
```

A second trap: "there's an error field, so there's an error"

Look at the responses: on four networks the `error` field is always there. The value `null` means "all is well". Check not whether the field exists but what is in it, as Chapter 11 advised.

And don't mix up networks. Relatives don't accept each other's coins. Bitcoin Cash's old address format looked just like Bitcoin's, so a new one was introduced there, with the prefix `bitcoincash:`.

Think it over



1. A wallet holds pieces of 0.3, 0.5 and 0.2 BTC. You need to pay 0.4, with a fee of 0.001. Is one piece enough, and what will the change be?
2. Why is sending 1 BTC assembled from a hundred small pieces more expensive than sending it as one whole?
3. Input 1.0 BTC, a single output of 0.4 BTC. Where did the 0.6 go?

Answers: 1) yes, the 0.5 piece; the change is 0.099. 2) More inputs mean a bigger transfer. 3) It is the fee. Most likely the change was forgotten.

Key points

- Bitcoin has no account balance; it has unspent pieces (UTXOs).
- The fee is inputs minus outputs; don't forget the change.
- The relatives are built alike, but they are different networks.

Next: Chapter 15. Ethereum: accounts, gas and events.

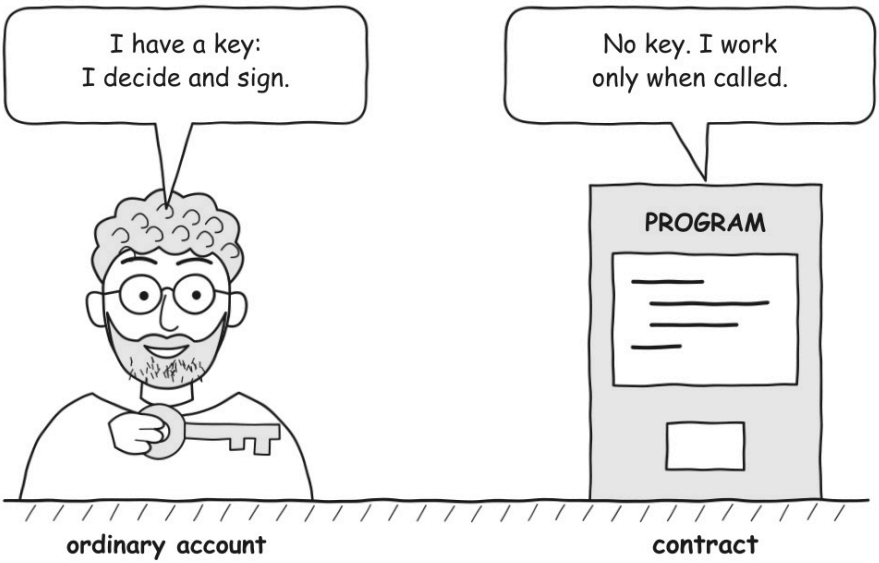
Chapter 15. Ethereum: accounts, gas and events

"Why is sending dollars more expensive?"

Artyom sent his supplier a little ETH, and then some USDT. The sums were similar, but the USDT took more than twice as much "gas". What is this gas, and why is it so?

Digging in: two kinds of accounts

In Ethereum an address has a balance, as at a bank. Nothing like the pieces in Bitcoin (Chapter 14). And accounts come in two kinds.



	Ordinary account	Contract
Who controls it	the key's owner	its program
Can start a transfer by itself	yes	no, only when called
Has code	usually not	yes

Programs are run by the **EVM**, the Ethereum Virtual Machine. It is like a game console inside every node: the same program gives the same result everywhere.

Try it: where the program lives

The `eth_getCode` method shows the code at an address. Let's ask about the USDT contract and keep only the beginning of the response:

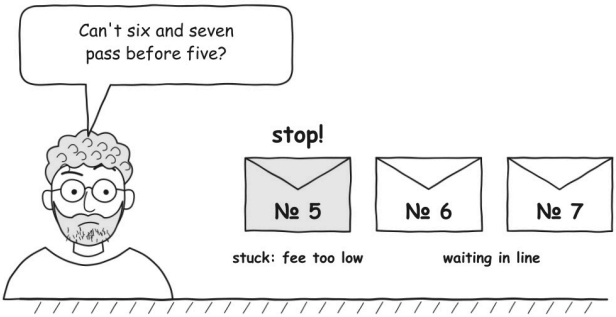
```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getCode","params":
  ["0xdac17f958d2ee523a2206206994597c13d831ec7","latest"]}' | head -c 80; echo
```

You will see the start of a long string of characters: that is the program, more than 11 thousand bytes. Replace the address with `0x0000000000000000000000000000000000000dEaD` and you get `"0x"`, meaning there is no code.

Since 2025 an ordinary account can also hand its actions over to a program (the Pectra upgrade). Then the response starts with `0xef0100`. So "there is code" no longer always means "this is a contract".

A running number: the nonce

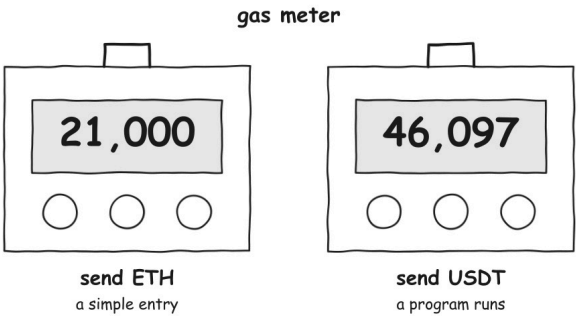
Every transaction of an account gets a number: 0, 1, 2 and so on. It is called the **nonce**. The network executes them strictly in order and won't accept the same number twice.



If a transfer with a small fee gets stuck, the following ones wait behind it. Wallets offer to "speed up": they send a new transfer with the same number and a bigger fee.

Gas: a meter of work

Gas measures how much work a transaction requires. A simple "send ETH" entry costs 21,000 today. A token transfer runs a program, and it needs more gas.



Fee = gas used × (base price + tip)

- The **base price** is set by the network itself. This money is **burned**.
- The **tip** goes to the validator: the higher it is, the more willingly you are picked.
- The gas price is counted in **gwei**, a billionth of an ETH.

A block holds a limited amount of gas. When blocks are more than half full, the base price rises; when less, it falls. For example, block 26,502,250 could hold 60 million gas and was 93% full.

The wallet sets a **gas limit**. Only what is used gets charged. But if the limit wasn't enough, the transfer fails and the gas spent isn't returned.

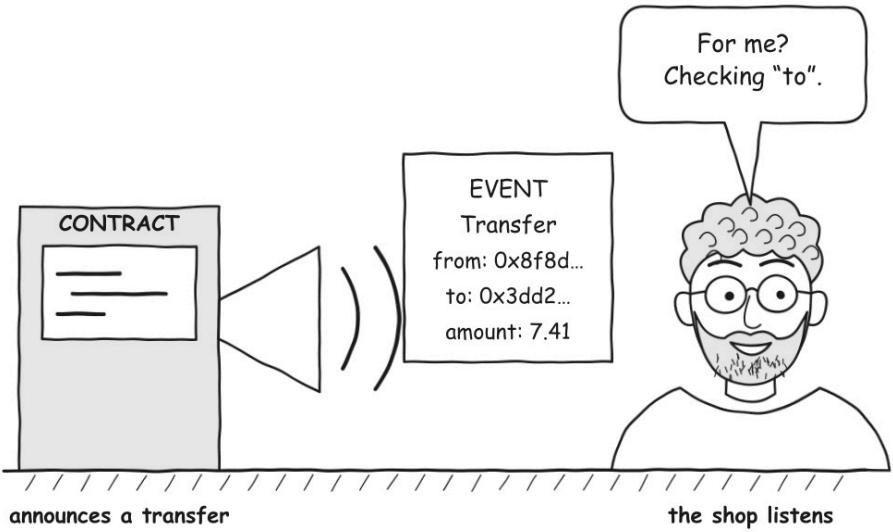
A real USDT transfer

This is a transfer of 7.41 USDT from that very block. By that time the sender had already made 423,464 transactions.

What	Value
gas used	46,097
base price	0.058 gwei
tip	2 gwei
fee	about 0.000095 ETH
of which burned	about 0.000003 ETH

Events: how a contract reports a transfer

Token balances sit inside the contract. So that the world learns of a transfer, the contract announces an **event** (log). For tokens it is called `Transfer`: from whom, to whom, how much.



Events are kept in the transaction receipt from Chapter 7. Let's ask for the receipt of this transfer:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionReceipt","params":
  ["0xc0015ff5178d72a68831b133187c3f9f1fbee2e440d3af311d0b32110e2cf016"]}'
```

Find `logs` in the response:

Field	Start and end	What it means
`address`	`0xdac1…1ec7`	who announced it: the USDT contract
`topics[0]`	`0xddf2…b3ef`	kind of event: Transfer
`topics[1]`	`…8f8d…884e`	from whom
`topics[2]`	`…3dd2…a9c0`	to whom
`data`	`…711150`	7,410,000, that is, 7.41 USDT

And the `eth_getLogs` method can search for events. Let's count the USDT transfers in the newest block:

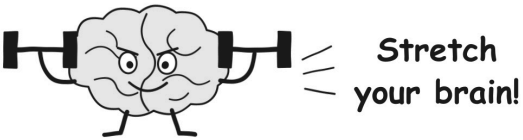
```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getLogs","params":[{
  "address":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"],
  "fromBlock":"latest","toBlock":"latest"}]}' | grep -o '"logIndex"' | wc -l
```

You get the number of transfers in one block. A public node won't answer about past blocks: that is already archive territory (Chapter 10).

The trap: "I only listen for events"

- **A transfer of ETH itself creates no Transfer event.** A shop that waits only for events will miss such a payment. Future network upgrades may change this.
- **ETH can come from a contract**, for example from some wallets. Then your address isn't visible in the transaction itself.
- **Any contract can announce an event.** Check `address`: is this the real token (Chapter 9)?

Think it over



1. An ETH transfer, 21,000 gas, price including tip 2 gwei. What is the fee in gwei and in ETH?
2. Several blocks in a row are 30% full. What happens to the base price?
3. A shop watches only the Transfer events of the USDT contract. A customer paid in ETH. Will the shop see it?

Answers: 1) 42,000 gwei, which is 0.000042 ETH. 2) It will go down. 3) No.

Key points

- Accounts are controlled either by a key or by a program.
- The fee is gas multiplied by price: base plus tip.
- You learn of token transfers from events, while ETH has to be caught separately.

Next: Chapter 16. Ethereum has many relatives: other EVM networks and "second floors".

Chapter 16. Ethereum's relatives and the "second floors"

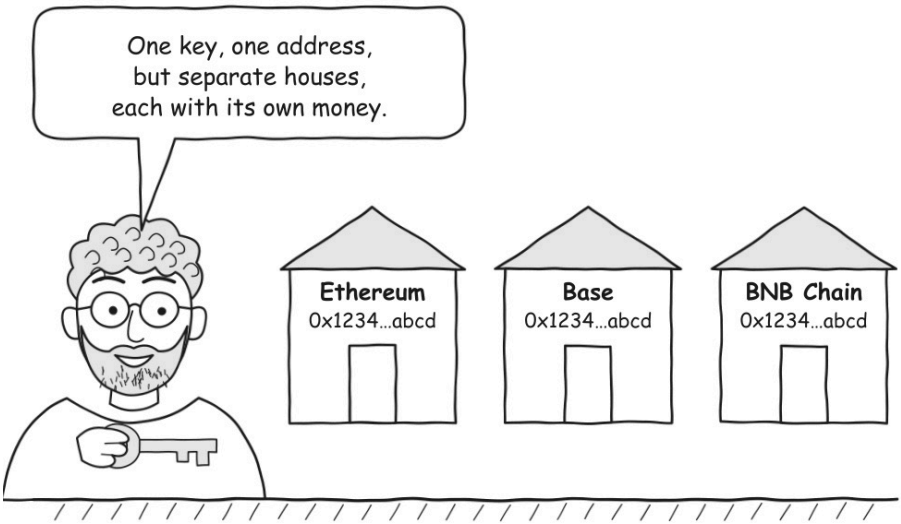
"I sent it on Base"

A customer writes: "I sent USDC on Base, same address." Artyom is puzzled. Is it a different token? A different network? And why is the address the same?

Digging in: one console, different houses

Many networks adopted the EVM from Chapter 15. So they have identical addresses, the same `eth_…` methods, and one contract program can work in each of them. Such networks are called **EVM-compatible**.

But their notebooks are **different**. Their own blocks, their own balances, their own fee.



To keep networks apart, each has a number — the **chain ID**. It is part of the signature, so a transfer signed for one network won't be accepted by another.

Try it: ask six networks for their number

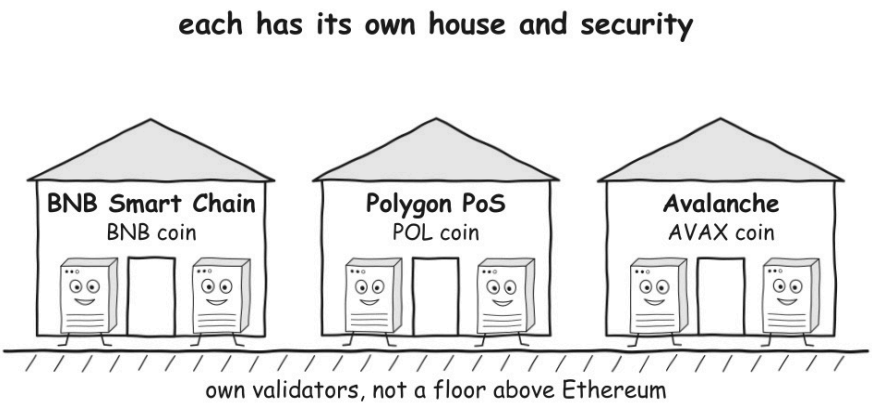
```
for net in ethereum base arb bsc mantle monad; do
  echo -n "$net: "
  curl -s https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_chainId","params":[]}'
  echo; sleep 1
done
```

Numbers in hexadecimal will come back. Here they are as ordinary numbers, along with how often blocks were coming out when checked:

Network	Chain ID	A block about every	Fee paid in
Ethereum	1	12 s	ETH
Base	8453	0.2 s	ETH
Arbitrum One	42161	0.3 s	ETH
BNB Smart Chain	56	0.45 s	BNB
Mantle	5000	2 s	MNT
Monad	143	0.3 s	MON

Relatives of the first kind: independent ones

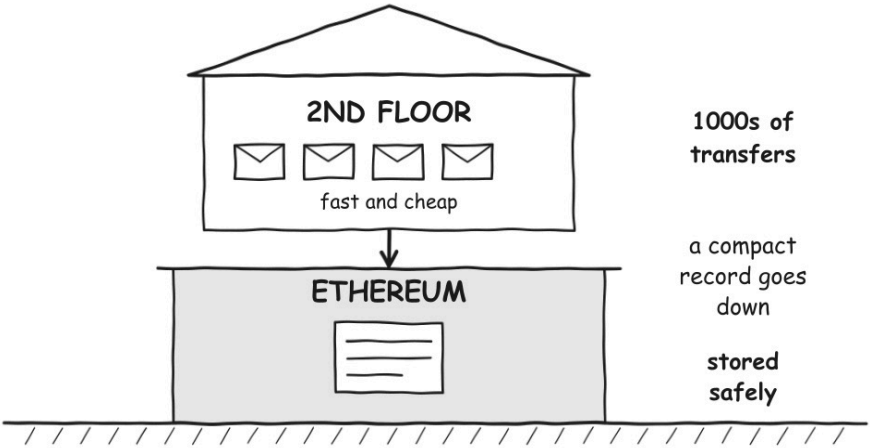
They have their own validators and agree on blocks by themselves.



Polygon PoS, however, periodically leaves marks about its blocks in Ethereum. Monad, launched in November 2025, is also one of the independent ones.

Relatives of the second kind: second floors (L2)

An **L2** carries out transfers on its own floor, quickly and cheaply, and sends compressed records of them to Ethereum. The most common kind is called a **rollup**.



- The order of transfers is usually set by a single operator, the **sequencer**. That is why everything is so fast. And that is why at this point you are trusting it.
- **Optimistic** rollups (Arbitrum One, OP Mainnet) treat the records as correct but leave time to challenge them.
- **ZK** rollups (Mantle, for example) attach a mathematical proof.
- In 2026 Base moved to its own code and combines several kinds of proofs.

Because of the time needed for checking, getting money back to Ethereum through the official bridge isn't quick:

From network	How long a withdrawal to Ethereum takes
Arbitrum One	at least 7 days
OP Mainnet	7 days
Base	5 days, in some cases 1 day

According to the networks' documentation as of November 27, 2026. These periods change: OP Mainnet has a redesign planned for January 2027.

“Seen quickly” doesn’t mean “final”

Remember `latest`, `safe` and `finalized` from Chapter 8? Here is what a Base node answered:

What was asked	Lag behind the newest block
`latest`	—
`safe`	about 17 seconds
`finalized`	about 19 minutes

A block appears in a fraction of a second, but becomes final only when the record of it is already final in Ethereum itself.

The fee on the second floor

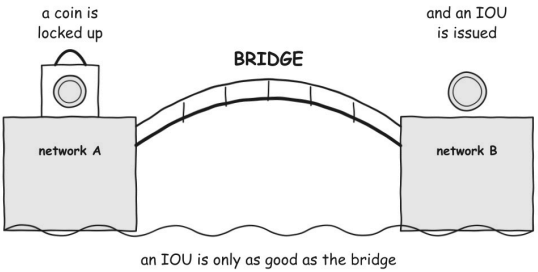
It consists mainly of two parts: the work on the L2 itself and the charge for recording in Ethereum. There can also be small extras. In a receipt from Base the second part sits in the `l1Fee` field:

```
curl https://public-base-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionReceipt","params":
  ["0xa1d4398ce62b570a2810ad2b2ed347f388f9ad85aee4b62ba3215917cf3ac7e5"]}'
```

Here the work on Base cost about 0.0000035 ETH, and the recording in Ethereum about 0.0000000006 ETH. It isn’t always this cheap.

Bridges

To “move” from network to network, people use a **bridge**. Often it locks coins in one network and issues their substitute in another.



That is why **USDC** and **USDC.e** are different tokens with different contract addresses. The first is issued by the company Circle itself; the second appears through a bridge.

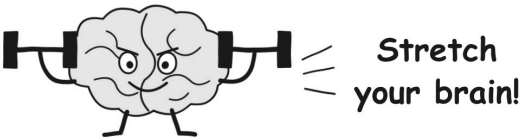
The trap: sent on the wrong network

In Chapter 6 we said the shop won't see such a payment. That is true. But is the money lost?

- **You hold the key to the address** — usually the money can be recovered: switch your wallet to that network. To send it onward you will need that network's coin for the fee.
- **It is an exchange address** — you don't have the key. If the exchange doesn't support that network, the money most likely can't be returned.
- **It is a contract wallet** — it may not exist on the other network. You need an expert's help.

> **The rule:** don't tell anyone your secret phrase, not even someone who promises to "recover" the transfer (Chapter 6).

Think it over



1. The shop expected USDT on Ethereum but received it on BNB Smart Chain at its own address. Artyom has the key. What should he do?
2. A block on Base appeared in 0.2 seconds. Can a large payment be considered final?
3. A customer sent USDC.e instead of USDC. Is it the same thing?

Answers: 1) switch the wallet to BNB Smart Chain; sending it onward requires BNB. 2) No, wait for finalized. 3) No, it is a different token: the shop decides whether to accept it.

Key points

- EVM networks are alike, but they are different notebooks: check the chain ID.
- There are independent networks and second floors on top of Ethereum.
- A fast block isn't yet final.

Next: Chapter 17. TRON: energy instead of gas, and the stablecoins' favorite road.

Chapter 17. TRON: energy instead of gas, and the stablecoins' favorite road

"Can I use TRC-20?"

A customer asks: "Do you accept USDT over TRC-20?" It is the third time this week Artyom has heard it. What is TRC-20, and why does everyone ask for it?

Digging in: a network of its own

TRON is a separate blockchain with its own coin, **TRX**. Its smallest fraction is called a **sun**: there are a million sun in one TRX.

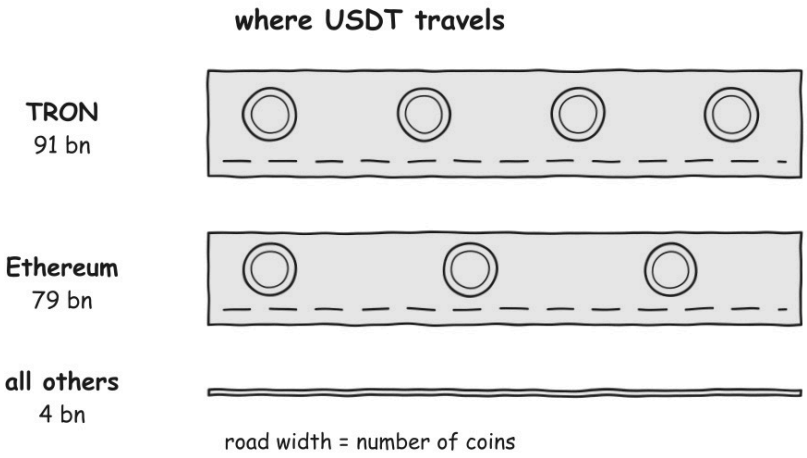
In many ways TRON is built like Ethereum: balances on accounts, smart contracts, events. Programs are run by its own virtual machine, a close relative of the EVM from Chapter 15.

TRC-20 is the set of rules for making tokens in TRON, a sibling of Ethereum's ERC-20 standard. "USDT over TRC-20" means: the USDT token on the TRON network.

Blocks come out every 3 seconds. They are written by 27 elected representatives (Chapter 8).

The stablecoins' favorite road

USDT lives on many networks, but very unevenly. Here are the figures from Tether itself, the company that issues the coin, accessed on November 27, 2026:



More than half of all USDT is on TRON. That is why customers so often choose this network.

Source: app.tether.to/transparency.json. Counted as issued coins minus the reserve not yet put into circulation.

The address starts with T

Inside, a TRON address is 20 bytes, as in Ethereum, with the prefix `41`. But people are shown it differently, starting with the letter `T`:

What people see	What is inside (the USDT contract address)
`TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t`	`41a614f803b6fd780986a42c78ec9c7f77e6ded13c`

The `T` form hides a checksum: a typo can be spotted. Let's ask a node about the real address, and then change its last letter:

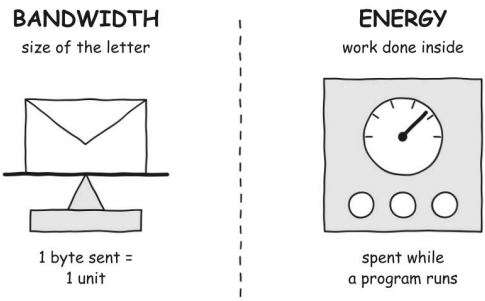
```
curl https://public-tron-mainnet.rpcnode.dev/wallet/validateaddress \
  -d '{"address":"TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t","visible":true}'

curl https://public-tron-mainnet.rpcnode.dev/wallet/validateaddress \
  -d '{"address":"TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6u","visible":true}'
```

```
{"result":true,"message":"Base58check format"}
{"result":false,"message":"Invalid address"}
```

`"visible":true` means "I write addresses in the T form".

Two charges instead of gas

In Chapter 7 we promised to explain how TRON counts a transfer's "weight". It counts it with two measures.



- **Bandwidth** is for space. Consumption roughly equals the transfer's size in bytes. Every transfer pays it.

- **Energy** is for the program's work. It is needed when a contract is called, which includes sending USDT.

An address that has received TRX at least once gets a gift: **600 units of bandwidth a day**. What is spent is restored gradually, over 24 hours. There is no free energy.

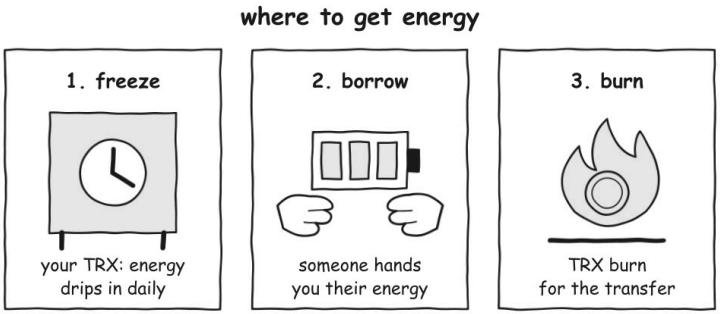
If something is short, the network takes TRX and burns it. At what rate? Let's ask a node ourselves:

```
curl -s https://public-tron-mainnet.rpcnode.dev/wallet/getchainparameters |
  python3 -c "import json,sys
for p in json.load(sys.stdin)['chainParameter']:
    if p['key'] in ('getFreeNetLimit','getTransactionFee','getEnergyFee'):
        print(p['key'], p['value'])"
```

```
getTransactionFee 1000
getEnergyFee 100
getFreeNetLimit 600
```

So a unit of bandwidth costs 1000 sun and a unit of energy 100 sun. These prices are changed by a vote of the representatives.

Where to get energy

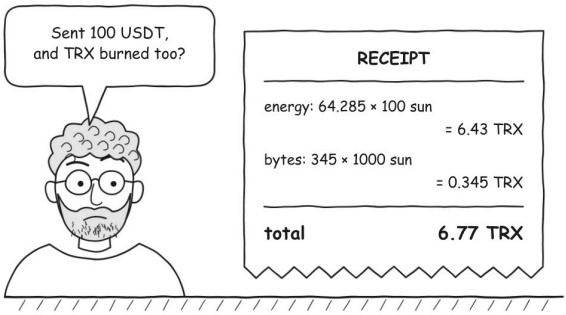


- **Freeze TRX** (staking). The coins remain yours, and the address receives a share of the total energy every day. To take them back, you wait 14 days after unfreezing.
- **Borrow it**: the owner of frozen TRX hands their energy to another address.
- **Do nothing**: burned TRX will pay for what is missing.

How much to freeze? It depends on how many coins everyone else has frozen. On the day we checked, about 6600 frozen TRX would have been enough for one USDT transfer a day.

The receipt of a real transfer

Here is a transfer of 100 USDT from block 88,352,467. The sender had neither energy nor free bandwidth:



The receipt is fetched with the `gettransactioninfobyid` method. It is closed on the public node and needs a personal endpoint, so this is a sketch:

```
curl https://YOUR_ENDPOINT/walletsolidity/gettransactioninfobyid \
  -d '{"value":"0eb7b372f5a268ce56430392de8db2fd970455bb0970c67b47a7b2673abb513c"}'
```

Field in the response	Value	What it means
`energy_usage_total`	64,285	total energy spent
`energy_penalty_total`	49,635	of which the popularity surcharge
`energy_fee`	6,428,500	burned for energy, in sun
`net_fee`	345,000	burned for bytes, in sun
`fee`	6,773,500	total, which is 6.7735 TRX
`result`	`SUCCESS`	the program ran successfully

What is this surcharge? When one contract is used too much, the network makes calls to it more expensive. For USDT that day, energy consumption was almost 4.4 times what it would have been without the surcharge.

And one more thing: if the recipient didn't have any USDT before, twice as much energy is needed, 130,285. The contract has to open a new line for them.

The fresh block and the final one

TRON nodes have two sets of addresses. `/wallet/…` answers from the freshest data. `/walletsolidity/…` answers only from final data. A block becomes final when 19

representatives out of 27 have produced their own blocks at its height or above. Usually this takes about a minute.

On the day we checked, the fresh block was 88,352,763 and the final one 88,352,746.

For experiments there is also the Nile test network, which has its own public node: `public-tron-nile.rpcnode.dev`.

The trap: "it's in a block, so it has arrived"

Let's look at the direct USDT transfers in the latest final block:

```
curl -s https://public-tron-mainnet.rpcnode.dev/walletsolidity/getnowblock > block.json
python3 <<'EOF'
import json
usdt = '41a614f803b6fd780986a42c78ec9c7f77e6ded13c'
rows = []
for t in json.load(open('block.json'))['transactions']:
    v = t['raw_data']['contract'][0]['parameter']['value']
    data = v.get('data', '')
    if v.get('contract_address') == usdt and data.startswith('a9059cbb'):
        amount = int(data[72:136], 16) / 1_000_000
        rows.append((t['txID'][:8], t['ret'][0]['contractRet'], amount))
print('USDT transfers:', len(rows))
print('failed:', sum(r[1] != 'SUCCESS' for r in rows))
for r in rows[:3]:
    print(*r)
EOF
```

Your numbers will differ. In block 88,352,467, two of 41 such transfers failed. One of them was for 100,000 USDT:

```
fbfec9fa OUT_OF_ENERGY 100000.0
```

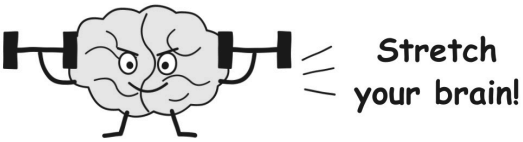
`OUT_OF_ENERGY` means the energy ran out, the program stopped and the USDT went nowhere. Yet the entry is in the block, and on a screenshot the sum looks impressive.

How much energy may be spent is set by the sender: the `fee_limit` field, a ceiling in sun. If the limit isn't enough, the transfer fails, and what has been burned isn't returned.

What a shop should remember

- Wait for a final block and check `result` in the receipt.
- Look in the receipt for a `Transfer` event from the contract `TR7NHq…Lj6t`. That way you will also see transfers made from other programs. USDT has six decimal places.
- To send USDT onward from your address, it needs TRX or energy. An address per order is handy, but collecting the money from a hundred addresses means a hundred paid transfers.
- The very first TRX transfer to a new address costs the sender at least 1 TRX more.

Think it over



1. An address has 600 free bandwidth and plenty of energy. It sends two transfers of 345 in a row. What happens to the second?
2. There is no energy, and the recipient has never held USDT: 130,285 units at 100 sun are needed. How many TRX will burn for energy?
3. A customer sent a hash. The transaction is in a block and says `OUT_OF_ENERGY`. Ship the coffee?

Answers: 1) 255 free units are left, not enough for 345; 0.345 TRX burns for all 345, the two aren't combined. 2) 13.0285 TRX. 3) No, there was no transfer.

Key points

- TRON has two measures: bandwidth for bytes and energy for the program's work.
- No energy means TRX is burned, and noticeably more for USDT than for a simple send.
- A payment is a successful transfer of real USDT in a final block.

Next: Chapter 18. Solana: everything sorted into accounts.

Chapter 18. Solana: everything sorted into accounts

"Where are my USDC?"

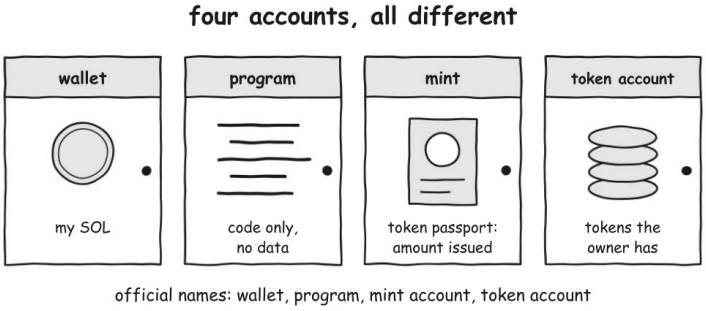
A customer paid in USDC. Artyom asks a node for his wallet's balance and sees only crumbs of SOL. No USDC at all. Was he cheated?

Digging in: a network of accounts

Solana is an independent network with the coin **SOL**. The smallest fraction is called a **lamport**: there are a billion of them in one SOL.

A Solana address is 32 bytes written in letters and digits. It has no special prefix (Chapter 6), and there is no checksum inside either.

Everything the network stores sits in **accounts**. An account here is not just money but any cell with an address. In Ethereum a contract keeps its data inside itself. In Solana **the program sits separately and the data separately**.



Every account has an owner program (the `owner` field). Only it can change the account's data.

Try it: a token's passport

Every token has a passport account, called the **mint**. Let's ask for the USDC passport:

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getAccountInfo","params":
  ["EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",{"encoding":"jsonParsed"}]}'
```

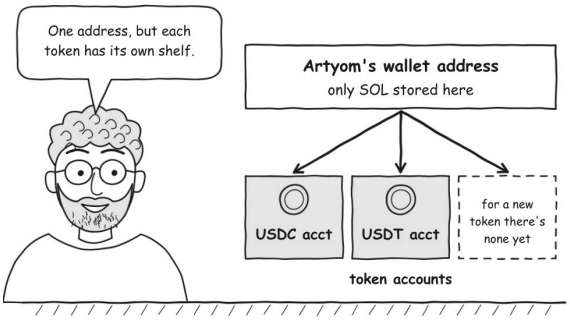
Here are the main points of the response:

Field	Value	What it means
`owner`	`Tokenkeg…Q5DA`	the account is managed by the token program
`supply`	`8513455604386747`	how much has been issued, in small fractions
`decimals`	6	so about 8.5 billion USDC
`mintAuthority`	`BJE5…5ruG`	who can issue new ones
`freezeAuthority`	`7dGb…Crar`	who can freeze (Chapter 9)

USDC has no program of its own. One common token program serves thousands of tokens, and they differ by their passports. There are two such common programs: the ordinary **Token** (USDC and USDT are on it) and the newer **Token-2022** with extra features.

The token account: a separate shelf

How much USDC Artyom has is recorded neither in the wallet nor in the passport, but in a **token account**. Each token has its own.



The main token account is called the **ATA** (associated token account). Its address is computed from three things: the wallet address, the token's passport and the token program. That is why a customer is given the **wallet address**, and their program finds the shelf itself.

Let's look at a real token account. First we ask how many tokens it holds, then how much SOL:

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getTokenAccountBalance","params":
  ["BnGihgNUu4X1JdBMNhbWnXUcpib8UQQC154isHTYwDNK"]}'

curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getBalance","params":
  ["BnGihgNUu4X1JdBMNhbWnXUcpib8UQQC154isHTYwDNK"]}'
```

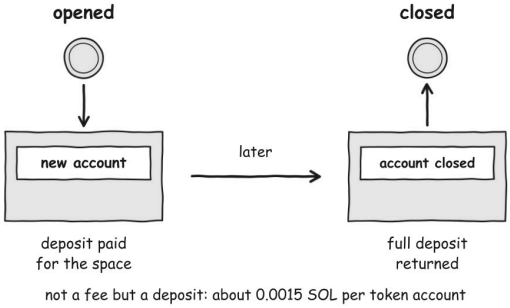
In the first response, `"uiAmountString":"207637.336362"` is USDT. In the second, `"value":2039280` is lamports, just 0.00203928 SOL. The `getBalance` method **doesn't see**

tokens. That is where Artyom's USDC "disappeared" to.

By the way, the owner of this shelf is the address of the USDT passport itself. It looks like people mistakenly sent USDT "to the USDT address", as in Chapter 9.

A deposit for the space

Every account takes up memory on all nodes. So it carries a **deposit** in SOL (called rent, although it isn't really rent).



A token account is 165 bytes. How much deposit does that require?

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,
  "method":"getMinimumBalanceForRentExemption","params":[165]}'
```

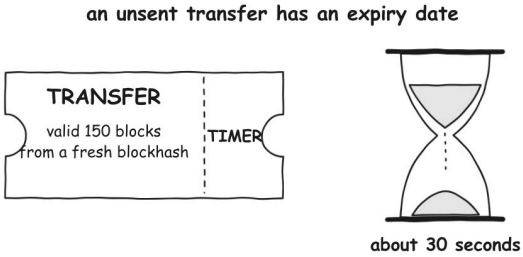
```
{"jsonrpc":"2.0","result":1488440,"id":1}
```

That is about 0.0015 SOL. In 2026 the deposit is being lowered in stages, so a program doesn't hard-code the number but asks for it. If the recipient has no shelf for the token yet, the sender usually creates it in the same transfer and pays the deposit.

The fee and the expiry date

The base fee is **5000 lamports per signature**: for an ordinary transfer with one signature that is 0.000005 SOL. Half is burned. On top you can add a priority fee, all of which goes to the validator. The fee is charged even if the transfer fails.

Every transfer includes a **blockhash** — the fingerprint of a recent block. It also sets the expiry date.




```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getLatestBlockhash",
  "params":[{"commitment":"finalized"}]}'

curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getBlockHeight",
  "params":[{"commitment":"finalized"}]}'
```

The first response gave `"lastValidBlockHeight":452330099`; in the second, a moment later, the height was 452,329,960. That leaves 139 blocks. If the deadline passes and the transfer isn't in any block, it will never go through. Then you can sign a new one. The exception is special deferred transfers (durable nonce), which have no such deadline.

2026: even faster

What changed	Before	After
Slot length, that is, the turn for one block (October)	0.4 s	0.2 s
Finality, the new Alpenglow voting procedure (October)	about 13 s	about 0.15 s
Maximum transaction size, the new v1 format (September)	1232 bytes	4096 bytes

Let's check the speed ourselves. This method returns how many slots passed in a minute:

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getRecentPerformanceSamples","params":[1]}'
```

We got `numSlots` = 277 over 60 seconds: about 0.22 s per slot. Slots can be empty, so the slot number and the block height are different numbers; don't confuse them.

A node can answer with different strictness: `processed` — the block has been seen but may still be skipped; `finalized` — final. There is also the middle word `confirmed`: since Alpenglow it means the same as `finalized`. Orders are shipped on `finalized`.

The trap: "the transaction can't be read"

Because of the new v1 format, you need to add `maxSupportedTransactionVersion` with a value of 1 to the `getTransaction` and `getBlock` methods. Without it, new transactions return an error. The `getTransaction` method is closed on the public node, so this is a sketch for a personal endpoint:

```
curl https://YOUR_ENDPOINT \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getTransaction","params":
  ["TRANSFER_SIGNATURE",{"encoding":"jsonParsed","commitment":"finalized",
  "maxSupportedTransactionVersion":1}]}'
```

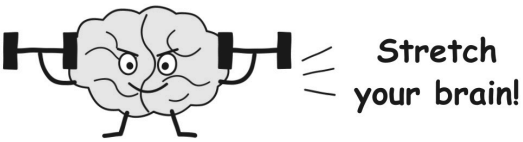
In Solana the "transfer signature" serves as its number, in place of a transaction hash.

What a shop checks

- The transaction succeeded: `meta.err` is `null`.
- The token account of **your wallet** with the right token passport grew by the right amount: compare `preTokenBalances` and `postTokenBalances`.
- It is the payment for this very order. For that, a **reference** is added to the transfer — a one-time marker address for the order.
- This signature hasn't been counted before.

And be more careful with plain SOL: a transfer to a wrong address will still go through. There will most likely be nobody to return such money.

Think it over



1. A wallet signed a transfer, but the network saw it two minutes later. What happens?
2. A shop asks `getBalance` for its wallet to catch a USDC payment. What is the mistake?
3. Artyom sends USDC to a supplier who doesn't have such a shelf yet. What will he have to pay besides the fee?

Answers: 1) the deadline has passed, the transfer won't go through. 2) `getBalance` *shows only SOL; USDC sits on a token account. 3) The deposit for the new token account, about 0.0015 SOL.*

Key points

- In Solana everything is accounts; programs and their data sit separately.
- Tokens sit on token accounts: one per token, with a deposit.
- Read transactions with `maxSupportedTransactionVersion` and wait for `finalized`.

Next: Chapter 19. TON: letters instead of phone calls.

Chapter 19. TON: letters instead of phone calls

“I’ve sent it, now what?”

A customer sent USDT on the TON network. Instead of a single entry, Artyom sees a whole chain in the explorer: five transactions, some unfamiliar addresses and a bit of small change. Is that one payment or five?

Digging in: everyone is a contract, everyone writes letters

TON (The Open Network) is an independent network. Its coin is called **Gram** (GRAM). It used to be called Toncoin, and you will still often come across that name. One Gram is a billion **nanograms**.

Since April 2026 blocks have been coming out about every 0.4 seconds, and a record becomes final in about a second.

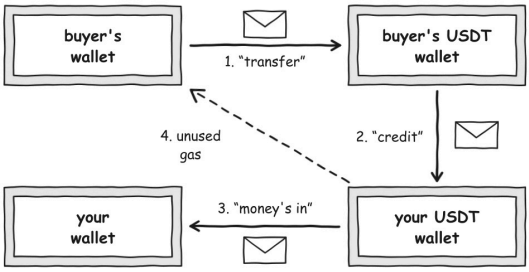
Here is the main difference. Every account here is a smart contract, **the wallet included**. Contracts don’t call each other directly; they send **messages**.

In Ethereum everything one transfer sets in motion happens inside one transaction: either completely or not at all. It is like a phone call. In TON it is correspondence: you send a letter and don’t wait on the line. The recipient deals with each letter later, in a separate transaction. This arrangement is called **asynchronous**.

All the correspondence that grows out of one instruction is called a **trace**. The success of one transaction in it doesn’t yet mean the whole job worked out.

How USDT travels

Tokens in TON are called **jettons**. Each has a main contract (the **master**), and each owner has their own separate **jetton wallet**. It resembles the token accounts from Chapter 18, only it is an independent contract. For USDT the master contract is `EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDs`.



each letter is handled by its own transaction

This isn't made up but a real transfer of 11.99 USDT, made on the night of November 27, 2026:

Whose transaction	What it received	What it sent on
sender's wallet	a signed instruction	"transfer" and 0.05 Gram for expenses
sender's USDT wallet	"transfer"	"credit"
recipient's USDT wallet	"credit"	a notification and change
recipient's wallet	the notification and 1 nanogram	nothing
sender's wallet	change, about 0.0494 Gram	nothing

It all fit into one second. Money for expenses is attached with a margin, and the remainder comes back.

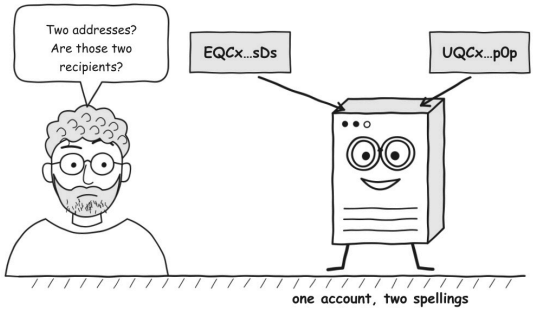
A curious detail: the fourth transaction is marked as `aborted`. The recipient didn't have enough gas to run the program: only a single nanogram arrived. Yet the USDT had already been credited by the third transaction. You can't judge by one entry, one way or the other.

You can view the whole chain with a request to an indexer. It is closed on the public node, so this is a sketch for a personal endpoint:

```
curl -G https://YOUR_ENDPOINT/api/v3/traces \
  --data-urlencode "trace_id=7rg6DNoAZeR1KPVJHd1FWIc/dobTITAsXqK1Y2kuQlM="
```

One account, two addresses

An account's real name is the "raw" one: the number of the network's part and 64 characters. People are shown a different 48-character form with a checksum. And one account has several such forms.



Let's ask a node about the USDT address:

```
curl -s "https://public-ton-mainnet.rpcnode.dev/api/v2/detectAddress?address=EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDs" |
  python3 -c "import json,sys
r = json.load(sys.stdin)['result']
print(r['raw_form'])
print(r['bounceable']['b64url'])
print(r['non_bounceable']['b64url'])"
```

```
0:b113a994b5024a16719f69139328eb759596c38a25f59028b146fecdc3621dfe
EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDs
UQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_p0p
```

- `EQ…` is the bounceable address; this is how ordinary contracts are written.
- `UQ…` is non-bounceable; this is the recommended way to write wallets.
- On the test network the forms start with `kQ` and `0Q`.

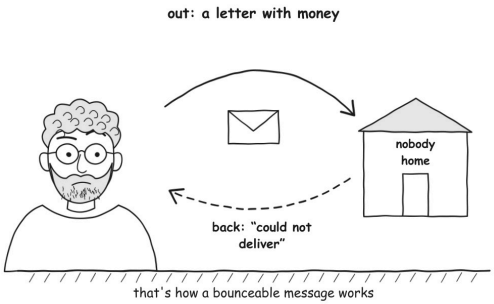
So in the shop's database you **compare the raw form**, not the strings with `EQ` and `UQ`.

The checksum catches typos. If you write `sDt` instead of `sDs` at the end, the node responds with code 422:

```
curl -s -o /dev/null -w '%{http_code}\n' \
  "https://public-ton-mainnet.rpcnode.dev/api/v2/detectAddress?address=EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDt"
```

A letter to an empty house

What if the recipient couldn't make sense of the letter, or their contract doesn't exist yet? That depends on a mark on the message.



A bounceable message returns to the sender with the rest of the money, if there is enough of it for the trip back. A non-bounceable one stays with the recipient. A new wallet may not yet be "deployed", meaning its program isn't written into the network. That is why wallets are paid at the non-bounceable address.

Try it: check the speed

Let's find the number of the latest master block, and the times when it and the one a thousand before it came out:

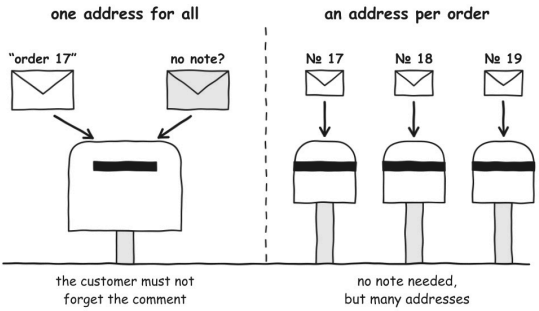
```
B=https://public-ton-mainnet.rpcnode.dev
last=$(curl -s $B/api/v2/getMasterchainInfo |
  python3 -c "import json,sys; print(json.load(sys.stdin)['result']['last']['seqno'])")
for s in $last $((last-1000)); do
  sleep 2
  curl -s "$B/api/v2/getBlockHeader?workchain=-1&shard=-9223372036854775808&seqno=$s" |
    python3 -c "import json,sys
r = json.load(sys.stdin)['result']
print(r['id']['seqno'], r['gen_utime'])"
done
```

```
108011078 1795732480
108010078 1795732076
```

The second numbers are time in seconds. The difference is 404 seconds over 1000 blocks, or about 0.4 seconds per block.

A comment instead of a separate address

A text **comment** can be attached to a transfer in TON. Hence two ways to tell whose payment it is:



A shared address is less trouble, but the customer may forget the comment or copy it wrong. Besides, not all wallets can add one. Such payments are sorted out by hand.

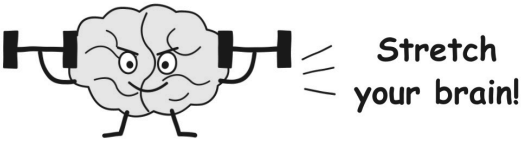
The trap: "a notification arrived, so I ship"

Anyone can write a contract that sends your wallet the very same kind of notification. Or issue their own jetton named "USDT". So you check:

- the notification came from **your** USDT wallet. Its address is asked in advance from the real master contract (`get_wallet_address`);
- the amount matches (USDT has six decimal places);
- the comment or address belongs to this order, and this payment hasn't been counted yet;
- the record made it into the main chain (masterchain), meaning it is final.

A notification arrives only if the sender attached at least a little Gram for it. That is the usual practice, but a homemade program might forget. Then the tokens arrive without a notification.

Think it over



1. The database has an address starting with `UQ`, while the explorer shows `EQ` with the same middle. How do you check whether it is the same account?
2. The first transaction in the chain succeeded. So the USDT arrived?
3. A customer paid to the shared address but forgot the comment. What should the shop do?

Answers: 1) convert both to the raw form (`detectAddress`) and compare. 2) Not yet: the recipient's USDT wallet has to credit it. 3) Don't count it automatically; sort it out by hand: amount, sender, transfer number.

Key points

- In TON contracts correspond: one instruction gives rise to a chain of transactions.
- One account has several address forms; compare the raw one.
- A jetton is verified by its master contract and your jetton wallet.

Next: Chapter 20. Other approaches: Sui and Aptos, XRP Ledger and Stellar, Cardano, Polkadot, Cosmos.

Chapter 20. Other approaches: Sui and Aptos, XRP Ledger and Stellar, Cardano, Polkadot, Cosmos

"Will you take XLM?"

One customer asks whether they can pay through Stellar. Another wants to send USDC on Sui. Artyom has already figured out Bitcoin and Ethereum. Is that knowledge enough? Not quite: these places have their own ways.

Digging in: five questions for a new network

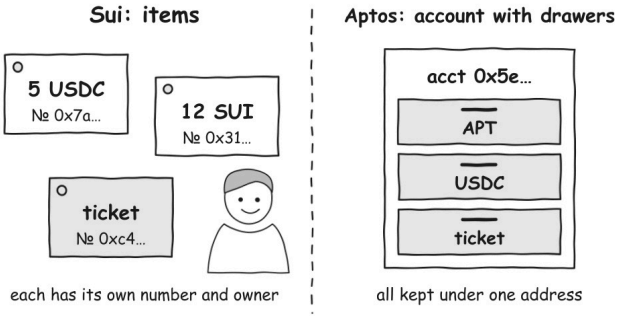
You don't have to learn an unfamiliar network in full. Five answers are enough to start:

1. Where does the money sit: on an account or in separate pieces?
2. What has to be done in advance to receive a transfer at all?
3. How do you tell which order it came for?
4. When is it final?
5. What do you pay the fee with when the money needs to be sent onward?

Sui and Aptos: a common language, different designs

Programs in both networks are written in the **Move** language. It makes sure a valuable thing can't simply be copied. But money is stored differently.

- **Sui** stores everything as **objects**: they are items, each with its own number. Money can sit as separate coin objects, almost like the pieces in Bitcoin. Since 2026 an address can also have a combined balance. The full balance is the sum of the two.
- **Aptos** is closer to a familiar account. Property is stored under the address; it is called **resources**. There is no need to open an account in advance.



Both networks usually make a transfer final in under a second. The fee is paid in their own coin: SUI or APT. Sui has an exception: since 2026 some stablecoins, USDC among them, can simply be sent without a fee (from 0.01).

An address in both is `0x` and usually 64 characters. It starts as in Ethereum, but the length differs: there it is 40 characters after `0x`.

Try it: find USDC in two networks

RpcNode has no free public nodes for the networks in this chapter. So we will ask at the open addresses of those who develop these networks themselves. They are shared and limited: fine for getting acquainted, not for a shop.

Sui, the method is called GraphQL (you list which fields you need):

```
curl -s https://graphql.mainnet.sui.io/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query":"{ coinMetadata(coinType: \"0xdba34672e30cb065b1f93e3ab55318768fd6fef66c15942c9f7cb846e2f900e7::usdc::USDC\") { symbol decimals } }"}'
```

```
{"data":{"coinMetadata":{"symbol":"USDC","decimals":6}}}
```

Aptos, ordinary REST from Chapter 11:

```
curl -s https://api.mainnet.aptoslabs.com/v1/accounts/0xbae207659db88bea0cbead6da0ed00aac12edcdda169e591cd41c94180b46f3b/resource/0x1::fungible_asset::Metadata
```

```
{"type":"0x1::fungible_asset::Metadata","data":{"decimals":6,"icon_uri":"https://circle.com/usdc-icon","name":"USDC","project_uri":"https://circle.com/usdc","symbol":"USDC"}}
```

In both places the token is found not by the word “USDC” but by a long address from the list of the issuer, the company Circle. That is the rule from Chapter 9. In both cases there are six decimal places.

Careful with old examples. Sui used to be queried over JSON-RPC. In July 2026 it was switched off on the Sui Foundation’s nodes; now they offer gRPC and GraphQL. Before copying an example from an article, look at its date.

XRP Ledger and Stellar: transfer networks

Both networks have long been able not only to send their own coin but also to keep track of other parties’ tokens. A page of the chain is called a **ledger** here. XRP Ledger approves a new one every 3–5 seconds, Stellar about every 5 seconds. What is approved isn’t rewritten.

But there are two unfamiliar rules.

First: a deposit for the space. An account doesn't exist until a minimum sum sits on it — the **reserve**. And to receive a token, such as USDC, the account owner opens a **trust line** in advance: "yes, I agree to hold this". Each such entry raises the reserve.

	XRP Ledger	Stellar
Address starts with	`r`	`G`
Minimum on the account	1 XRP	1 XLM
Extra per trust line	0.2 XRP (the first two may do without it)	0.5 XLM
Fee, at the least	0.00001 XRP	0.00001 XLM per operation
Order number	destination tag	memo or an address starting with `M`

The network can change these numbers by vote, so you ask a node for them:

```
curl -s https://s1.ripple.com:51234/ -H 'Content-Type: application/json' \
  -d '{"method":"server_info","params":[{}]}' | grep -o '"validated_ledger":{[^}]*}'
```

```
"validated_ledger":{"age":10,"hash":"AD65FEE0E665A2CA5D08D3757A9E5A65047ECC2391B5E118F3912F25C76AB64F","seq":10859
3026,"base_fee_xrp":1E-5,"reserve_base_xrp":1E0,"reserve_inc_xrp":2E-1}
```

`1E-5` is 0.00001, `1E0` is 1, `2E-1` is 0.2. Now Stellar:

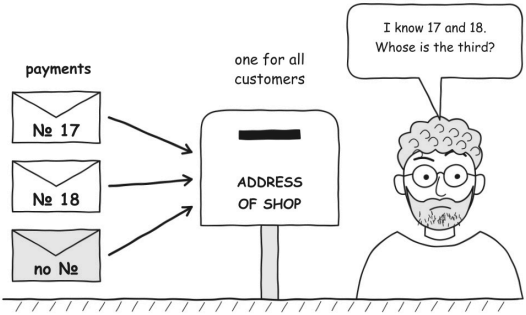
```
curl -s 'https://horizon.stellar.org/ledgers?order=desc&limit=1' \
  | grep -E '"(base_fee|base_reserve)_in_stroops"'
```

```
        "base_fee_in_stroops": 100,
        "base_reserve_in_stroops": 5000000,
```

One XLM is ten million **stroops**. So the fee is 0.00001 XLM, and one share of the reserve is 0.5 XLM (an account needs at least two).

One address for everyone

The second rule. Setting up a separate address for every order is costly here: each one would carry a reserve. So the shop shows a single address and asks for a number to be added: the **destination tag** in XRP Ledger, the **memo** in Stellar.

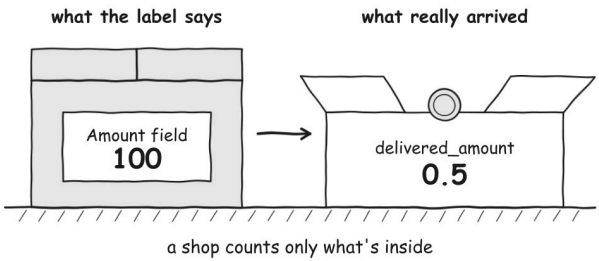


If the customer forgets the number, the money arrives but it is unclear what for. What helps:

- in XRP Ledger you can turn on the `RequireDest` setting: the network won't let a transfer without a number through;
- the number can be hidden in the address itself: an X-address in XRP Ledger, an address starting with `M` in Stellar, if the customer's wallet understands them.

The trap: the label says 100, inside is 0.5

In XRP Ledger the sender can allow a **partial payment**: the transfer counts as successful even if less arrives. Then the `Amount` field says not "received" but "at most". How much actually arrived is recorded in `delivered_amount`.



Here is a real record from the network; let's ask about it by hash:

```
curl -s https://s2.ripple.com:51234/ -H 'Content-Type: application/json' \
  -d '{"method":"tx","params":[{"transaction":"18F797F64D0255522223B1A404BA9CFF7EAE28E2ABB74CC91DD986E162D42DE
C"}]}' \
  | grep -o '"Amount":{[^}]*}\|"delivered_amount":{[^}]*}\|"TransactionResult":"[^"]*"'
```

```
"Amount":{"currency":"XXX","issuer":"rPXeYRpA9Ne3H7fwcM26ZNe98M3rkzTeg6","value":"9.552474017069"}
"TransactionResult":"tesSUCCESS"
"delivered_amount":{"currency":"XXX","issuer":"rPXeYRpA9Ne3H7fwcM26ZNe98M3rkzTeg6","value":"4.5344917829"}
```

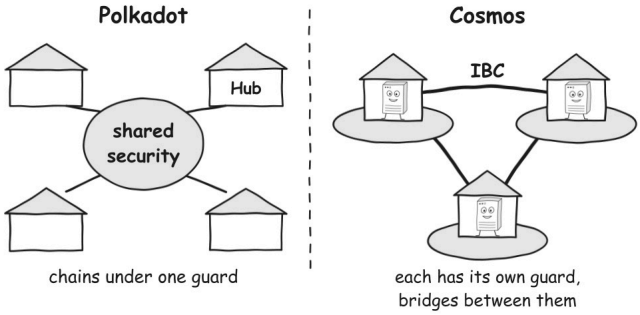
The result is `tesSUCCESS`, `Amount` is about 9.55, but about 4.53 arrived. Such a record isn't fraud in itself. But a scammer can write a large sum, send a crumb and wait for the goods. **Count only** `delivered_amount` and check which token it is and from which issuer.

Cardano: pieces with contents

Cardano is built closer to Bitcoin (Chapter 14): not a balance but unspent pieces. Only a piece can do more, and the model is called **extended UTXO** (eUTXO).

- Alongside the ADA coin, a piece holds other tokens. No separate contract is needed for them.
- A token doesn't travel alone: the same piece must contain a little ADA. So a token arrives with a few coins.
- The fee depends on the transfer's size in bytes.
- Confidence grows gradually: the more blocks on top, the safer.

Polkadot and Cosmos: not one network but many



Polkadot is a central chain that guards the networks connected to it. An important change: since November 2025, users' balances no longer live in the central chain but in the **Polkadot Hub** network (also called Asset Hub).

- USDT (asset number 1984) and USDC (number 1337) are there too.
- An account has a survival threshold, the **existential deposit**: 0.01 DOT. If there is neither enough DOT nor enough USDT or USDC, the account is erased.
- An address starts with `1`, like an old one in Bitcoin. Don't confuse them.

Cosmos is a set of independent networks built from common parts. Each has its own security, coin and address prefix (for Cosmos Hub it is `cosmos1…`). Blocks become final quickly. A transfer has a **memo** field for the order number.

The networks are linked by the **IBC** protocol. A token that has arrived from a neighboring network is named not by a familiar word but by `ibc/` and a long seal, for example `ibc/7F1D3FCF…`. The seal tells which route it took.

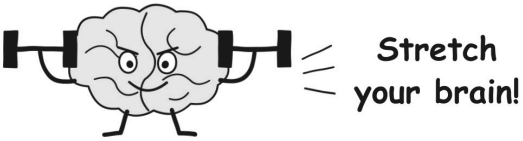
And a stablecoin can leave, too. Circle announced it is ending USDC issuance on the Noble network of the Cosmos family: new issuance has been off since October 13, 2026, and on January 12, 2027 USDC there will be stopped. Holders are advised to move it in advance.

All together

Network	Address	Needed in advance	Final
Sui	`0x…`	nothing	under a second
Aptos	`0x…`	nothing	under a second
XRP Ledger	`r…`	reserve, trust line	3–5 seconds
Stellar	`G…`	reserve, trust line	about 5 seconds
Cardano	`addr1…`	nothing	gradually
Polkadot Hub	`1…`	balance above the threshold	by a mark
Cosmos Hub	`cosmos1…`	nothing	quickly

A trust line is needed only for tokens. Checked on November 27, 2026; rules change.

Think it over



1. Artyom created a new empty address in Stellar and immediately issued an invoice in USDC. What is wrong, and how much XLM is needed at the very least?
2. A transfer arrived: `Amount` is 100 USDC, `delivered_amount` is 0.5, the result is `tesSUCCESS`. Ship coffee worth 100?
3. Customers in XRP Ledger keep forgetting the destination tag. What should be turned on?

Answers: 1) the account doesn't exist yet and no trust line is open; at least 1.5 XLM of reserve is needed, plus a little for the fee. 2) No: 0.5 arrived. 3) The RequireDest setting.

Key points

- Ask an unfamiliar network five questions before showing a customer an address.
- XRP Ledger and Stellar need a reserve, a trust line and an order number; in XRP Ledger count only `delivered_amount`.
- Habits from Ethereum and old examples don't carry over to another network.

Next: Chapter 21. Which network to choose.

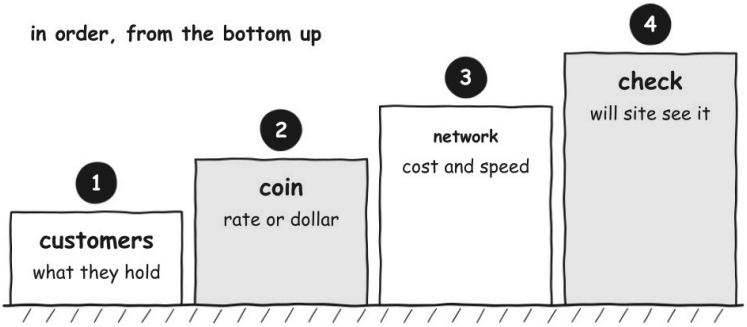
Chapter 21. Which network to choose

“So which of them all is for me?”

Seven chapters have piled up a lot of names: Bitcoin, Ethereum, TRON, Solana, TON and a dozen more. Artyom has one online shop and one coffee roastery. Connect them all? Pick one? By what criterion?

Digging in: four steps

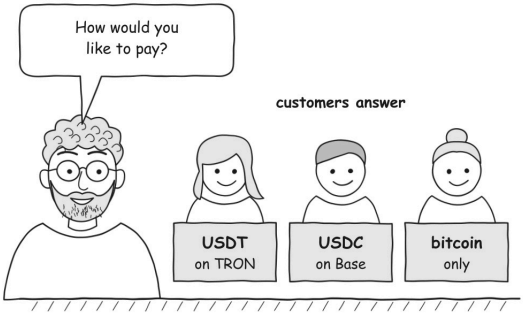
There is no “best network”. There is a network that suits your customers and your order. It is convenient to go in order.



This is about what is convenient to accept as payment. It isn't advice on what to invest in.

Step 1. Ask your customers

The one who pays is the one who already has coins. If they have them on one network and you accept on another, most likely there will be no payment.



Ask in your newsletter, in a chat, at checkout. Two questions are enough: "What do you have?" and "On which network?"

Step 2. Which coin

	The network's own coin	Stablecoin (USDT, USDC)
Price in dollars	changes, sometimes sharply	tries to stay near a dollar
Who is in charge	the network's rules	there is an issuer, which can freeze (Chapter 9)
What the fee is paid in	the coin itself	usually the network's coin, not the stablecoin

If Artyom's coffee prices are in dollars or tied to them, a stablecoin brings fewer surprises. But that is trust too: in the company that issues it.

Step 3. Which network

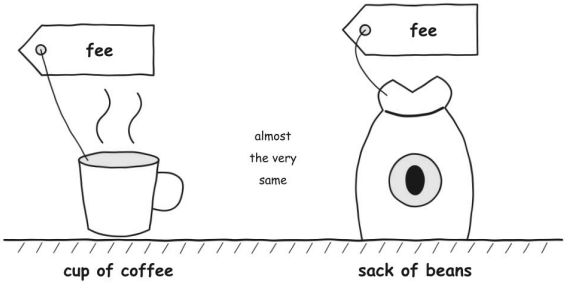
The same stablecoin doesn't live everywhere. Here are the networks from Chapters 14–20. "Yes" means: the issuer's website lists an address of exactly this coin for this network.

Network	Fee	USDT by Tether	USDC by Circle	Don't forget
Bitcoin	BTC	no	no	confirmations
Litecoin, Doge etc.	own	no	no	separate networks
Ethereum	ETH	yes	yes	gas
Base, Arbitrum	ETH	no	yes	finalized
BNB Smart Chain	BNB	no	no	the issuer
TRON	TRX	yes	no	energy
Solana	SOL	yes	yes	token accounts
TON	Gram	yes	no	messages
Aptos	APT	yes	yes	long address
Sui	SUI	no	yes	gRPC, GraphQL
Polkadot Hub	DOT	yes	yes	account threshold
XRP Ledger	XRP	no	yes	reserve, tag
Stellar	XLM	no	yes	reserve, tag
Cardano	ADA	no	no	ADA with a token

According to tether.to/en/supported-protocols and developers.circle.com as of November 27, 2026. "No" doesn't mean there is no token with that name there: check who issues it (Chapter 9). For Cosmos it all depends on the particular network.

Transfer price and order size

The fee depends not on the amount but on the transfer's "weight" and how busy the network is (Chapter 7).



For a sack of beans that is a trifle. For a cup of coffee the fee may cost more than the cup itself. So for small orders people look for where a transfer is cheaper. Prices change, so you find them out in the wallet before sending, not in an old table.

And remember the way back: to send on what you have received, the address needs a coin for the fee. A hundred addresses means a hundred paid transfers.

How long to wait

Final after	Networks
seconds or faster	BNB Smart Chain, Solana, TON, Aptos, Sui, XRP Ledger, Stellar
about a minute	TRON
tens of minutes	Ethereum, Base, Arbitrum One
about an hour	Bitcoin (6 confirmations)

Roughly, based on Chapters 8 and 17–20.

Try it: how old is the final block

Let's ask EVM networks for the `finalized` block (Chapter 8) and work out how many seconds ago it came out:

```
for net in ethereum base arb bsc; do
  t=$(curl -s https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockByNumber","params":["finalized",false]}' \
    | grep -o '"timestamp":"0x[0-9a-f]*"' | cut -d'"' -f4)
  echo "$net: $(( $(date +%s) - t )) s"
  sleep 1.5
done
```

```
ethereum: 929 s
base: 1343 s
arb: 990 s
bsc: 1 s
```

For Ethereum that is about 15 minutes, for Base about 22. For BNB Smart Chain, a second. Your numbers will differ: the mark moves in jumps. You don't have to wait for the most reliable mark for a small order, but then the risk is yours.

Step 4. Will your website see it

Before printing an address on the website, check:

- whether a provider has an endpoint for this network (Chapter 13);
- whether there are payment notifications for this very network. That is not the same thing. At RpcNode you can get an endpoint for more networks than are processed for events (Chapters 26 and 27);
- whether there is a test network to try without real money;
- whether your wallet, and the place where you exchange coins, can handle this coin **on this very network**.

And look at your plan: on Free, RpcNode offers endpoints only for Ethereum and Solana (Chapter 22).

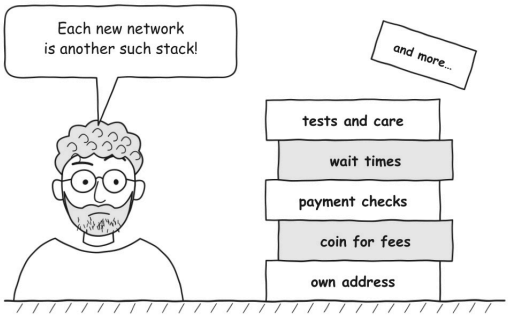
An example: how Artyom decides

Suppose he asked 20 customers. The figures are for practice; your shop will have its own.

What they want to pay with	How many people	Decision
USDT on TRON	9	connect right away
USDC on Base	5	connect right away
BTC	3	later, if people keep asking
ETH	2	later
XLM	1	not for now

Two networks cover 14 people out of 20. Artyom honestly tells the rest that he doesn't accept everything yet.

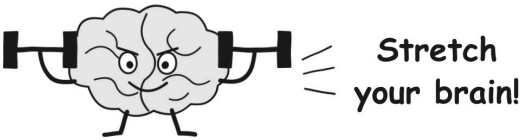
The trap: "I'll connect everything at once"



An extra line on the payment page looks free. But behind it are its own address, its own coin for fees, its own verification rules and its own surprises after upgrades. Start with one or two and add more on request.

And a second trap: writing just "USDT" on the payment page. The customer will choose the network themselves and may choose the wrong one (Chapter 16). Write coin and network together, in large type: "USDT on the TRON network".

Think it over



1. An order is for 5 dollars, and the fee on the chosen network today is 2 dollars. Who finds this inconvenient, and what can be done?
2. A customer asks you to accept USDT on Base. What does the list on Tether's website show?
3. Artyom wants to accept eight networks from day one. What would you say against it?

Answers: 1) the customer: the fee adds 40% on top of the order; offer a network where a transfer is cheaper right now. 2) Base isn't there: before accepting a token with that name, find out who issues it. 3) Eight stacks of chores; start with the one or two that customers need most.

Key points

- Choose in order: customers, coin, network, check.
- Compare the fee with the order size and take into account how long you will wait.
- Start with one or two networks and always name the coin together with the network.

Part four is over. Artyom knows how networks differ and has decided where to begin. **Next:** Part 5, Chapter 22. Your first five minutes in RpcNode.

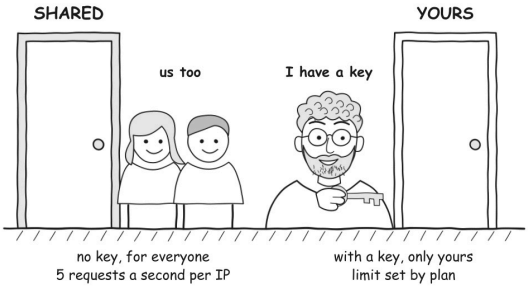
Chapter 22. Your first five minutes in RpcNode

“I want a door of my own”

All through the book Artyom has been knocking on public nodes. They answered without any sign-up, but that door is shared. A shop needs its own. Does it take long to make? What will he have to fill in?

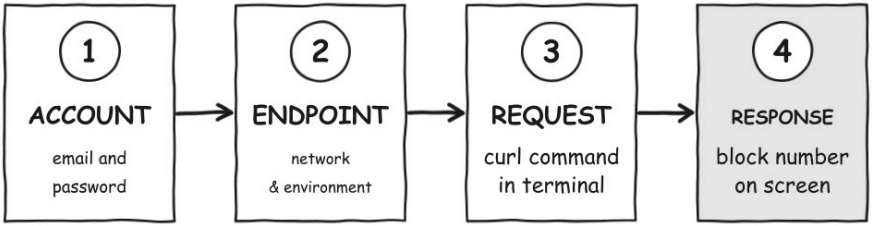
Digging in: the shared door and your own

The address at which a program asks a node questions is called an **endpoint** (Chapter 3). At RpcNode they come in two kinds.



	Public address	Personal endpoint
Sign-up	not needed	needed
Key	none	built into the address
Limit	5 requests per second per IP	by plan
Heavy methods	closed	open
Archive (Chapter 24)	closed	on paid plans
WebSocket and gRPC	no	where the network supports them

Your own endpoint appears in four steps.



Step 1. Sign up

Open `cabinet.rpcnode.dev/register` or press "Get started" on rpcnode.dev. The cabinet is your page in the service: your endpoints, usage and plan are here.

1. Enter your **Email**, make up a **Password** and repeat it. The name is optional.
2. Read and tick "I agree to the terms of service". Pass the "are you not a robot" check if it appears.
3. Press "Continue".
4. A one-time code will arrive by email. Enter it and press "Verify and continue".

There is a shortcut too: sign in with Google or GitHub.

A new account gets the **Free** plan, no card needed. It includes 15 million credits a month, 15 requests per second, 3 endpoints and two networks: Ethereum and Solana. We will look at what credits are in Chapter 25.

Step 2. The endpoint

In the cabinet menu choose "Endpoints" and press "New endpoint". A three-step window opens:

Step in the window	What to do
Select network	for example, Ethereum
Select environment	the main network or a test one (more on them below)
Confirm	check and press "Create"

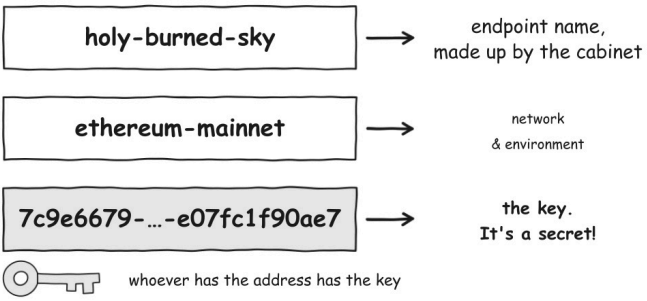
The **environment** is the variant of the network: real or practice. The cabinet makes up the endpoint's name itself, from three English words.

On the Free plan you will see a warning about archive nodes before creating. It isn't an error; Chapter 24 is about it.

What the address is made of

Here is an example from the RpcNode documentation. It is a single line, broken here before the slash:

```
https://holy-burned-sky-ethereum-mainnet.rpcnode.dev
/7c9e6679-7425-40de-944b-e07fc1f90ae7
```



The long tail after the slash is called a **UUID**: a number that is practically impossible to guess. It is the key. You don't need to attach a separate password to a request.

Copy the address with the button in the cabinet; don't type it by hand. Part of the key is hidden on screen, but it is copied in full.

The menu also has "API keys". That is something else: they sign commands to the service itself, for example "create a payment". Ordinary questions to a node don't need them.

Step 3. Try it: your first request

We put the address into a setting and ask for the number of the latest block. This is a **sketch**: replace the placeholder with your own address.

```
export RPC_URL='YOUR_ADDRESS_FROM_THE_CABINET'
curl -s "$RPC_URL" \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
```

Only you have your personal address, so in the book we show the response to the same lines with the public one: `RPC_URL='https://public-ethereum-mainnet.rpcnode.dev'`.

```
{"jsonrpc":"2.0","id":1,"result":"0x19464e4"}
```

`0x19464e4` is block number 26,502,372. In Chapter 3 the node named 26,502,035: the network has managed to add 337 blocks.

For Solana write `getSlot` instead of `eth_blockNumber`.

The cabinet's "Overview" page has a "Get started" list. In it you will find a ready command with your address already in place and the caption "Waiting for the first request…". And on the endpoint page, on the "Live" tab, you can see the requests themselves: time, method, response and latency.

If it didn't work

Let's deliberately ask with a made-up key of all zeros:

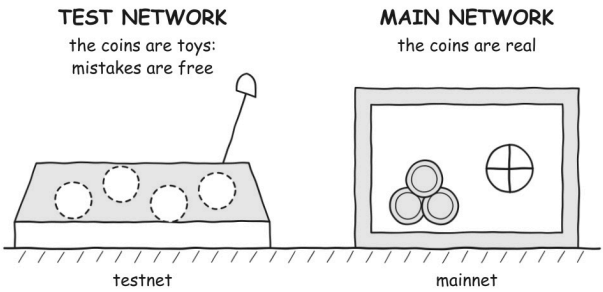
```
HOST=https://holy-burned-sky-ethereum-mainnet.rpcnode.dev
KEY=00000000-0000-0000-0000-000000000000
curl -s -w '  HTTP %{http_code}\n' $HOST/$KEY \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32000,
"message":"endpoint not found"}}  HTTP 404
```

`endpoint not found` means "there is no such endpoint". Check that the address was copied in full. Don't repeat a wrong address in a loop: for persistent guessing you will be ignored for a while.

Test networks: a training ground with toy money

A **test network** (testnet) works by the same rules as the real one, but its coins are worth nothing. The real one is called **mainnet**. Here mistakes are free.



Test coins come from a "faucet": a page that sends them to the address you give it.

Network	Test environments in RpcNode, for example
Ethereum	Sepolia, Hoodi
Solana	testnet, devnet
Bitcoin	testnet4, signet
TRON	Nile, Shasta

For applications on Ethereum, the ethereum.org website recommends Sepolia. Hoodi is for those who test the network itself and validators.

There are also test networks without sign-up: as of November 27, 2026, the list of public nodes includes Bitcoin testnet4, TRON Nile and TON testnet. Let's ask TRON, real and test, for the network number (Chapter 16):

```
for host in public-tron-mainnet public-tron-nile; do
  echo -n "$host: "
  curl -s https://$host.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_chainId","params":[]}'
  echo; sleep 2
done
```

```
public-tron-mainnet: {"jsonrpc":"2.0","id":1,"result":"0x2b6653dc"}
public-tron-nile: {"jsonrpc":"2.0","id":1,"result":"0xcd8690dc"}
```

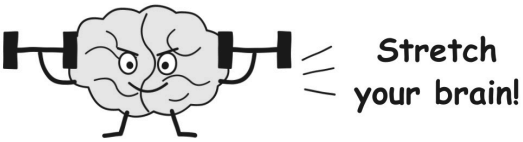
The numbers differ: these are two different notebooks. An endpoint, too, is tied to one network and one environment. To move from the test one to the real one, you create a new endpoint.

The trap: "I'll paste the address right into the web page"

Anything that ends up in a page's code is visible to every visitor. They will take your address and spend your quota.

- Query the node from your **server**, not from the customer's browser.
- Keep the address in server settings, not in the code or in a shared code repository.
- On the endpoint page, under "Settings", fill in the "IP allowlist": then someone else's computer will be refused.
- "Allowed origins" stop others' websites from calling the endpoint from a browser. But this list doesn't protect against a program outside a browser.
- Has the address leaked? Create a new one, switch the website over and delete the old one: it can't be brought back.

Think it over



1. Artyom wrote the endpoint address into a script on the shop's page to display the block number. Who will see the key?
2. Everything has been tested on Sepolia. Can you accept real payments with the same endpoint?
3. You get `endpoint not found`. Where do you start looking?

Answers: 1) every visitor. 2) No: you need an endpoint on mainnet. 3) Compare the address with the one in the cabinet: copy it again in full.

Key points

- A personal endpoint is an address with a key inside: guard it like a password.
- An endpoint is tied to one network and one environment.
- Test network first, then the real one.

Next: Chapter 23. What happens to a request: the gateway, node selection and three geographies.

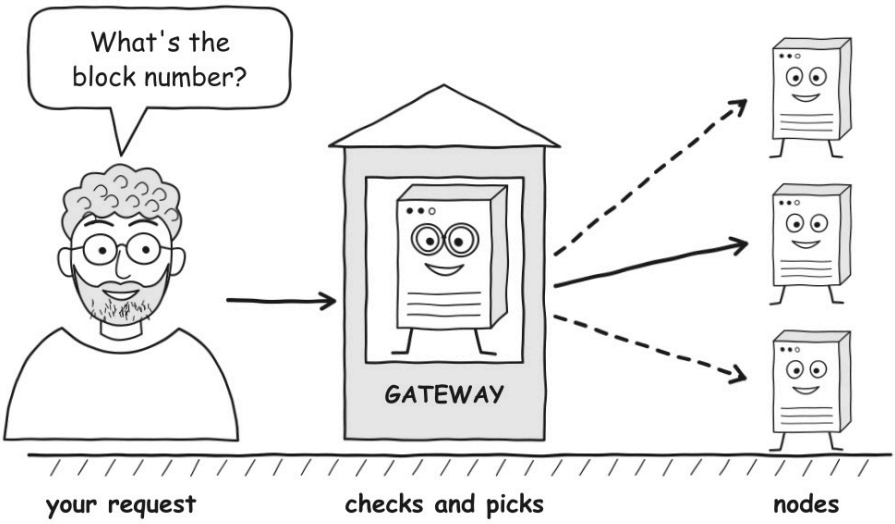
Chapter 23. What happens to a request

“So who answered me?”

Artyom sent a request and got a block number. But where did the question go? Which node answered? And what happens if it breaks in the middle of a sale?

Digging in: a gateway stands at the entrance

Your request doesn’t go straight into a node. First it reaches the **gateway**. This is a dispatcher program: it receives requests, checks them and passes them to a suitable node.



What the gateway checks along the way

The procedure is like a security checkpoint. Fail one check and the request goes no further.

Check	If it fails
Does this endpoint exist, is the address correct	HTTP 404, `-32000`
Is it switched on; are your IP and website allowed	HTTP 403, `-32000`
Are you asking too often	HTTP 429, `-32005`
Does this network have such a method	HTTP 404, `-32601`
Are there enough credits	HTTP 402, `-32006`

Each network has its own list of methods. Let's ask for something that isn't on it:

```
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_noSuchMethod","params":[]}'
```

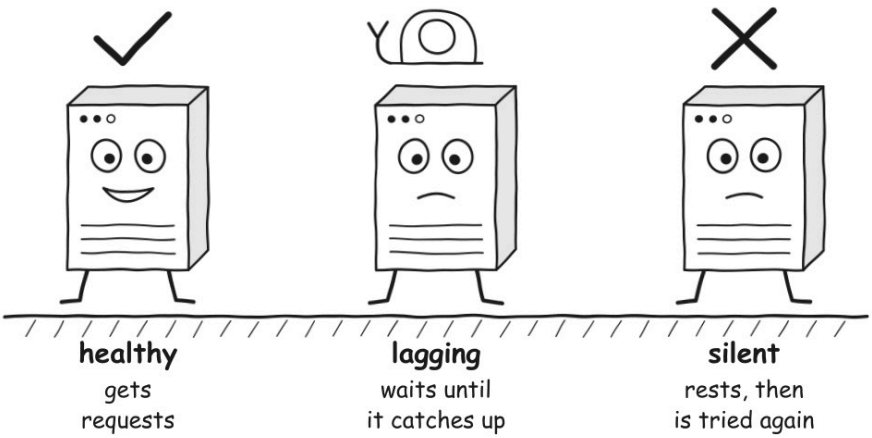
```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,
"message":"method not found: eth_noSuchMethod"}}  HTTP 404
```

This question never reached a node: the gateway answered by itself.

Choosing a node: load balancing

A network may have several nodes. Handing out requests among them is called **load balancing**. It is like checkouts in a store: a new shopper is sent where the line is shorter.

First the gateway looks at whether a node is in good shape.



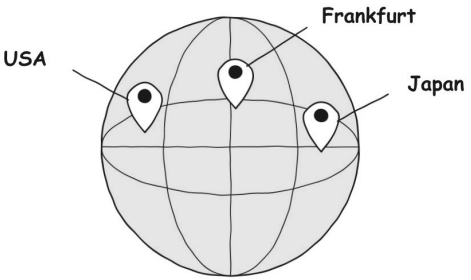
- A **lagging** node answers cheerfully, but with yesterday's data (Chapter 12). While there are up-to-date ones, requests go to them.

- Among healthy nodes, a busy one gets less and a free one more.
- The connection broke or the node returned an internal error — the same request goes to the next one. You don't notice.
- A node that has failed several times in a row is left alone for a while and then tried again.

There is a caveat. If no healthy nodes are left at all, the gateway still tries the lagging ones: sometimes an old answer is better than none. So where freshness matters, check the block time yourself (Chapter 12).

Three geographies

RpcNode operates in three geographies: Frankfurt, the USA and Japan. The closer the node, the shorter the trip there and back. The gateway tries to take the healthy node nearest to it and keeps the far ones in reserve.



RpcNode's three geographies (a sketch, not a map)

Not every network has nodes in all three places. The page `rpcnode.dev/status` shows where and how the network you need is answering.

Try it: read the marks on the envelope

Along with the response come **headers** — service marks, like stamps on an envelope. Two of them tell which country the gateway was in and which the node was in:

```
for net in ethereum base arb; do
  echo "== $net"
  curl -s -D - -o /dev/null https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}' |
    grep -iE 'x-gateway-region|x-upstream-country'
  sleep 2
done
```

```
== ethereum
x-gateway-region: DE
x-upstream-country: DE
== base
x-gateway-region: DE
x-upstream-country: JP
== arb
x-gateway-region: DE
x-upstream-country: JP
```

`DE` is Germany, `JP` is Japan. This time a gateway in Germany accepted all the requests, and went to Japan for Base and Arbitrum. Your letters may be different. These are service marks: don't build your website's operation on them.

A node's error is an answer too

Let's ask the USDT contract: what happens if you transfer a huge sum from an empty address? The `eth_call` method only tries it out, sending nothing. The `data` field glues together the number of the transfer function, the recipient (`TO`) and the amount (`SUM`).

```
TO=0000000000000000000000000000000000000000000000000000000000000dead
SUM=0000000000000000000000000000000000000000000000000fffffffffffffff
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[{
  "from":"0x0000000000000000000000000000000000000dEaD",
  "to":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "data":"0xa9059cbb'$TO$SUM'"},"latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32000,
"message":"invalid opcode: INVALID"}}  HTTP 200
```

That is how this old contract says "no way". The node did its job: it honestly reported that the program stopped. Another node would answer the same, so the gateway passes such a response to you as it is, with HTTP 200.

When nobody answered

It can be worse: the gateway has gone through the nodes and there is no answer. Then you get code `-32002`, the text `upstream unavailable` (the node is unreachable) and an HTTP code from the 500s. Credits for such a call are refunded.

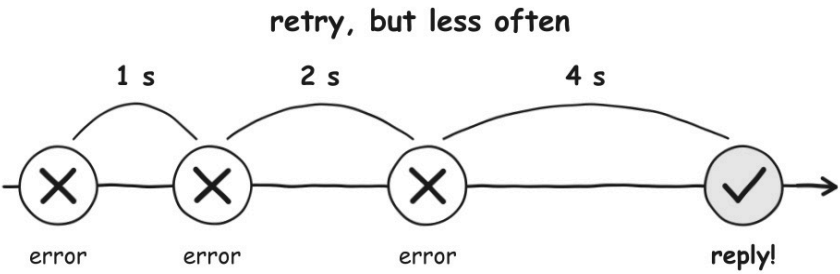
And what if it is sending money?

If a node **refused** to send a transaction, that is the answer. But if it **didn't respond** at all, the gateway may pass the same signed transfer to another node.

The money won't go twice because of this: a signed transaction has one and the same hash, and the network accepts it once. But if **you** didn't get an answer, don't sign a new transaction on a guess. First look for the old one by its hash (Chapter 7).

Try it: retry wisely

The gateway goes through nodes, but the link between you and the gateway can fail too. Retry yourself, with growing pauses: 1, 2, 4, 8 seconds. This is called **backoff**.



The file `ask.js`, run with `node ask.js` (requires Node.js):

```
const url = 'https://public-ethereum-mainnet.rpcnode.dev'
const sleep = (ms) => new Promise((ok) => setTimeout(ok, ms))

async function ask(method, params = []) {
  for (const pause of [1, 2, 4, 8]) {
    try {
      const res = await fetch(url, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ jsonrpc: '2.0', id: 1, method, params }),
        signal: AbortSignal.timeout(10000),
      })
      if (res.status !== 429 && res.status < 500) return await res.json()
      console.log('HTTP', res.status)
    } catch {
      console.log('no connection')
    }
    await sleep(pause * 1000)
  }
  throw new Error('could not get an answer')
}

ask('eth_blockNumber').then(console.log)
```

```
{ jsonrpc: '2.0', id: 1, result: '0x19464ee' }
```

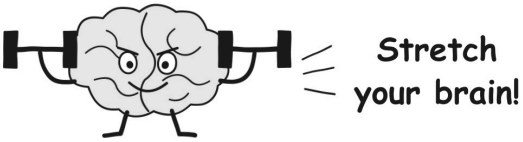
You retry only what may succeed on the second try: "too often", a dropped connection, the 500s. Retrying an error in the request itself is useless. And don't forget to check the `error` field after the response (Chapter 11).

The trap: "no answer" doesn't mean "no payment"

If you couldn't ask, the website **doesn't know** whether the customer paid. Write "checking your payment", not "unpaid", and ask later.

And remember: a provider with several nodes goes down less often than a single node, but not "never" (Chapter 3).

Think it over



1. You got HTTP 200 and `execution reverted`. Will retrying the request five times help?
2. A program asked for `eth_blokNumber`. Did the request reach a node?
3. Artyom sent a transfer and never got a response. What should he do first?

Answers: 1) no: it is a real answer, not a failure. 2) No, the gateway rejected the unknown method. 3) Look for the transaction by its hash.

Key points

- The gateway checks a request and picks a healthy node; if one doesn't answer, it tries another.
- A node's own error about your request reaches you as it is.
- Your program still has to retry requests with pauses.

Next: Chapter 24. History and archive data.

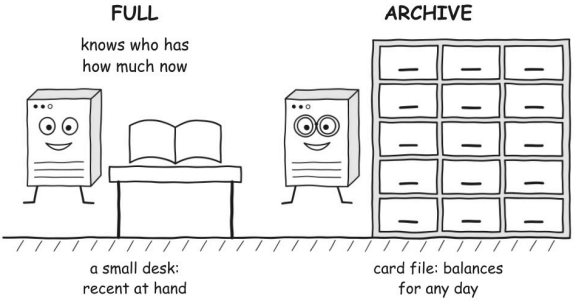
Chapter 24. History and archive data

“How much was there on January 1?”

The accountant asks Artyom: “Tell me how much sat on the shop’s address at the start of the year.” A node answers about today instantly. But about back then?

Digging in: records and balances

Let’s recall Chapter 10. **History** is blocks and transfers. **State** is who owned what at a particular moment.



A **full node** keeps recent balances at hand. An **archive** node remembers balances at any block, but needs far more disk.

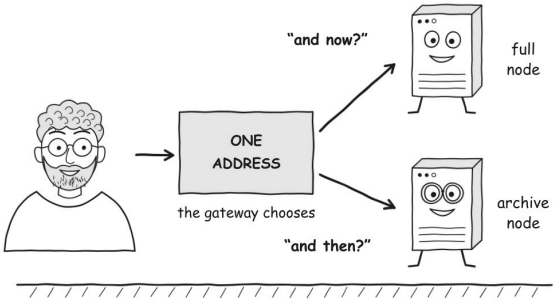
Which questions need the archive

Question	Example
A balance or a contract’s answer at an old block	`eth_getBalance`, `eth_call` with a block number
Events from past blocks	`eth_getLogs`
Replaying a transaction step by step (tracing)	`trace_…`, `debug_…`
A proof about the state	`eth_getProof` at an old block
Long-past Solana slots	`getBlock` with an old number

But “how much is on the account now” and an ordinary recent payment don’t need an archive. A shop usually doesn’t need one in day-to-day work.

How it works in RpcNode

There is one endpoint address. You don't have to choose a node yourself: the gateway looks at the question and decides whom to give it to.



Archive access is included in paid plans; Free doesn't have it. When creating an endpoint on Free, you will see this in the cabinet:

> "Archive nodes are not available on the free plan. Your requests are served by regular nodes. They answer everything they hold. What only an archive node keeps (old state, deep history, some traces) is not available: the node returns its own error."

- There is no separate price for an "old" question: you pay for the method (Chapter 25).
- Archive depth differs between networks. Before building something important on it, test your oldest question on your network.
- Public addresses don't serve archive questions.

Try it: the public node is strict

Let's ask for a balance not as of "now" but at the number of a block from just ten blocks ago, about two minutes:

```
E=https://public-ethereum-mainnet.rpcnode.dev
H='Content-Type: application/json'
N=$(curl -s $E -H "$H" \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}' |
  grep -o '0x[0-9a-f]*')
OLD=$(printf '0x%x' $((N - 10)))
sleep 2
curl -s -w '  HTTP %{http_code}\n' $E -H "$H" \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance",
  "params":["0xdac17f958d2ee523a2206206994597c13d831ec7","'$OLD'"]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,"message":
"archive data is not available on the public endpoint,
use a personal endpoint"}}  HTTP 403
```

The public address treats even such a recent past as archive. Query it with the words from Chapter 8: `latest`, `safe` or `finalized`.

A second probe: the node may have thrown it out

A block is history, not state. But not everyone keeps even that. Since July 2025, Ethereum node software can skip holding the contents of blocks from before the switch to staking (Chapter 10). Let's ask for block 46,147 (`0xb443`) from August 2015:

```
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockByNumber",
  "params":["0xb443",false]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":4444,
"message":"pruned history unavailable"}}  HTTP 200
```

"Pruned history unavailable." The node honestly says: I don't have this. On a paid plan, when an ordinary node says "I don't have this", the gateway can ask an archive node again, if the network has one.

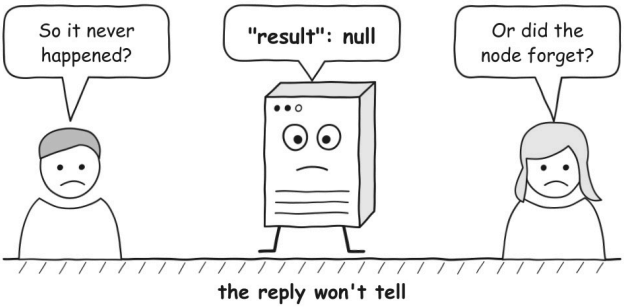
The trap: "null means it never happened"

There is a transfer in that very block. Let's ask about it by hash, and then about a hash we simply made up:

```
for hash in \
  0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060 \
  0x0000000000000000000000000000000000000000000000000000000000000001
do
  curl -s https://public-ethereum-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionByHash",
    "params":["'$hash'"]}'
  echo; sleep 2
done
```

```
{"jsonrpc":"2.0","id":1,"result":null}
{"jsonrpc":"2.0","id":1,"result":null}
```

The answers are identical! The first transfer really happened; the second never existed.



Don't draw conclusions from a single `null`, especially about old records. Sometimes it means "it never happened", sometimes "I don't keep it", and for a fresh transfer, "not in a block yet".

How much a Solana node remembers

Solana makes several blocks a second, and records pile up in mountains. Let's ask for the number of the latest slot and the oldest block the node still holds:

```
for method in getSlot getFirstAvailableBlock; do
  echo -n "$method: "
  curl -s https://public-solana-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"'$method'"}'
  echo; sleep 2
done
```

```
getSlot: {"jsonrpc":"2.0","result":474316453,"id":1}
getFirstAvailableBlock: {"jsonrpc":"2.0","result":473916707,"id":1}
```

The difference is about 400,000 slots. When checked, that was just one day. And block number 1000 was cleared away long ago:

```
curl -s https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getBlockTime","params":[1000]}'
```

```
{"jsonrpc":"2.0","error":{"code":-32001,"message":"Block 1000 cleaned up,
does not exist on node. First available block: 473916707"},"id":1}
```

Back to the accountant

The balance "at the start of the year" is the balance after the last block of the old year (we count time in universal time, UTC). A node won't name a block number by date right away. But blocks go in order, so you can search by **halving**: take the block in the middle, look at its time and throw away the half you don't need.

Let's find the first block of 2026. The script takes about a minute:

```
E=https://public-ethereum-mainnet.rpcnode.dev
ask() {  # method, params
  curl -s $E -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"'$1'","params":'$2'}'
  sleep 2
}
WHEN=1767225600   # January 1, 2026, 00:00 UTC
low=20000000
high=$(( $(ask eth_blockNumber '[]' | grep -o '0x[0-9a-f]*') ))
steps=0
while [ $low -lt $high ]; do
  mid=$(( (low + high) / 2 ))
  hex=$(printf '0x%x' $mid)
  t=$(ask eth_getBlockByNumber '["'$hex'",false]' |
    grep -o '"timestamp":"0x[0-9a-f]*"' | grep -o '0x[0-9a-f]*')
  if [ -z "$t" ]; then echo "no answer"; exit 1; fi
  if [ $(( t )) -lt $WHEN ]; then low=$(( mid + 1 )); else high=$mid; fi
  steps=$(( steps + 1 ))
done
echo "block $low, questions: $steps"
```

```
block 24136053, questions: 23
```

Out of millions of blocks, the right one was found in 23 questions (your count may differ by one). So the last block of 2025 is 24,136,052 (`0x1704974`). You ask for the balance at it the same way as in the first probe, only with this number. The public address will refuse; you need a personal endpoint on a paid plan. For a token, instead of `eth_getBalance` you use `eth_call` with a `balanceOf` question to the contract.

A long stretch goes in pieces

A node most likely can't handle "all events for a year": the response is too big. The stretch of blocks is cut into parts, which are asked for in turn, and the answers are added together.

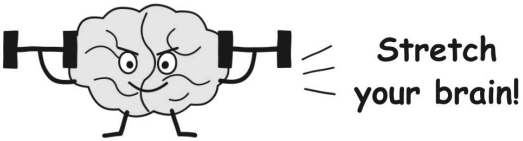
This is a **sketch**: `rpc` is your function that queries a personal endpoint, something like `ask` from Chapter 23.

```
async function logsByChunks(rpc, filter, from, to, size = 1000) {
  const all = []
  for (let a = from; a <= to; a += size) {
    const b = Math.min(a + size - 1, to)
    const answer = await rpc('eth_getLogs', [{
      ...filter,
      fromBlock: '0x' + a.toString(16),
      toBlock: '0x' + b.toString(16),
    }])
    if (answer.error) throw new Error(answer.error.message)
    all.push(...answer.result)
  }
  return all
}
```

If it doesn't go through, make the piece smaller; each one is a separate paid call. And a shop's cheapest archive is **its own**: write payments into your database as soon as you see them

(Chapter 26).

Think it over



1. A node returned `null` for the hash of an old transfer. Does that prove the customer didn't pay?
2. You need the USDC balance as of July 1. Is the Free plan enough?
3. You need to go through the events of 5000 blocks in pieces of 1000. How many calls will that be?

Answers: 1) no: the node may not have kept that record. 2) Free doesn't include the archive; you can't count on it. 3) Five.

Key points

- Old balances, old events and traces are work for an archive node.
- Archive access comes with paid plans; the endpoint address stays the same.
- `null` and an empty response don't yet prove that nothing happened.

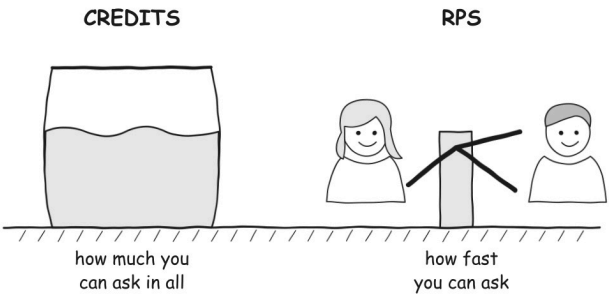
Next: Chapter 25. Plans, the price of a call and limits.

Chapter 25. Plans, the price of a call and limits

“And how much will that be a month?”

The endpoint works. Now Artyom wants to know what the shop will cost. The pricing page has two main numbers: credits and requests per second. Which is which?

Digging in: two different measures



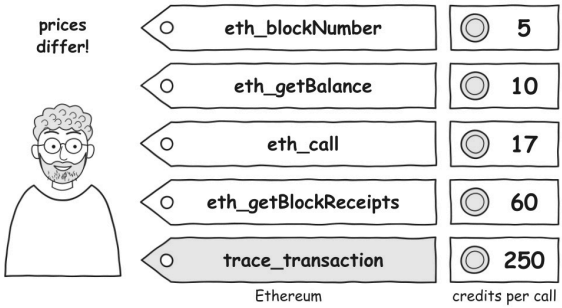
Credits are the service’s internal “money”: how much you can ask in total in a month. **RPS** means requests per second: how fast you can ask. You need enough of both.

Try it: find out the price of a call

Different questions cost different amounts: for one the node answers from memory, for another it goes through a whole block. Prices are public and can be queried without sign-up. Let’s take a few Ethereum methods (you will need `python3`):

```
curl -s https://rpcnode.dev/api/v1/public/networks/ethereum/methods |
python3 -c 'import json, sys
want = """eth_blockNumber eth_getBalance eth_call
eth_getLogs eth_getBlockReceipts trace_transaction""".split()
for m in json.load(sys.stdin)["methods"]:
    if m["method"] in want and m["variant"] == "default":
        print(m["method"], m["credits"])'
```

```
eth_blockNumber 5
eth_getBalance 10
eth_call 17
eth_getLogs 17
eth_getBlockReceipts 60
trace_transaction 250
```



- One `trace_transaction` costs as much as 50 questions about the block number.
- Each network has its own price list. Replace `ethereum` in the address with `solana` or `tron` and change the method names.
- A batch (Chapter 11) costs the sum of its calls.
- WebSocket subscriptions are paid too. In Ethereum, opening a subscription to new blocks costs 10 credits, and each event sent costs 2.

In the cabinet the prices are gathered in the "Networks" section.

Plans

Data as of November 27, 2026. For current figures see `rpcnode.dev/pricing`.

Plan	$ per month	$ per year	Credits per month	RPS	Endpoints
Free	0	—	15M	15	3
Starter	19	190	50M	40	5
Build	36	360	100M	75	12
Grow	99	990	300M	150	25
Scale	219	2,190	700M	300	50
Business	449	4,490	1.5B	500	100

A plan isn't only RPC. Here is what else it includes (Chapters 26 and 27):

Plan	Watched addresses	Payments: open / per month	Transaction watches	Archive and all networks
Free	—	—	—	no: only Ethereum and Solana
Starter	15	10 / 200	15	yes
Build	50	25 / 1,000	40	yes
Grow	150	50 / 5,000	80	yes
Scale	400	100 / 20,000	150	yes
Business	1,000	250 / 100,000	400	yes

- A new account can try Starter for 7 days once, free and without a card.
- If the ready-made plans don't fit, the pricing page has "Build your own".
- To buy a plan, press "Buy" and confirm. An invoice valid for 15 minutes appears; it can be paid in cryptocurrency.
- When the term ends and there is no new payment, the account moves to Free. Set yourself a reminder.

Let's do the math for Artyom's shop

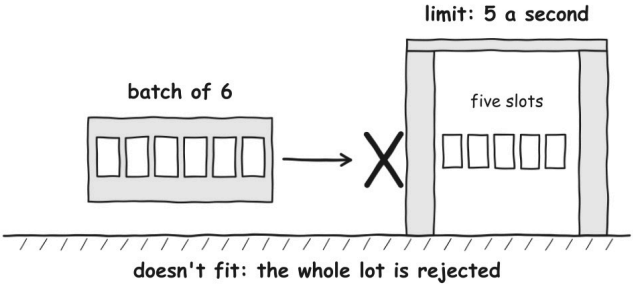
Twenty orders are awaiting payment, each with its own address on Ethereum. We take 30 days and assume orders are always waiting: this is an upper estimate.

How the website asks	Requests	Credits	Plan that is enough
each address's balance every 10 s	5,184,000	51,840,000	Build
the same every 30 s	1,728,000	17,280,000	Starter
block number and one `eth_getLogs` every 12 s	432,000	4,752,000	Free

The third way is more than ten times cheaper than the first: one question about all the addresses at once. But the comparison comes with a caveat: the first two ways notice incoming ETH, while the third notices token transfers (Chapter 15). And Free has only two networks, Ethereum and Solana.

The per-second limit

The RPS counter resets every second and, according to the RpcNode documentation, each endpoint has its own. An extra request gets HTTP 429 and code `-32005`. **A batch of N calls is N requests**, and it goes through completely or not at all.



Let's check on the public address, where the limit is 5. One batch of six questions is enough:

```
q='{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
curl -s -i https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d "[$q,$q,$q,$q,$q,$q]" | grep -iE '^(HTTP|retry-after)|error'
echo
```

```
HTTP/2 429
retry-after: 1
{"jsonrpc":"2.0","id":null,"error":{"code":-32005,"message":
"rps limit exceeded: a batch of 6 calls counts as 6 requests
(the limit is 5 a second)"}}
```

`retry-after: 1` means "try again in a second". The public address also guards expensive methods:

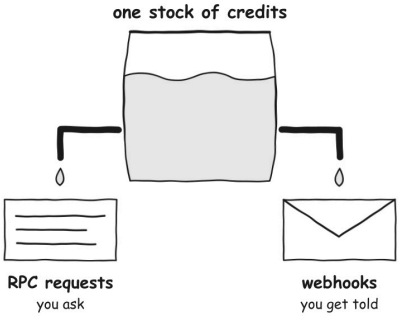
```
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockReceipts","params":["latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,"message":
"method eth_getBlockReceipts is not available on the public endpoint,
use a personal endpoint"}}  HTTP 403
```

What is charged and what isn't

What happened	Credits
The node answered with a result	charged
The node answered with an error about your request, for example `execution reverted`	charged: that is an answer too
Rejected by the per-second limit	not charged
The method isn't on the network's list	not charged
No node answered	refunded
A request to a public address	free

One stock for everything



The same stock pays for **webhooks** — messages the service itself sends to your website (Chapter 26). On the Ethereum main network, an event on a watched address costs 450 credits, a payment 750. On Ethereum test networks it is half that.

Check the balance in the cabinet, in the "Usage" section: the most frequent methods are there too.

When credits run out

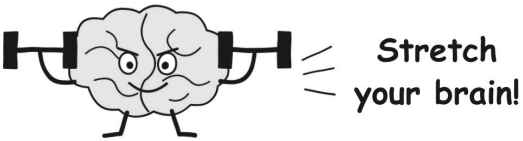
There will be no overage bill: there are no extra charges beyond the package. But don't count on the service carrying on as if nothing had happened. Build the refusal from Chapter 11 (`-32006`) into your program and keep an eye on the balance in advance.

The trap: choosing a plan by its price tag

First work out **how many times × the method's price**. A few heavy calls can eat more than thousands of light ones.

Method prices and terms may change. Under the terms of service, plan payments aren't refunded unless the law requires otherwise, and uninterrupted operation isn't guaranteed: for a major outage on a paid plan, the paid period is extended. Read the terms before you pay.

Think it over



1. A website calls `eth_call` (17 credits) every 10 seconds all month, 30 days. How many credits is that?
2. On the Free plan someone sent a batch of 20 calls. What happens?
3. A request got a 429. Were credits charged for it?

Answers: 1) 259,200 × 17 = 4,406,400. 2) It is rejected as a whole: 20 is more than 15. 3) No.

Key points

- Credits limit the volume per month, RPS the speed.
- Every method has its price; a batch is counted by the number of calls.
- Asking more cleverly is often better value than buying a bigger plan.

Next: Chapter 26. Address events: webhooks and sockets instead of constant polling.

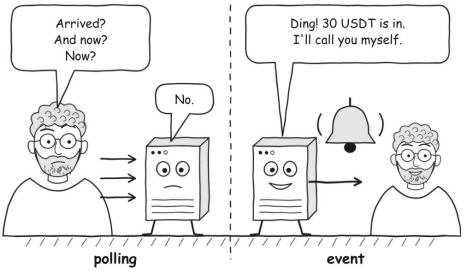
Chapter 26. Address events: webhooks and sockets instead of constant polling

“Twenty orders means twenty questions a second?”

In Chapter 3 we worked it out: twenty pending orders, each checked once a second, and that is already more than the 15 requests per second the Free plan allows. And almost all the answers are the same: “not arrived”.

Digging in: ask, or wait for a call

Polling is when your website asks again and again by itself: “Has it arrived?” An **event** is when you are told, as soon as something happens.



Try it: one question about all the addresses

First let’s ask more cleverly. There is no need to ask about each order separately. The `eth_getLogs` method from Chapter 15 can be given a list of recipients at once:

```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getLogs","params":[{
  "address":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
  null,
  ["0x000000000000000000000000000000000000000000000000000000000000dead",
   "0x0000000000000000000000000000000000000000000000000000000000000001",
   "0x000000000000000000000000dac17f958d2ee523a2206206994597c13d831ec7"]],
  "fromBlock":"latest","toBlock":"latest"}]}'
```

The question is: “Are there USDT transfers in the latest block to any of these three addresses, from anyone at all?” The addresses are padded with zeros on the left to 64 characters. We got this answer:

```
{"jsonrpc":"2.0","id":1,"result":[]}
```

An empty list: nobody paid in this block. There could be twenty addresses, and it would still be one request. But you would have to ask about **every** new block, count confirmations yourself, and catch transfers of ETH itself separately (Chapter 15).

Address Watch: let someone watch for you

Address Watch, address tracking, takes this work on. RpcNode goes through blocks and, when a transaction touches your address, tells your server.

An address is added in the cabinet: "Notifications" → "Addresses" → "Add address". A program does the same with a `POST /v1/address-watches` request with the body `{"network_slug": "tron", "address": "YOUR_ADDRESS"}`.

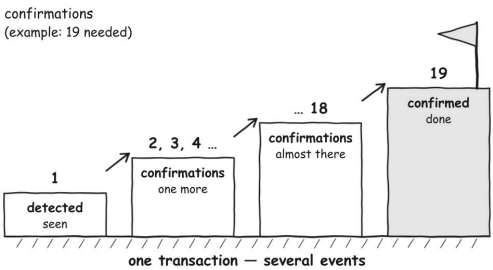
The response contains a `watch_id`, the number of the watch. One address in one network takes up one **slot** of your plan. Delete the watch and the slot is free, but additions and deletions have a limit too. Starter has 15 slots. Free has none on main networks, only a "sandbox": a few addresses on test networks.

You can get an endpoint for more networks than are processed for events. As of November 27, 2026 these are Ethereum, Bitcoin, Solana, TRON, BNB Smart Chain, Base, Arbitrum One, TON, Litecoin, Dogecoin, Bitcoin Cash, Dash, Mantle and Monad, and among test networks only TRON Nile.

Money coming in is an event of type `payment`, money going out is `payout`, everything else (an approve allowance, say) is `address`. New blocks arrive as type `block`, while `tx` and `payment_order` are covered in Chapter 27.

One transaction, several calls

You are told about a transfer not once but as it "hardens" (Chapter 8). What exactly happened is stated in the `event` field:



event	Means	What the shop should do
`detected`	the transaction has been seen in a block	show “we see your transfer”
`confirmations`	one more confirmation	update the counter
`confirmed`	enough confirmations, success	consider the money received
`failed`	enough confirmations, the transaction failed	don’t ship

How many confirmations to wait for is set under “Settings” → “Confirmations”; if you don’t set it, the network’s default applies.

In Bitcoin, Litecoin and Dash, `pending` may arrive before all the others: the transfer is still in the mempool (Chapter 7). Its type is always `address`, not `payment`, and it isn’t money: such a transfer can be replaced.

A webhook: a letter in your mailbox

A **webhook** is a request the service itself sends to your address on the internet. The address and the per-network subscriptions are set under “Notifications” → “Delivery settings”. The address must be reachable from the internet: `localhost` won’t do.

If you turn on a webhook without selecting anything, you will get incoming and outgoing transfers on all networks, but not blocks.

Here is part of an example from the documentation — the first call about 30 USDT on TRON:

```
{ "delivery_id": "3f9c…6b3d90",
  "type": "payment", "event": "detected", "network": "tron",
  "tx_hash": "bb08…c320c", "address": "TToL…QRzRG",
  "direction": "in", "currency": "USDT",
  "amount": "30", "amount_raw": "30000000",
  "confirmations": 1, "required_confirmations": 19,
  "confirmed": false }
```

Amounts come as strings so that nothing gets rounded; `amount_raw` is the same without the decimal point (Chapter 9). A token that isn’t in the service’s catalog will have only `amount_raw`. The headers carry `X-RpcNode-Delivery` (delivery number), `X-RpcNode-Event` (type) and `X-RpcNode-Signature` (signature).

“Is that really you?”

A webhook address is open: anyone can knock and write “you’ve been paid”. So every delivery is signed. The time, a dot and the body are taken, and a fingerprint of them is computed with the webhook secret. This method is called **HMAC-SHA256**. The secret is in the “Secret and IPs” block; it isn’t the API key secret.

Try it: sign it yourself

The secret and body here are made up:

```
printf '%s' '1787347206.{"type":"test"}' |
  openssl dgst -sha256 -hmac 'demo-secret' | awk '{print $NF}'
```

```
e17efb57778b0e0c758a9bf2c5705049c4c383d2af3a2bd2783c8b665c151cde
```

Change even one character in the body or the secret and you get a completely different string. Your server computes the same and compares it with a header of the form `t=…,v1=…`.

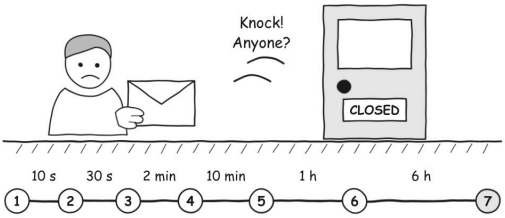
Here is a **sketch** of a receiver in Node.js. We tested it with homemade requests, but not with real deliveries: those require a cabinet account. The 300-second check keeps anyone from sending yesterday's delivery again.

```
import { createServer } from 'node:http'
import { createHmac, timingSafeEqual } from 'node:crypto'

const SECRET = process.env.RPCNODE_WEBHOOK_SECRET
const seen = new Set()      // in a real shop this is a database
createServer(async (req, res) => {
  const parts = []
  for await (const part of req) parts.push(part)
  const body = Buffer.concat(parts)   // the bytes as they arrived
  const [, t, v1] = /^t=(\d+),v1=([0-9a-f]{64})$/
    .exec(req.headers['x-rpcnode-signature'] ?? '') ?? []
  const ours = createHmac('sha256', SECRET)
    .update(`${t}.`).update(body).digest()
  const ok = v1 && Math.abs(Date.now() / 1000 - t) < 300 &&
    timingSafeEqual(ours, Buffer.from(v1, 'hex'))
  if (!ok) return res.writeHead(401).end()
  const id = req.headers['x-rpcnode-delivery']
  if (!seen.has(id)) {
    seen.add(id)
    console.log(id, body.toString())  // your work goes here
  }
  res.writeHead(200).end()
}).listen(3000)
```

If you weren't home

A delivery succeeds if the server answers with a code from the 200s within 10 seconds. Otherwise it is repeated, and after the seventh failure it is dropped. So:

- One event may arrive twice. The delivery number is the same — filter by it.
- Order isn't promised: `confirmations` may overtake `detected`.
- Do long work **after** responding.



pauses grow; seven tries in all, about seven hours

Check your address in advance with a test delivery: it is free. The delivery log in the cabinet keeps entries for a day.

A socket: an open line

The second way is **WebSocket**: your program opens the connection itself and listens, so you don't need your own address on the internet for this. It isn't the `eth_subscribe` subscription from Chapter 11: there the node talks about the network, here the service talks about your addresses and orders. The body is the same as a webhook's, but what you missed while you were away is **not repeated**.

The protocol is compatible with Pusher (the `pusher-js` library will do), and the key is the number of your API key. The socket is turned on separately, with the "Deliver to WebSocket" switch. Turn on both ways and every message is delivered and paid for twice.

What it costs

Events spend the same credits as requests. As of November 27, 2026, an address event on the TRON or Ethereum main network costs 450 credits — **once per transaction**: confirmations and the final result are free. You pay for delivery even if your server then didn't accept it.

Let's estimate a 30-day month on Ethereum, where each request in the table costs 17 credits:

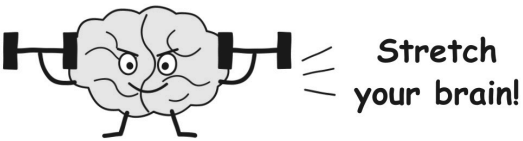
How we learn of a payment	Requests or events	Credits
20 orders, `eth_call` every 10 seconds (Chapter 13)	5,184,000	88,128,000
one `eth_getLogs` per block, every 12 seconds	216,000	3,672,000
Address Watch, 200 transfers a month	200	90,000

Starter has 50 million credits: the first way doesn't fit. This is only an example. The more transfers, the more the events cost, and other people's tiny transfers to your address count too.

The trap: "an event arrived, so I ship"

- **Not every event is a payment:** `detected` isn't final yet, and `payout` is your own spending.
- **A token is recognized by the contract address** in the `token` field, not by name (Chapter 9).
- **Test unusual transfers in advance,** for example ETH sent from inside a contract (Chapter 15).
- **Address events aren't an archive:** they are available through the API for a day.

Think it over



1. The shop's server was off for an hour. What happens to webhook deliveries, and what to socket messages?
2. Two deliveries arrived with the same `X-RpcNode-Delivery`. How many times do you credit the money?
3. In a month, 1000 transfers on TRON to watched addresses, delivered by webhook only. How many credits?

Answers: 1) webhooks are retried until the seven attempts run out; the socket doesn't repeat what was missed. 2) Once. 3) 450,000.

Key points

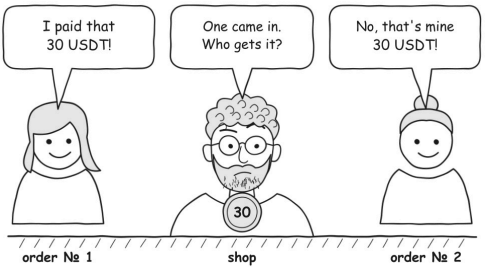
- Don't ask endlessly: let yourself be told about a transfer.
- Check the signature, filter out repeats, respond quickly.
- The money has arrived at `confirmed`, not at the first call.

Next: Chapter 27. Payments and transaction tracking.

Chapter 27. Payments and transaction tracking

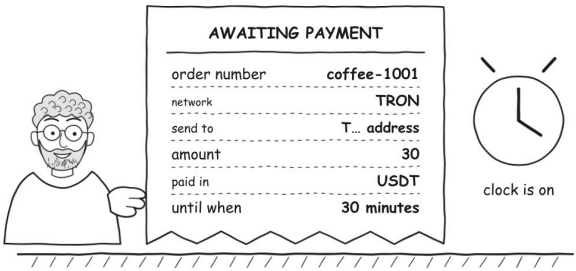
“The money arrived. But whose is it?”

Artyom got an event: 30 USDT arrived at an address. He opens the order list, and there are two for thirty. The event knows about the address and the amount but nothing about the customer’s cart.



Digging in: a payment is an expectation

A **payment** (the “Payments” section in the cabinet) is an expectation written down in advance: such-and-such an order is waiting for so much at such-and-such an address until such-and-such a time. When a matching transfer arrives, the service itself names the order number.



You supply your own address. The money goes from the customer straight to you, and the service doesn’t need your secret phrase (Chapter 6).

A payment is created with the “New payment” button or with a `POST /v1/payments` request:

Field	What it is
`order_id`	your order number, never repeated
`network_slug`, `address`	the network and your address in it
`amount`	the amount **as a string**: `"30"`, `"12.5"`
`token_contract`	the token contract; without it the network's coin is expected
`ttl_seconds`	the lifetime in seconds; without it an hour, at most 30 days

Try it: knock without a key

```
curl https://api.rpcnode.dev/v1/payments
```

```
{"ok":false,"error_code":"public_api.key_missing",
 "error":"Missing X-Api-Key header."}
```

This is a response with code 401: "and who are you?". Every request to this API is signed with the API key secret (the "API keys" section). Four lines are signed: the time, the method, the path and a fingerprint of the body.

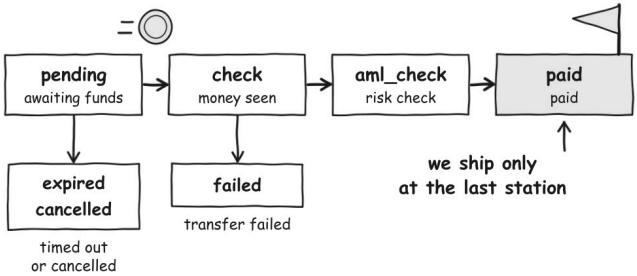
Here is a **sketch** in shell. We checked its syntax and the signature calculation but didn't send it: a personal key is needed.

```
BODY='{"order_id":"coffee-1001","network_slug":"tron",
"address":"YOUR_ADDRESS","amount":"30","ttl_seconds":1800,
"token_contract":"TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t"}'
TS=$(date +%s)
HASH=$(printf '%s' "$BODY" | openssl dgst -sha256 | awk '{print $NF}')
SIG=$(printf '%s\nPOST\n/v1/payments\n%s' "$TS" "$HASH" |
  openssl dgst -sha256 -hmac "$SECRET" | awk '{print $NF}')

curl https://api.rpcnode.dev/v1/payments \
  -H 'Content-Type: application/json' \
  -H "X-Api-Key: $KEY_ID" -H "X-Timestamp: $TS" \
  -H "X-Signature: $SIG" -d "$BODY"
```

The response contains a `payment_id`, the status `pending` and the deadline `expires_at`. If you repeat the request with the same `order_id` and the same network, address, token and amount, the same payment comes back. With different ones, you get the error `payment.order_exists`.

The road of a payment

Every step arrives as a `payment_order` event — by webhook or into the socket (Chapter 26).



`event`	What happened
`check`	the money has been seen, waiting for confirmations
`confirmations`	there are more confirmations
`aml_check`	enough confirmations, a risk check is under way
`paid`	paid: you can ship
`expired`, `cancelled`	the deadline passed, or you cancelled a pending payment
`failed`	the transaction failed on the network (don't build everything on this status: a payment may simply expire)

AML is a check of whether the money sent is linked to a bad history. Where it exists, the `aml` object in the `paid` event carries a `score` from 0 to 100 (higher means riskier) and a `risk_level`. There is no separate charge. It isn't available on all networks: without it a payment becomes `paid` right away.

It is a hint, not a verdict and not a legal opinion: what to do with a risky transfer is up to you. If a payment is stuck in `aml_check`, it isn't paid yet; write to support.

Part of an example from the documentation:

```
{ "type": "payment_order", "event": "paid", "order_id": "ord_123",
  "network": "tron", "status": "paid", "tx_hash": "bb08…c320c",
  "received_amount_raw": "30000000",
  "confirmations": 19, "required_confirmations": 19,
  "aml": { "score": 0, "risk_level": "low" } }
```

The fine print

- **Less than required doesn't count.** The payment keeps waiting and expires. The documentation has an `underpaid` status, but don't build a shop on it: notice

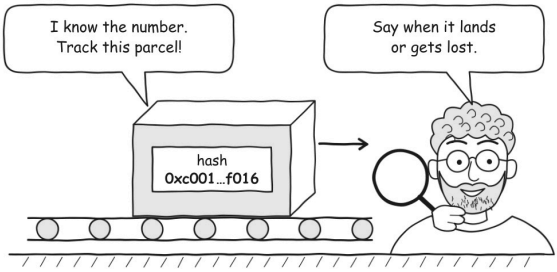
underpayment from the address event.

- **More than required counts.** `received_amount_raw` tells how much actually arrived.
- **A late transfer doesn't pay.** One seen in time calmly gathers confirmations even after the deadline.
- **Give every order its own address.** Then two "for thirty" can't be confused (Chapter 28).
- **An address slot isn't freed by itself** after `paid` or `expired`: delete the watch. After a payment is cancelled, an address added only for it is removed automatically.

Payments are available on paid plans: Starter has 10 open at once and 200 a month. One payment on the TRON or Ethereum main network costs 750 credits, and they are charged only once the money has been seen. Free has a sandbox: 3 open payments on test networks, currently TRON Nile. The risk check there isn't real and is marked `"sample": true`.

Transaction tracking

Sometimes it is the other way round: you already have the hash. Artyom has refunded a customer or paid a supplier and wants to know whether it got there.



"Notifications" → "Transactions" → "Add hash", or `POST /v1/tx-watches` with the body `{"network_slug": "tron", "tx_hash": "HASH"}`. A hash can be added before it appears on the network.

You receive `tx` events: `detected`, then `confirmations` and finally `confirmed` or `failed`. If the hash is never found, `cancelled`. They arrive if incoming or outgoing transfers are turned on for that network.

An open watch takes up a slot and frees it when it ends. Starter has 15 slots; Free doesn't have this service. A transaction is paid for once: on TRON and Ethereum it is 150 credits.

Try it: count confirmations yourself

This is what happens under the hood of any such watch. Let's take the USDT transfer from Chapter 15:

```
URL=https://public-ethereum-mainnet.rpcnode.dev
H=0xc0015ff5178d72a68831b133187c3f9f1fbee2e440d3af311d0b32110e2cf016
curl -s $URL -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionReceipt",
  "params":["'$H'"]}' | grep -oE '"(status|blockNumber)":"0x[0-9a-f]+"' | sort -u
sleep 1
curl -s $URL -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":2,"method":"eth_blockNumber","params":[]}'
```

```
"blockNumber":"0x194646a"
"status":"0x1"
{"jsonrpc":"2.0","id":2,"result":"0x19464c5"}
```

Status `0x1` means success. The block with the transfer counts as the first confirmation, so `echo $((0x19464c5 - 0x194646a + 1))` gives **92**. You will get more.

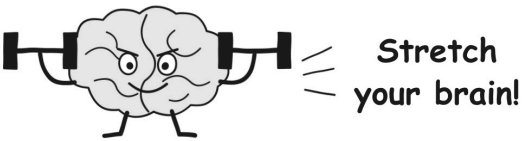
The trap: two calls about the same money

A payment puts the address under watch. So one transfer can bring both a `payment` (address event) and a `payment_order` (payment event). Each is charged at its own price.

> **The rule:** only a `payment_order` with `event = paid` closes an order. And check the order number, `payment_id`, token and amount.

And don't ask about a payment endlessly: the API has its own limit, 100 requests a minute per account. You ask after downtime, to reconcile: `GET /v1/payments/orders/{orderId}`.

Think it over



1. A payment for 30 USDT, and 29.5 arrived. What happens to the payment?
2. The deadline passed, and the transfer was seen a minute later. Where is the money, and what is the status?
3. A `check` event arrived. Can you ship the coffee?

Answers: 1) it isn't paid and will expire on schedule. 2) The money is on your address, the payment stays `expired`*: you sort it out by hand. 3) No, wait for* `paid`*.*

Key points

- A payment links a transfer to an order; a hash watch follows the fate of a transaction.
- You ship on `paid`, not on the first call.
- Underpayments, late arrivals and occupied slots are your concern.

Part five is over. Artyom can get an endpoint, understands a request's path and the plans, and learns about money without polling.

Next: Part 6, Chapter 28. Building the shop: invoice, address, waiting for payment, confirmations.

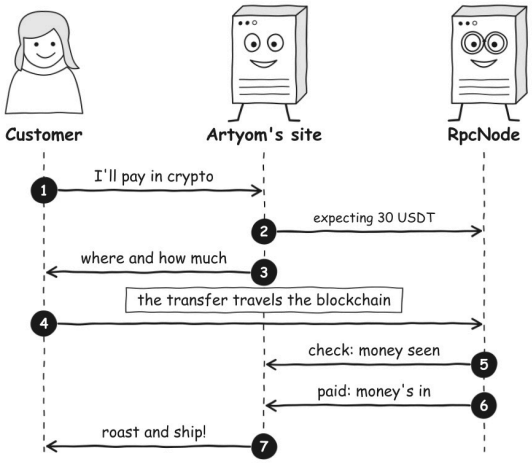
Chapter 28. Building the shop: invoice, address, waiting for payment, confirmations

“Each piece makes sense. But all together?”

Artyom knows about addresses, confirmations, webhooks and payments. And then a customer presses “Pay with cryptocurrency”. What should the website do next, step by step?

Digging in: who takes part

There are four participants. The **customer** with their wallet. The **shop’s website** with its order database. **RpcNode**, which watches the network. And **Artyom’s wallet**, where the keys are. The wallet takes no part in the conversation between the website and the service.

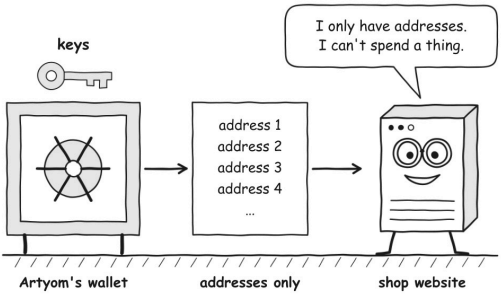


Look at step 4. The money goes neither to the website nor to RpcNode. It lands directly on Artyom’s address, and RpcNode only notices it in a block.

Where to get addresses with no keys on the server

In Chapter 6 we promised: every order gets its own address. And there we also said: a private key has no place on the shop’s server. How do you combine the two?

Addresses can be prepared separately from keys.



There are two ways.

	A list of addresses	An extended public key
What sits on the server	ready addresses, one per order	one key (often starting with `xpub`), from which a program computes addresses itself
Where it works	in any network	Bitcoin, Ethereum, TRON and relatives; not possible this way in Solana and TON
If the server is hacked	the addresses on the list are exposed	all your addresses and everything received on them are exposed
Effort	top up in time	set up once

Money can't be spent with such a key. But the BIP-32 standard warns: this key **together with one** private key from the same branch allows all the private keys of the branch to be computed. So don't export the keys of such a wallet one by one. A beginner finds it easier to start with a list.

Take the list from a wallet whose keys don't sit on a server on the internet. A **hardware wallet** is handy: a separate device that keeps the keys inside itself. Before uploading, check a few addresses against its screen. And in TON, show the customer the address in the `UQ` form (Chapter 19).

Try it: the database won't let you mix them up

Handing out addresses is a job for a database. Here is a toy one: two free addresses and three orders. You need the `sqlite3` program, which many systems already have.

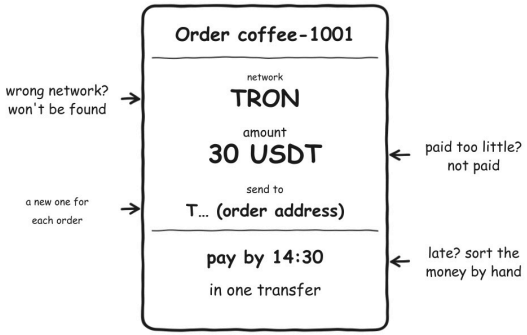
```
sqlite3 :memory: <<'SQL'
CREATE TABLE addresses(address TEXT PRIMARY KEY, order_id TEXT UNIQUE);
INSERT INTO addresses(address) VALUES ('address-1'), ('address-2');

UPDATE addresses SET order_id = 'coffee-1001' WHERE address =
  (SELECT address FROM addresses WHERE order_id IS NULL ORDER BY address LIMIT 1)
  RETURNING order_id, address;
UPDATE addresses SET order_id = 'coffee-1002' WHERE address =
  (SELECT address FROM addresses WHERE order_id IS NULL ORDER BY address LIMIT 1)
  RETURNING order_id, address;
UPDATE addresses SET order_id = 'coffee-1003' WHERE address =
  (SELECT address FROM addresses WHERE order_id IS NULL ORDER BY address LIMIT 1)
  RETURNING order_id, address;
SELECT 'no address left for the third' WHERE changes() = 0;

UPDATE addresses SET order_id = 'coffee-1001' WHERE address = 'address-2';
SQL
```

```
coffee-1001|address-1
coffee-1002|address-2
no address left for the third
Runtime error near line 15: UNIQUE constraint failed: addresses.order_id (19)
```

There is nothing to issue the third invoice with: time to top up the list. And the last command tried to give order 1001 a second address, and the database refused. The words `PRIMARY KEY` and `UNIQUE` stand guard: an address is written down once, and an order has no more than one.

And don't hand out old addresses a second time round. A late transfer for a past order would pay for the new one.

Issuing the invoice



The customer needs five things: network, coin, exact amount, address and deadline. And one request: send it all **in one transfer**, because two halves won't be added together. Warn them: if their wallet or exchange takes its fee out of the amount itself, less will arrive.

Convert prices in rubles or dollars into the coin at the moment of ordering and hold the rate until the deadline. That is why the deadline is made short, for example half an hour.

Now you need the payment from Chapter 27. Below is a **sketch** in Python. The `rpcnode` function in it sends a signed request: write it following Chapter 27 or take the official RpcNode client (available for Python, TypeScript, PHP, Go and Kotlin). The database is the same, with two tables.

```
USDT = 'TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t'    # USDT contract on TRON
# addresses(address, order_id)
# orders(id, payment_id, amount_raw, status — starts as 'waiting')

def take_address(db, order_id):
    with db:                                   # one database transaction
        db.execute('INSERT OR IGNORE INTO orders(id) VALUES (?)',
                   (order_id,))
        mine = db.execute('SELECT address FROM addresses'
                          ' WHERE order_id=?', (order_id,)).fetchone()
        if mine:
            return mine[0]                     # asked again — same one
        free = db.execute(
            'UPDATE addresses SET order_id=? WHERE address='
            ' (SELECT address FROM addresses WHERE order_id IS NULL'
            '  ORDER BY address LIMIT 1) RETURNING address',
            (order_id,)).fetchall()
        if not free:
            raise RuntimeError('out of addresses')
        return free[0][0]
```

```
def checkout(db, rpcnode, order_id, amount):   # amount is a string
    address = take_address(db, order_id)
    reply = rpcnode('POST', '/v1/payments', {
        'order_id': order_id, 'network_slug': 'tron',
        'address': address, 'amount': amount,
        'token_contract': USDT, 'ttl_seconds': 1800})
    if not reply['ok']:
        raise RuntimeError(reply['error_code'])
    payment = reply['payment']
    with db:
        db.execute('UPDATE orders SET payment_id=?, amount_raw=?'
                   ' WHERE id=?', (payment['payment_id'],
                   payment['amount_raw'], order_id))
    return payment        # for the customer: address, amount, expires_at
```

The main thing here is the order: the address is assigned to the order **before** calling the service. If the connection drops, a repeated call takes the same address and order number, and the service returns the same payment. We tested this with a made-up `rpcnode` function; we didn't run the sketch against the real service.

Create the payment when the customer has chosen to pay with cryptocurrency, not for every cart: the plan's monthly limit counts created payments, unpaid ones included.

RpcNode recognizes a payment by address, coin and amount. The comment from Chapter 19 and the `reference` marker from Chapter 18 play no part here: a payment has no such field. XRP Ledger and Stellar from Chapter 20 aren't on the list of networks with payments (Chapter 26).

Waiting for the payment

The order page asks **your server** every few seconds, and the server answers from its own database. Webhooks keep the database up to date. You mustn't ask RpcNode directly from the browser: the key secret would end up with the customer, and besides, the API limit is 100 requests a minute per account.

Status	What the customer sees	What the shop does
`pending`	"Waiting for your transfer until 14:30"	nothing
`check`	"We see your transfer. 3 of 19 confirmations"	doesn't ship
`aml_check`	"Checking your transfer"	waits
`paid`	"Paid, we're packing your order"	ships
`expired`	"Time is up. Don't send money for this invoice"	closes the invoice
`failed`	"The transfer failed"	offers to start over
`cancelled`	"Invoice cancelled"	closes the invoice

Until a transfer gets into a block, the payment is silent. So write "waiting", not "unpaid".

How many confirmations to wait for

This is set under "Settings" → "Confirmations", separately for each network. If you haven't set it, the network's default applies. The required number is visible in the payment, in the `required_confirmations` field.

Setting it lower than the default is risky: fresh blocks are sometimes replaced (Chapter 8). And there is one number per network, not per order. If you want to wait longer for a sack of beans than for a cup, after `paid` wait some more and check yourself, as in Chapter 27.

Free up room

There are two kinds of room. A **payment slot** is freed by itself when the payment becomes `paid`, `expired`, `failed` or `cancelled`. An **address slot** isn't freed by itself (except on cancellation, Chapter 27).

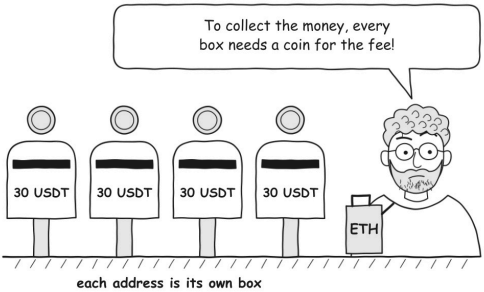
On Starter that is 10 open payments, 15 watched addresses and 200 payments a month. The eleventh open payment is refused with `payment.slot_limit`. And when all the addresses are taken, you get `watch.address_limit`.

Let a closed order sit for a while, in case a transfer is late. Then remove the watch: find its `watch_id` with a `GET /v1/address-watches?network_slug=tron&address=…` request and send `DELETE /v1/address-watches/{watch_id}`. After that there will be no news about the address, although money can still reach it.

Adding and deleting addresses is limited too: per day, per week and per month, and the more slots you have, the bigger the allowance. A new address for every order uses up this allowance. On a sale day you can get `watch.quota` even with free slots. If you expect many orders, find out the limit in advance.

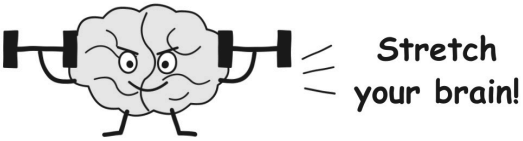
The trap: money on a hundred addresses

An address per order is handy for bookkeeping, but afterwards the money sits in separate places. To send USDT from an address, that same address needs a coin for the fee (Chapters 9 and 17): ETH on Ethereum, TRX or energy on TRON. A hundred orders means a hundred separate transfers. In Bitcoin, many small "pieces" make sending more expensive too (Chapter 14). Work this out before choosing a network for small sales.



And one more thing. Wallets following the BIP-44 standard stop looking after 20 empty addresses in a row. With many abandoned orders, a wallet may not show a payment at a distant address. The money is safe, but check whether it can search further.

Think it over



1. A customer reloaded the payment page. Are a new address and a new payment needed?
2. Artyom already has 10 open payments on Starter. An eleventh customer arrives. What happens?
3. The payment page asks RpcNode directly from the browser. Name two troubles.

Answers: 1) no: the same order number and the same address. 2) A `payment.slot_limit` *refusal: ask them to wait or offer another way to pay. 3) The customer can see the key secret, and the request limit will soon run out.*

Key points

- Keys stay in the wallet, the website has only addresses: one order, one address, no reuse.
- The customer asks your website, and the website learns the news from webhooks.
- Work out slots and the cost of collecting the money in advance.

Next: Chapter 29. When everything goes wrong: errors, retries, stuck transactions, double crediting.

Chapter 29. When everything goes wrong: errors, retries, stuck transactions, double crediting

“The coffee went out twice!”

In the morning Artyom saw that two packages had been sent for one order. And another customer writes: “I paid, but you still say ‘waiting’.” No miracle here. This is what happens when a program was written only for fair weather.

Digging in: three kinds of trouble

Trouble	Example	Remedy
Your request went unanswered	the connection dropped while creating a payment	retry without harm
The news arrived wrong	twice, out of order or never	a lock in the database and reconciliation
The transfer went wrong	stuck, late, too little arrived	rules and manual handling

When to retry an API request

We covered growing pauses in Chapter 23. But you can retry without fear only what is **idempotent**: however many times you do it, the outcome is the same as doing it once. Creating a payment is like that thanks to `order_id`. Retry with the same number and the same data. Sign every retry anew: the signature includes the time.

The official RpcNode clients don’t retry a `POST` by themselves after a dropped connection or a 500-series response. So retrying a payment creation is your job.

What to do with different API responses:

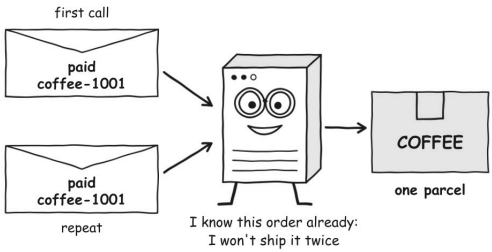
What came back	What to do
nothing; 502, 503, 504	retry with pauses
429 with a `Retry-After` header	wait as long as it says
429 `watch.quota` (Chapter 28)	retrying right away is useless: the allowance has run out
400	an error in the request: `errors` names the fields
401	check the key, the signature and the server clock
403	plan, slot limit or allowed IP list
409 `payment.order_exists`	this number is already taken by a payment with different data: investigate, don't change the number on a guess

It is another matter when a payment has expired and the customer wants to try again. The old `order_id` returns the same expired payment. You need a new number, say `coffee-1001-2`.

One piece of news, twice

Duplicates come from everywhere:

- the webhook was repeated because you didn't answer within 10 seconds;
- both the webhook and the socket are turned on;
- both `payment` and `payment_order` arrived about the same money (Chapter 27);
- the webhook and your scheduled check fired at the same time;
- an employee pressed "ship" by hand.



The delivery number (Chapter 26) filters out repeats of one delivery. But it won't save you from a second piece of news about the same money arriving by another route. The main lock is the order's state: it can pass from "waiting" to "paid" only once.

Try it: one webhook twice

Here is the whole lock. We made up the messages ourselves. Save both pieces into one file `once.py` and run `python3 once.py`.

```
import sqlite3
db = sqlite3.connect(':memory:')
db.executescript("""
CREATE TABLE orders(id TEXT PRIMARY KEY, payment_id TEXT,
                    amount_raw TEXT, status TEXT);
CREATE TABLE to_ship(order_id TEXT PRIMARY KEY);
INSERT INTO orders VALUES('coffee-1001','p1','30000000','waiting');
""")

def on_news(n):
    if n['type'] != 'payment_order' or n['event'] != 'paid':
        return 'not a payment yet'
    with db:               # everything inside is one database transaction
        cur = db.execute(
            "UPDATE orders SET status='paid' WHERE id=? AND"
            " payment_id=? AND amount_raw=? AND status='waiting'",
            (n['order_id'], n['payment_id'], n['amount_raw']))
        if cur.rowcount != 1:
            return 'already counted or not ours'
        db.execute('INSERT INTO to_ship VALUES (?)', (n['order_id'],))
    return 'shipping'
```

Then four messages in a row and the program's answer:

```
paid = {'type': 'payment_order', 'event': 'paid',
        'order_id': 'coffee-1001', 'payment_id': 'p1',
        'amount_raw': '30000000'}
print(1, on_news({**paid, 'event': 'check'}))
print(2, on_news(paid))
print(3, on_news(paid))                      # the same webhook again
print(4, on_news({**paid, 'event': 'check'}))  # a late message
print('parcels:', db.execute('SELECT count(*) FROM to_ship').fetchone()[0])
```

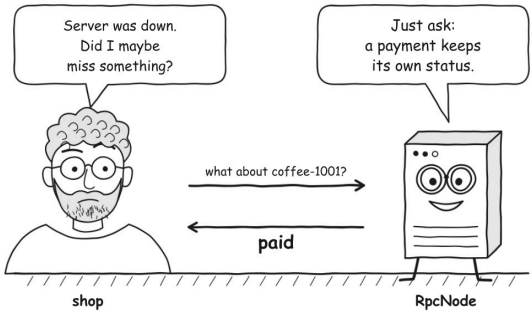
```
1 not a payment yet
2 shipping
3 already counted or not ours
4 not a payment yet
parcels: 1
```

One parcel. Everything rests on a single command: "set `paid` **only if** the order is still waiting and the payment and amount are the right ones". The mark and the shipping task are written together: either both or neither. This is called a **database transaction** (the same word as in blockchain, but a different thing). We also tried eight identical messages at once: there was one shipment.

In the sketch from Chapter 26, delivery numbers lived in memory and were remembered before the work was done. You can't do that with money: the program would crash halfway and mix everything up. First write to the database, then answer 200.

The news never arrived at all

A webhook is dropped after seven attempts, which is about seven hours. A socket doesn't repeat what was missed at all. So some news can get lost. The remedy is **reconciliation**: ask yourself about the orders you still have open.



A `GET /v1/payments/orders/{orderId}` request returns the payment. The names there are a bit different: instead of `event` look at `status`, the network is called `network_slug`, the token contract `token_contract`. Assemble the same kind of "message" from the response and pass it through the same lock.

Reconcile after server downtime, after a socket reconnect and on a schedule. Ten open orders once a minute is 10 requests out of the 100 allowed.

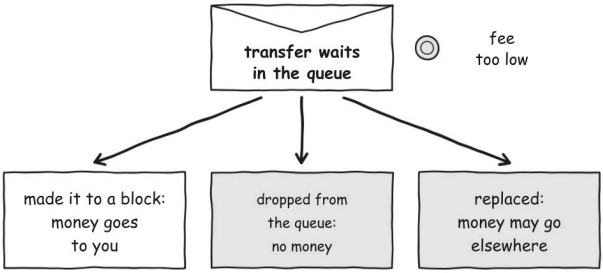
The transfer is stuck

A customer sent a hash: "I sent it an hour ago." The payment is silent. Let's ask a node whether it knows such a transaction. The hash here is made up:

```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionByHash","params":
  ["0x0000000000000000000000000000000000000000000000000000000000000001"]}'
```

```
{"jsonrpc":"2.0","id":1,"result":null}
```

`null` means "this node doesn't know it". The reasons vary: a typo, a different network, the transfer hasn't arrived yet or has already been thrown out. But if the transaction is found and its `blockNumber` is `null`, it is waiting in the queue.



until a transfer is in a block, you have no money

Network	What happens to one that waits long
Ethereum and similar	holds up all the sender's following transfers (Chapter 15). It can be replaced by a transfer with the same `nonce` and a bigger fee: the geth node software by default requires at least 10% more
Bitcoin	the sender can replace it with a transfer that has a bigger fee. The Bitcoin Core software by default drops from the queue whatever has sat there for two weeks
TRON	a transaction has an expiry time, one minute by default. If it didn't make a block in time, it never will
Solana	an ordinary transaction is valid while the block it refers to is fresh: 150 blocks, which is a few dozen seconds

As a rule, a shop can't speed up someone else's transfer: the sender's key is needed. Wait for the block and don't ship. If the payment expires in the meantime and the transfer goes through after all, the money lands on your address, but the payment stays `expired`.

Is it **your own** send that is stuck, a refund say? The same signed transfer can be sent once more: the hash won't change. But a new transfer is a second lot of money, unless it replaces the first. Whether an address has a queue in Ethereum can be seen like this:

```
URL=https://public-ethereum-mainnet.rpcnode.dev
A=0x8f8d1206d1bce12ff892731f8a140ed18b93884e
for tag in latest pending; do
  curl -s $URL -H 'Content-Type: application/json' -d '{"jsonrpc":"2.0",
    "id":1,"method":"eth_getTransactionCount","params":["'$A'","'$tag'"]}'
  echo; sleep 2
done
```

```
{"jsonrpc":"2.0","id":1,"result":"0x67641"}
{"jsonrpc":"2.0","id":1,"result":"0x67641"}
```

The first number is how many of the address's transfers are already in blocks, the second includes the pending ones. They are equal: this node sees no queue. This is the sender's address from Chapter 15.

And be careful with hash watching (Chapter 27): a `cancelled` answer means the service didn't find the transaction, not that it will never appear.

The money arrived, but not right

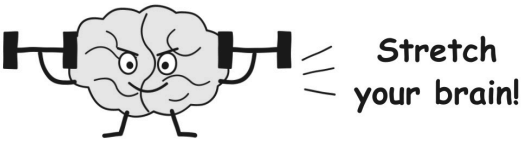
What happened	What happens to the payment
too little arrived	not paid, expires on schedule; you already have the money
in two parts	the parts aren't added together: the same thing
too much arrived	`paid` ; the actual amount is in `received_amount_raw`
sent again after `paid`	the payment is no longer waiting: you can learn of it only from the address event
after the deadline	`expired`
wrong token or wrong network	not counted (Chapter 16)
the transaction failed	there is no money, don't ship: the payment becomes `failed` or simply expires

Decide in advance what you do in each case and tell the customer on the payment page.

The trap: "I'll refund to where it came from"

A customer overpaid, and Artyom wants to send the extra to the sender's address. But it could have been an exchange's shared address: the exchange would get the money, not the customer. Ask the customer for the refund address and network. Send it from your wallet by hand, not from the website.

Think it over



1. The response to creating a payment didn't arrive. Can you retry? With which `order_id`?
2. The same `paid` arrived by webhook and by socket. How many parcels will the program from the example send?
3. Instead of 30 USDT, two transfers of 15 were sent. What happens to a payment for 30?

Answers: 1) yes, with the same one and the same data. 2) One. 3) It isn't paid: transfers aren't added together.

Key points

- Retry only what is safe to retry.
- A lock in the database guards shipping, and reconciliation finds lost news.
- Until a transfer is in a block there is no money; rules for odd payments are needed in advance.

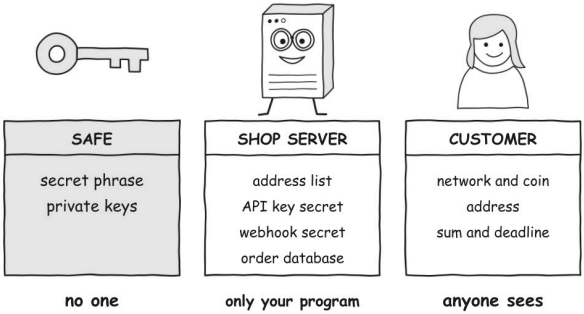
Next: Chapter 30. Security and a pre-launch checklist.

Chapter 30. Security and a pre-launch checklist

“We open tomorrow. What have I forgotten?”

Everything works on the test network. Artyom is already writing to customers: “We accept crypto!” But he sleeps badly. What can go wrong when the money becomes real?

Digging in: what is kept where



An API key secret doesn’t let anyone spend your coins: RpcNode has no private keys. But it can still cause trouble. Set “Allowed IPs” on the key: then a stolen secret won’t work from someone else’s computer.

Two-factor authentication is when you need a code from an app besides the password. In the cabinet it is under “Settings”. Turn on the same for your email: access to everything else is often recovered through email.

What happens if any of this leaks:

What leaked	What a stranger can do	What to do
secret phrase, private key	take the money for good	urgently move what is left to a new wallet
API key secret	manage payments, watches, endpoints and the webhook, read its secret	delete the key, issue a new one, change the webhook secret, check the settings
webhook secret	forge a `paid` message	“Generate new secret”
endpoint address	spend your credits	create a new one, delete the old (Chapter 22)
cabinet sign-in	almost everything in this table, except your coins	change the password, turn on two-factor authentication

Secrets don't live in code

A secret in code travels off into a shared repository, into an archive for a contractor, into a screenshot. Keep secrets in server settings (environment variables), and close the file that holds them to outsiders.

Try it: search your own place

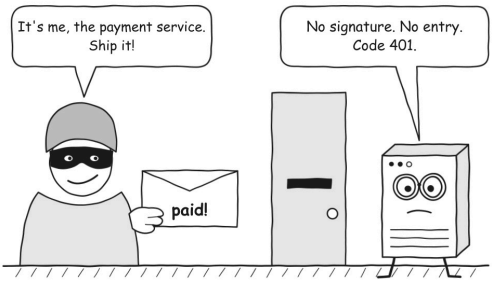
An API key secret starts with `sk_live_`. Let's make a practice folder with one mistake and find it:

```
mkdir demo
echo "const SECRET = 'sk_live_example'" > demo/pay.js
echo "const SECRET = process.env.RPCNODE_API_SECRET" > demo/ok.js
grep -rnE 'sk_live_|PRIVATE KEY|xprv' demo
```

```
demo/pay.js:1:const SECRET = 'sk_live_example'
```

The search found the first file and left the second alone. Replace `demo` with your project's folder. `xprv` is the start of an extended **private** key. If something is found, erasing the line isn't enough: it may remain in the change history. Consider the secret leaked and issue a new one. A secret phrase can't be caught this way, though, as it is ordinary words: just don't keep it on a computer.

Don't trust what comes in



Let's test our own door. Save the receiver sketch from Chapter 26 into `receiver.mjs` and start it:

```
RPCNODE_WEBHOOK_SECRET=demo-secret node receiver.mjs
```

In another terminal window, knock in three outsiders' ways:

```
URL=http://localhost:3000
BODY='{"type":"payment_order","event":"paid"}'
# 1) no signature
curl -s -o /dev/null -w '%{http_code}\n' $URL -d "$BODY"
# 2) a guessed signature
curl -s -o /dev/null -w '%{http_code}\n' $URL -d "$BODY" \
  -H "X-RpcNode-Signature: t=$(date +%s),v1=$(printf 'a%.0s' {1..64})"
# 3) a genuine signature, but an hour old
T=$(( $(date +%s) - 3600 ))
SIG=$(printf '%s.%s' $T "$BODY" |
  openssl dgst -sha256 -hmac 'demo-secret' | awk '{print $NF}')
curl -s -o /dev/null -w '%{http_code}\n' $URL -d "$BODY" \
  -H "X-RpcNode-Signature: t=$T,v1=$SIG"
```

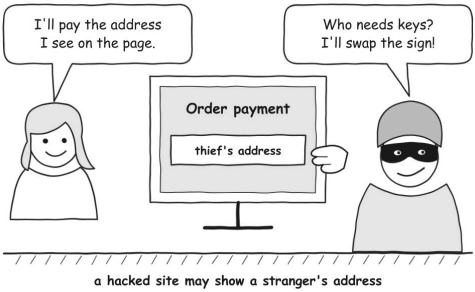
```
401
401
401
```

Three times "no entry". If a 200 came back anywhere, the door is open. Replace `- 3600` with `- 0`, and the third knock gets a 200: the signature is fresh and correct.

For expensive orders, hang a second lock. On receiving `paid`, ask for the payment yourself (`GET /v1/payments/orders/{orderId}`) and check `payment_id`, address, token, amount and status. Then even a stolen webhook secret won't get anyone free coffee.

Also accept webhooks over `https` and limit the size of an incoming request: the sketch from Chapter 26 doesn't do that. In addition, you can let in only the addresses from the "Secret and IPs" section. That is an addition to the signature, not a replacement for it.

The switched sign



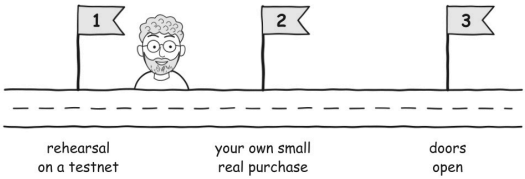
A thief doesn't have to get hold of keys. It is enough to hack the website and show the customer his own address. What helps:

- the address list can't be quietly replaced through the website's admin panel;
- from time to time, check the issued addresses against your wallet, from another computer;

- now and then buy from yourself and see where you are asked to pay;
- update the software on the server, and sign in to the admin panel with two-factor authentication.

Thieves write to you personally too: “support”, “return a stuck payment”, a tiny transfer from a look-alike address (Chapter 6). Real support doesn’t ask for your secret phrase.

The rehearsal



First, everything on a test network. Payments and events among the test ones work on TRON Nile (Chapter 26). Free has a sandbox: by default 3 open payments and 3 addresses, and the risk check there isn’t real. Addresses aren’t freed by themselves in the sandbox either.

What to act out	What should happen
pay the exact amount	one parcel
pay less	nothing shipped, the payment expired
send yourself one message twice	one parcel
turn your server off, pay, turn it on	the payment is found: by a webhook retry or by reconciliation
knock without a signature	401

Then one small real purchase from yourself on the network where you will trade. At the same time check that your wallet sees the money and that you can send it onward: whether there is enough coin for the fee.

Law and taxes

In many countries accepting cryptocurrency is restricted or taxed. Before launching, check your country’s laws. This isn’t legal advice. While you are at it, ask your accountant which records of payments they need, and keep them from day one.

The trap: "checked once and forgot"

The shop is working, and everything around it keeps changing. The plan term runs out and the account moves to Free, where there are no main-network payments (Chapter 25). Events spend credits too. The address list is running low. Set yourself reminders.

Pre-launch checklist

Keys and access

- ☐ the secret phrase is written down and isn't on any server or computer;
- ☐ the website has only addresses, and several of them have been checked against the wallet;
- ☐ there are no secrets in the code, the project search comes up empty;
- ☐ allowed IPs are set on the API key;
- ☐ two-factor authentication is on for the cabinet, email and the website's admin panel.

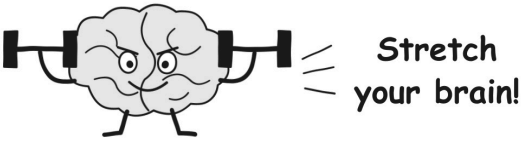
Accepting money

- ☐ one order, one address, no reuse;
- ☐ the invoice states network, coin, amount, address, deadline and "in one transfer";
- ☐ the token contract is taken from `GET /v1/network-tokens` and checked against the issuer's website;
- ☐ the number of confirmations is chosen to suit your amounts;
- ☐ the webhook signature is verified, and a lock in the database guards shipping;
- ☐ open orders are reconciled on a schedule.

Matters around it

- ☐ there are rules for underpayment, overpayment, late payment and refunds;
- ☐ slots, credits, plan term and the stock of addresses are being watched;
- ☐ it is clear how the fees for collecting the money will be paid;
- ☐ a copy of the order database is made, and it has been restored from at least once;
- ☐ the rehearsal is passed, your own real purchase is made;
- ☐ your country's laws and taxes are checked.

Think it over



1. An API key secret has leaked. Can anyone take the coins from your wallet? What must you still do immediately?
2. Your receiver answered 200 to a message without a signature. What does that threaten?
3. All checks passed on TRON Nile. Can you start accepting payments on Ethereum without a new check?

Answers: 1) no, there are no wallet keys there; but delete the API key, issue a new one, change the webhook secret and check the settings. 2) Anyone can "pay" for an order without money. 3) No: a different network, different addresses, fees and confirmations.

Key points

- Keep the keys to the money away from the website, and secrets away from the code.
- Don't take an incoming message, a screenshot or an email at its word.
- First a rehearsal, then a small real purchase, then the opening.

Part six is over. Artyom has put the shop together, knows what to do when it breaks and how to guard it.

Next: Chapter 31. DeFi: finance without a middleman.

Chapter 31. DeFi: finance without a middleman

"Can coins do more than just sit there?"

Artyom has accumulated part of his earnings in USDC. An acquaintance advises: "Put it into DeFi, the money makes money by itself there." What is this place with no counter and no teller?

> This is **not financial advice** but an explanation. Protocol names are only examples. In your country such services may be restricted or taxed: check the laws.

Digging in: a bank and a program

DeFi, "decentralized finance", means exchange, lending and other money services carried out by smart contracts (Chapter 9). A set of contracts with common rules is called a **protocol**.

	Bank	DeFi
Where the money is	at the bank	at an address or in a contract
Who carries things out	staff and their systems	a program on the network
When it works	on its own schedule	around the clock
Where the rules are	in a contract on paper	in the code
You made a mistake	you can turn to the bank	usually nobody can undo it
You lost access	your passport will help	no way back without the key

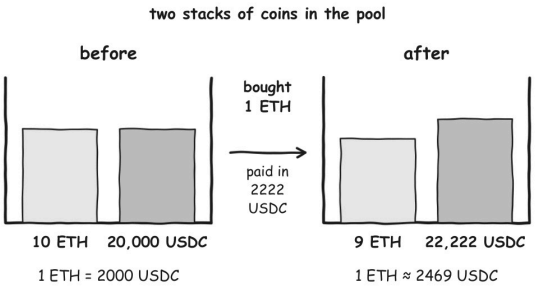
"Without a middleman" doesn't mean "without trust". You trust the code, those who can change it, the price suppliers and the coin's issuer.

Exchange without an exchange: the liquidity pool

On an ordinary exchange, a buyer is matched with a seller. In DeFi people often set up a **liquidity pool** — a contract holding two coins at once. You put one into the pool and take the other out. No need to wait for a second person.

How much you get is decided by a formula. For simple pools it is: $x \times y = k$, where x and y are the reserves of the two coins. A swap must not make this product smaller.

The price here is the ratio of the reserves. Say the pool holds 10 ETH and 20,000 USDC. That means 1 ETH costs about 2000 USDC, and k = 200,000. Artyom wants to take 1 ETH (we ignore the fee for now). 9 ETH will remain, so the USDC must become 200,000 ÷ 9 ≈ 22,222.



He will pay 2222 USDC, not 2000: **his own purchase raised the price**. This is called **price impact**. In a pool a hundred times bigger (1000 ETH and 2,000,000 USDC), the same 1 ETH would cost about 2002 USDC.

If ETH has become dearer in the pool than elsewhere, someone will bring ETH there and take USDC until the prices come closer. This is **arbitrage**.

The coins in a pool are put there by **liquidity providers**, who receive a share of the fees. But when prices diverge, the pool gives away the coin that has risen and accumulates the other. Ignoring fees, if the price doubles, a deposit is worth about 5.7% less than if the coins had simply been held. This is called **impermanent loss**: withdraw at that moment and it becomes quite permanent.

Try it: ask a real pool

Let's take the USDC and WETH pool of the Uniswap v2 protocol on Ethereum. **WETH** is ETH in token form, one to one.

A pool address must not be taken from a random search. We took the address of the pool "factory" from the Uniswap documentation, the USDC address from its issuer Circle, and the WETH address from ethereum.org. Then we asked the factory the `getPair` question: "where is the pool for this pair?" The answer: `0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc`.

Now the `getReserves` question — "what are the reserves?". This is the `eth_call` from Chapter 9: it only reads.

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[
  {"to":"0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc",
   "data":"0x0902f1ac"},"latest"]}'
```

On the night of November 27, 2026, the node answered:

```
{"jsonrpc":"2.0","id":1,"result":"0x000000000000000000000000000000000000000000000000000079ecc8266bc00000000000000
0000000000000000000000000000000010b762ae141ddef4f50000000000000000000000000000000000000000000000000000000006b08b4a
f"}
```

After `0x` come three numbers of 64 characters each:

Number (without leading zeros)	What it is
`79ecc8266bc`	the USDC reserve, 6 decimal places: 8,378,617 USDC
`10b762ae141ddef4f50`	the WETH reserve, 18 places: 4933.80 WETH
`6b08b4af`	when the reserves last changed, in seconds since 1970

Which coin comes first is told by the `token0` and `token1` questions: in this pool the first is USDC. All that remains is to divide:

```
python3 -c 'u = 0x79ecc8266bc / 10**6; w = 0x10b762ae141ddef4f50 / 10**18; print(u, w, u / w)'
```

It prints `8378617.325244 4933.79553339896 1698.2092728662094`.

One ETH cost about **1698 USDC** in this pool. Your numbers will differ.

And how much would you get for a large batch? The swap fee here is 0.3%: of that 0.25% goes to the providers and 0.05% to the protocol. With the same reserves it comes out like this:

We give	We get in USDC	Per 1 ETH	Worse than 1698.21 by
1 ETH	1692.77	1692.77	0.3%
100 ETH	165,957.86	1659.58	2.3%
1000 ETH	1,408,492.58	1408.49	17%

The calculation matched to the last digit the answers of Uniswap's helper contract (the router) to the `getAmountsOut` *question.*

Important: dividing the reserves works for v2-type pools. In Uniswap v3 and v4, deposits are spread across price ranges, and the price there is computed differently.

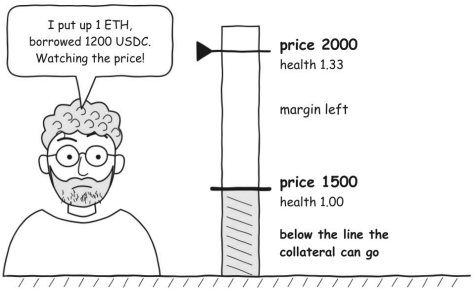
Collateralized loans and liquidation

In a lending protocol (Aave is an example) you put up **collateral** and borrow less than it is worth. Why? Say you need dollars but don't want to sell your ETH.

A loan has a "health" (health factor):

```
health = collateral value × liquidation threshold ÷ debt
```

A made-up example: collateral of 1 ETH at 2000 USDC, threshold 80%, debt 1200 USDC. Health: 2000 × 0.8 ÷ 1200 ≈ 1.33. If the price falls to 1500, it becomes exactly 1.



If health is below 1, any participant can repay part of the debt, sometimes all of it, and take collateral with a bonus in return. This is **liquidation**. There will be no phone call asking you to "top up": the program does it all, even in the middle of the night. The debt also grows by itself, through interest.

Stablecoins: what holds up "one dollar"

Chapter 9: a stablecoin only **tries** to be worth about a dollar. They hold on to it in different ways. **Reserves at the issuer**: dollars and other assets (USDT, USDC). **Crypto collateral with a margin** in contracts (DAI, USDS; other stablecoins can be among the collateral too). **An algorithm** that changes the number of coins, with no collateral.

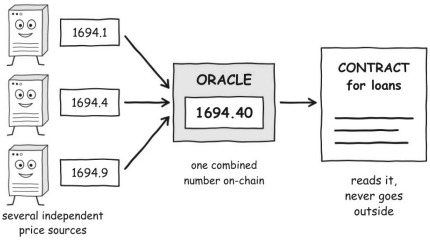
No method is a guarantee. If such a coin gets cheaper in a pool, it is exactly the one the providers will accumulate.

Staking in brief

The validator from Chapter 8 needs 32 ETH. With a smaller sum people take part through a **shared staking pool** (this is not a swap pool). In return you are often given a receipt token. Risks: penalties for validators, a bug in the contract, trust in the operator. Nobody promises how big the income will be.

Oracles: how a contract learns a price

A lending contract needs the price of ETH. But it doesn't go out to the internet itself: every node must get one and the same result (Chapter 15). The price is written into the network by an **oracle**.



An example is the Chainlink oracle. The address of its "ETH / USD" feed is taken from Chainlink's own directory. The `latestRoundData` question asks for the latest record:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[
  {"to":"0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419",
   "data":"0xfeaf968c"},"latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"result":"0x0000000000000000000000000000000000000000000000070000000000008e4f00000000000000
00000000000000000000000000000000000000002773693800000000000000000000000000000000000000000000000000000000006b08aefa
000000000000000000000000000000000000000000000000000000000006b08af0f0000000000000000000000000000000000000000000000700
00000000008e4f"}
```

The second number of the five, `2773693800`, is the price with 8 decimal places: **1694.40 dollars**. The pool showed 1698.21 USDC, a difference of about 0.2%. The fourth number is the time of the record: 25 minutes before our question. This feed is updated when the price moves 0.5% away from the recorded one, and on a schedule, about once an hour.

Why not take the price straight from a pool? A large trade can shift it for a moment. But an oracle can also be wrong or late, and liquidation will go by its number.

How a DeFi application reads the network

A swap or lending website is a storefront. It gets its data from a node:

- `eth_call` — what is there now: pool reserves, balances, the oracle price, loan health.
- **Events** (Chapter 15) — what has changed. After every change a v2 pool announces a `Sync` event with the new reserves. They are found with `eth_getLogs` or listened for over WebSocket (Chapter 11).

- **Sending** — a swap changes the notebook. The wallet signs it, a node passes it on (Chapter 7), and it costs gas.

Here is a question about the `Sync` events of our pool in the latest block:

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getLogs","params":[{
  "address":"0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc",
  "topics":["0x1c411e9a96e071241c2f21f7726b17ae89e3cab4c78be50e062b03a9fffbbad1"],
  "fromBlock":"latest","toBlock":"latest"}]}'
```

```
{"jsonrpc":"2.0","id":1,"result":[]}
```

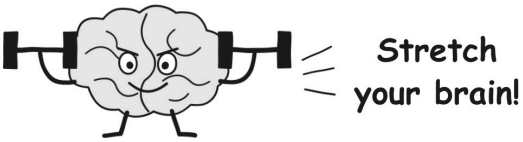
An empty list is an answer too: the pool didn't change in this block. The third number from `getReserves` confirms it: the last change was about 10 minutes before our question. Past changes, for a price chart say, are already a matter for an archive or your own database (Chapter 10).

The trap: what can go wrong

Danger	What happens	What lowers the risk
A bug in the code	money is stolen or gets stuck	a code audit, though it is no guarantee
A fake token	a familiar name, a foreign address	check the address (Chapter 9)
Unlimited allowance (`approve`)	tokens are taken later	allow little, reset what is left over to zero
Slippage	you got less than you expected	set a minimum amount
Liquidation	the collateral is gone, with a bonus on top	a large margin and supervision
A look-alike site	a copy of the site asks for a signature	check the address; don't enter your secret phrase
"Guaranteed income"	they promise and vanish	don't take part

Slippage is the difference between the expected and the actual price. While a swap waits in the queue, other people's trades move the price, sometimes on purpose. A minimum amount protects you: the swap won't go through at anything worse, although gas will be spent.

Think it over



1. A pool holds 100 ETH and 200,000 USDC. Someone took 20 ETH. How much USDC did they put in (without the fee)?
2. Collateral of 2 ETH at 2000 USDC, threshold 80%, debt 2400 USDC. What is the health? At what ETH price will it equal 1?
3. A website promises 3% a day with no risk and asks you to allow it to spend USDT without limit. What do you do?

Answers: 1) 80 ETH will remain, so the USDC becomes 250,000: they put in 50,000, or 2500 per ETH. 2) About 1.33; at a price of 1500. 3) Sign nothing and close the site.

Key points

- In DeFi contracts carry out the services, while the risk and responsibility largely fall on you.
- In a v2-type pool the price follows from two reserves, and a large trade shifts it.
- An application reads the network with the same requests you do: `eth_call` and events.

Next: the conclusion. From a customer's email to a shop of your own.

Checked on November 27, 2026: developers.uniswap.org, developers.circle.com/stablecoins/usdc-contract-addresses, ethereum.org (wrapped-eth, defi, stablecoins, staking, developers/docs/oracles), aave.com/help/borrowing/liquidations, docs.chain.link.

Conclusion. From a customer's email to a shop of your own

It all started with one line in an email: "Can I pay in cryptocurrency?" Back then Artyom didn't know what a node was and checked his wallet by hand. Look at the road you have travelled with him.

What Artyom can do now

Part	What we learned
1. Why you need a provider	To count the cost of your own servers and choose between running a node yourself and getting one from a provider
2. Blockchain from scratch	To understand blocks, keys, signatures, fees, confirmations and tokens
3. Nodes and RPC	To ask a node questions, tell kinds of nodes apart and check whether one has fallen behind
4. How networks work	To see how networks differ and pick one that suits your customers
5. RpcNode	To get an endpoint, work out the price of calls and learn about payments without polling
6. Practice	To assemble payment acceptance, survive failures and protect a shop
7. What's next: DeFi	To read a pool straight from the network and weigh the risks honestly

Seven rules to remember

1. Show the address to everyone; the private key and the secret phrase to no one.
2. Check everything at once: network, token contract address, recipient and amount.
3. The money has arrived when the transaction succeeded and has gathered confirmations, not when a screenshot comes in.
4. Code 200 isn't success yet: read what is inside the response.
5. One payment, one shipment, however many times the news of it arrives.
6. Guard endpoints and secrets like passwords.
7. Don't take anyone's word: ask the network and check the answer.

Where to start tomorrow

1. Repeat the request from Chapter 3 to a public node.
2. Set up your own endpoint and a test network (Chapter 22).

3. Act out a payment to yourself with toy money (Chapters 28 and 30).
4. Go through the checklist in Chapter 30.
5. Check your country's laws and taxes.

If something is out of date

The book describes the networks and the service as they were on November 27, 2026. Networks upgrade, prices and plans change. If your node's answer doesn't match the book, believe the node and the current documentation. See RpcNode's plans and features at rpcnode.dev.

Goodbye

Artyom is still thinking about whether to put part of his earnings into DeFi. But he knows which questions to ask. That, perhaps, is the main thing he has taken from the whole story.

Some of the numbers in this book will go out of date, but the habit of checking won't. Good luck to you and your business!



Appendix A. Glossary

In brackets is the chapter where the word is explained in more detail.

Address — the "box number" that money is sent to (6).

AML — a check of whether money is linked to a bad history (27).

API — a service window through which one program addresses another (11).

Archive node — remembers the state at any block in the past (10).

Block — a "page of the notebook": transactions, a number, a time and the seal of the previous block (5).

Blockchain — a shared list of records kept jointly by common rules (4).

Bridge — a way to carry value from network to network (16).

Chain ID — the number of an EVM network (16).

Confirmations — blocks that have piled on top of your transaction, counting its own (5).

Consensus — the rules by which participants choose one chain (8).

Credits — the units in which the service counts request usage (3, 25).

DeFi — money services carried out by smart contracts (31).

Double spend — an attempt to pay with one coin twice (4).

Endpoint — the address at which a program asks a node questions (3).

Event (log) — a contract's announcement of what has happened (15).

EVM — the Ethereum Virtual Machine, which runs contracts (15).

Fee — the network's charge for a place in a block (7).

Finality — the moment after which a block won't be replaced (8).

Gas — the measure of work for which a fee is charged in Ethereum and its relatives (15).

gRPC — a way for programs to communicate in compact messages following a strict schema (11).

Hash, "seal" — a short fingerprint of any data (5).

Idempotency — however many times you repeat, the outcome is the same as doing it once (29).

Indexer — a program that files blockchain records into a "card catalog" (10).

L2, "second floor" — a network that carries out transfers on its own and sends records of them to Ethereum (16).

Liquidation — the sale of collateral when it no longer secures the loan (31).

Liquidity pool — a contract with two coins, where they are swapped by a formula (31).

Load balancing — handing out requests among several nodes (23).

Mempool — the queue of transfers waiting for a block (7).

Mining — solving a puzzle by trial and error for the right to add a block (8).

Node — a computer with the network's software: it stores, checks and answers (1).

Nonce — the sequence number of an account's transaction (15).

Oracle — whatever writes outside information, such as a price, into the network (31).

Private key — what controls the money; shown to no one (6).

Provider — a service that runs nodes and gives access to them (1, 3).

Pruned node — checked everything and then deleted the old blocks (10).

Reorganization — the replacement of some fresh blocks by another branch (8).

Request limit (RPS) — how many times a second you may ask (3, 25).

REST — a way of asking where every question has its own path in the address (11).

RPC, JSON-RPC — a way to ask a program on another computer to do something (11).

Secret phrase — 12 or 24 words from which keys are restored (6).

Signature — proof that the owner of the key authorized a transfer (6).

Smart contract — a program written into the blockchain (9).

Stablecoin — a token that tries to stay near a dollar (9).

Staking — taking part in securing a network with a deposit of coins (8).

Syncing — how a node catches up with the network (12).

Test network — a training ground with coins that have no price (22).

Token — records kept by a smart contract (9).

UTXO — an unspent "piece of money" in Bitcoin and its relatives (14).

Validator — a participant who puts up a deposit and takes part in creating blocks (8).

Wallet — a program or device that stores keys (6).

Webhook — a request the service itself sends to your address when something happens (26).

WebSocket — a connection that stays open to receive news (11).

Wei — the tiniest fraction of an ETH (1).

Appendix B. The book's networks on one page

Information from Chapters 6, 8 and 14–20, as of November 27, 2026. It changes as networks upgrade.

Bitcoin and its relatives (Chapter 14)

Network	Coin	Block, roughly	Address starts with
Bitcoin	BTC	10 min	`1`, `3`, `bc1`
Litecoin	LTC	2.5 min	`L`, `M`, `ltc1`
Dogecoin	DOGE	1 min	`D`
Bitcoin Cash	BCH	10 min	`bitcoincash:q…`
Dash	DASH	2.5 min	`X`
Zcash	ZEC	75 s	`t1`, `zs`, `u1`

Ethereum and EVM networks (Chapters 15 and 16)

All their addresses start with `0x`.

Network	Chain ID	Fee paid in	Block, roughly
Ethereum	1	ETH	12 s
Base	8453	ETH	0.2 s
Arbitrum One	42161	ETH	0.3 s
BNB Smart Chain	56	BNB	0.45 s
Mantle	5000	MNT	2 s
Monad	143	MON	0.3 s

The rest (Chapters 17–20)

Network	Coin	Address starts with	Final
TRON	TRX	`T`	about a minute
Solana	SOL	no special prefix	fractions of a second
TON	Gram	`EQ`, `UQ`	about a second
Sui	SUI	`0x`	under a second
Aptos	APT	`0x`	under a second
XRP Ledger	XRP	`r`	3–5 seconds
Stellar	XLM	`G`	about 5 seconds
Cardano	ADA	`addr1`	gradually
Polkadot Hub	DOT	`1`	by a mark
Cosmos Hub	ATOM	`cosmos1`	quickly

Appendix C. Request cheat sheet

All the examples work on RpcNode's public nodes: no sign-up and no key, with a rate limit. For your own project, replace the address with a personal endpoint.

JSON-RPC: one command for every case

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"METHOD","params":[PARAMETERS]}'
```

Put in the method and parameters from the tables. For another network, change `ethereum` in the address.

Ethereum and EVM networks

To find out	Method	Parameters
latest block number	`eth_blockNumber`	empty
network number	`eth_chainId`	empty
coin balance	`eth_getBalance`	`"0xADDRESS","latest"`
transaction receipt	`eth_getTransactionReceipt`	`"0xHASH"`
gas price	`eth_gasPrice`	empty
ask a contract	`eth_call`	`{"to":"0x…","data":"0x…"},"latest"`

Bitcoin and Solana

Address	Method	To find out
`public-bitcoin-mainnet`	`getblockcount`	chain height
`public-bitcoin-mainnet`	`getblockchaininfo`	node state
`public-solana-mainnet`	`getSlot`	slot number
`public-solana-mainnet`	`getBalance`	balance in lamports; the parameter is an address

TRON and TON: the question is in the address

```
curl https://public-tron-mainnet.rpcnode.dev/wallet/getnowblock
```

```
curl https://public-ton-mainnet.rpcnode.dev/api/v2/getMasterchainInfo
```

Error codes

Code	What happened
`-32700`	the request can't be parsed
`-32601`	method not found or unavailable
`-32602`	invalid parameters
`-32005`, HTTP 429	too often
`-32006`, HTTP 402	not enough credits

Full documentation and current plans

https://rpcnode.dev

We are on Telegram

@rpcnode_dev

Author of the book: rpcnode.dev

The end.

In plain words

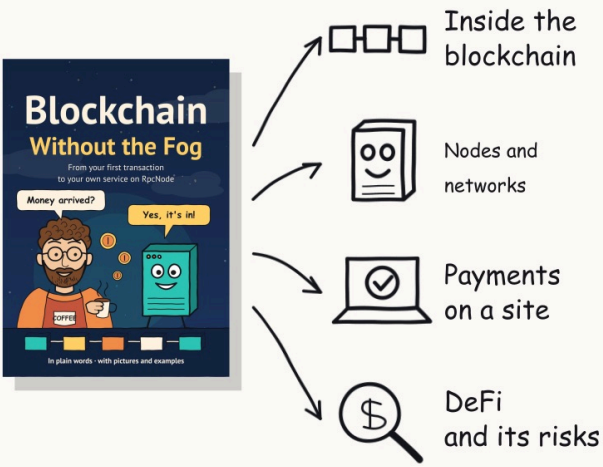
Blockchain Without the Fog

What will this book teach?

A customer asks: “Can I pay in crypto?” What to reply, how to tell the money arrived, and not lose it en route?

Inside: blocks, keys, fees and tokens, nodes and how to query them, Bitcoin, Ethereum, TRON, Solana, TON and more. Then practice: an RpcNode endpoint, payments, webhooks, shop security. Last, an honest talk about DeFi.

> “My website can total up a cart and send emails. How can it know what's happening on the blockchain?”
>
> **— Artyom,** the book's hero



What's special about this book?

No prior knowledge is needed. Chapters bring the hero's questions, simple answers, art and tasks. You can run the requests yourself and see a real response.

rpcnode.dev
docs.rpcnode.dev
Telegram: @rpcnode_dev
Author: rpcnode.dev