

ETHEREUM BASICS

In this chapter we will start exploring Ethereum, learning how to use wallets, how to create transactions, and also how to run a basic smart contract.

ETHER CURRENCY UNITS

Ethereum's currency unit is called *ether*, identified also as "ETH" or with the symbols Ξ (from the Greek letter "Xi" that looks like a stylized capital E) or, less often, \blacklozenge : for example, 1 ether, or 1 ETH, or $\Xi 1$, or $\blacklozenge 1$.

Tip Use Unicode character U+039E for Ξ and U+2666 for \blacklozenge .

Ether is subdivided into smaller units, down to the smallest unit possible, which is named *wei*. One ether is 1 quintillion wei ($1 * 10^{18}$ or 1,000,000,000,000,000,000). You may hear people refer to the currency "Ethereum" too, but this is a common beginner's mistake. Ethereum is the system, ether is the currency.

The value of ether is always represented internally in Ethereum as an unsigned integer value denominated in wei. When you transact 1 ether, the transaction encodes 1000000000000000000 wei as the value.

Ether's various denominations have both a *scientific name* using the International System of Units (*SI*) and a colloquial name that pays homage to many of the great minds of computing and cryptography.

[Ether denominations and unit names](#) shows the various units, their colloquial (common) names, and their SI names. In keeping with the internal representation of value, the table shows all denominations in wei (first row), with ether shown as 10^{18} wei in the 7th row.

Table 1. Ether denominations and unit names

Value (in wei)	Exponent	Common name	SI name
1	1	wei	Wei
1,000	10^3	Babbage	Kilowei or femtoether
1,000,000	10^6	Lovelace	Megawei or picoether
1,000,000,000	10^9	Shannon	Gigawei or nanoether
1,000,000,000,000	10^{12}	Szabo	Microether or micro
1,000,000,000,000,000	10^{15}	Finney	Milliether or milli
1,000,000,000,000,000,000	10^{18}	<i>Ether</i>	<i>Ether</i>
1,000,000,000,000,000,000,000	10^{21}	Grand	Kiloether
1,000,000,000,000,000,000,000,000	10^{24}		Megaether

CHOOSING AN ETHEREUM WALLET

The term "wallet" has come to mean many things, although they are all related and on a day-to-day basis boil down to pretty much the same thing. We will use the term "wallet" to mean a software application that helps you manage your Ethereum account. In short, an Ethereum wallet is your gateway to the Ethereum system. It holds your keys and can create and broadcast transactions on your behalf. Choosing an Ethereum wallet can be difficult because there are many different options with different features and designs. Some are more suitable for beginners and some are more suitable for experts. The Ethereum platform itself is still being improved, and the "best" wallets are often the ones that adapt to the changes that come with the platform upgrades.

But don't worry! If you choose a wallet and don't like how it works—or if you like it at first but later want to try something else—you can change wallets quite easily. All you have to do is make a transaction that sends your funds from the old wallet to the new wallet, or export your private keys and import them into the new one.

We've selected three different types of wallets to use as examples throughout the book: a mobile wallet, a desktop wallet, and a web-based wallet. We've chosen these three wallets because they represent a broad range of complexity and features. However, the selection of these wallets is not an endorsement of their quality or security. They are simply a good starting place for demonstrations and testing.

Remember that for a wallet application to work, it must have access to your private keys, so it is vital that you only download and use wallet applications from sources you trust. Fortunately, in general, the more popular a wallet application is, the more trustworthy it is likely to be. Nevertheless, it is good practice to avoid "putting all your eggs in one basket" and have your Ethereum accounts spread across a couple of wallets.

The following are some good starter wallets:

MetaMask

MetaMask is a browser extension wallet that runs in your browser (Chrome, Firefox, Opera, or Brave Browser). It is easy to use and convenient for testing, as it is able to connect to a variety of Ethereum nodes and test blockchains. MetaMask is a web-based wallet.

Jaxx

Jaxx is a multiplatform and multicurrency wallet that runs on a variety of operating systems, including Android, iOS, Windows, macOS, and Linux. It is often a good choice for new users as it is designed for simplicity and ease of use. Jaxx is either a mobile or a desktop wallet, depending on where you install it.

MyEtherWallet (MEW)

MyEtherWallet is a web-based wallet that runs in any browser. It has multiple sophisticated features we will explore in many of our examples. MyEtherWallet is a web-based wallet.

Emerald Wallet

Emerald Wallet is designed to work with the Ethereum Classic blockchain, but is compatible with other Ethereum-based blockchains. It's an open source desktop application and works under Windows, macOS, and Linux. Emerald Wallet can run a full node or connect to a public remote node, working in a "light" mode. It also has a companion tool to do all operations from the command line.

We'll start by installing MetaMask on a desktop—but first, we'll briefly discuss controlling and managing keys.

CONTROL AND RESPONSIBILITY

Open blockchains like Ethereum are important because they operate as a *decentralized* system. That means lots of things, but one crucial aspect is that each user of Ethereum can—and should—control their own private keys, which are the things that control access to funds and smart contracts. We sometimes call the combination of access to funds and smart contracts an "account" or "wallet." These terms can get quite complex in their functionality, so we will go into this in more detail later. As a fundamental principle, however, it is as easy as one private key equals one "account." Some users choose to give up control over their private keys by using a

third-party custodian, such as an online exchange. In this book, we will teach you how to take control and manage your own private keys.

With control comes a big responsibility. If you lose your private keys, you lose access to your funds and contracts. No one can help you regain access—your funds will be locked forever. Here are a few tips to help you manage this responsibility:

- Do not improvise security. Use tried-and-tested standard approaches.
- The more important the account (e.g., the higher the value of the funds controlled, or the more significant the smart contracts accessible), the higher security measures should be taken.
- The highest security is gained from an air-gapped device, but this level is not required for every account.
- Never store your private key in plain form, especially digitally. Fortunately, most user interfaces today won't even let you see the raw private key.
- Private keys can be stored in an encrypted form, as a digital "keystore" file. Being encrypted, they need a password to unlock. When you are prompted to choose a password, make it strong (i.e., long and random), back it up, and don't share it. If you don't have a password manager, write it down and store it in a safe and secret place. To access your account, you need both the keystore file and the password.
- Do not store any passwords in digital documents, digital photos, screenshots, online drives, encrypted PDFs, etc. Again, do not improvise security. Use a password manager or pen and paper.
- When you are prompted to back up a key as a mnemonic word sequence, use pen and paper to make a physical backup. Do not leave that task "for later"; you will forget. These backups can be used to rebuild your private key in case you lose all the data saved on your system, or if you forget or lose your password. However, they can also be used by attackers to get your private keys, so never store them digitally, and keep the physical copy stored securely in a locked drawer or safe.
- Before transferring any large amounts (especially to new addresses), first do a small test transaction (e.g., less than \$1 value) and wait for confirmation of receipt.
- When you create a new account, start by sending only a small test transaction to the new address. Once you receive the test transaction, try sending back again from that account. There are lots of reasons account creation can go wrong, and if it has gone wrong, it is better to find out with a small loss. If the tests work, all is well.
- Public block explorers are an easy way to independently see whether a transaction has been accepted by the network. However, this convenience has a negative impact on your privacy, because you reveal your addresses to block explorers, which can track you.
- Do not send money to any of the addresses shown in this book. The private keys are listed in the book and someone will immediately take that money.

Now that we've covered some basic best practices for key management and security, let's get to work using MetaMask!

GETTING STARTED WITH METAMASK

Open the Google Chrome browser and navigate to <https://chrome.google.com/webstore/category/extensions>.

Search for "MetaMask" and click on the logo of a fox. You should see something like the result shown in [The detail page of the MetaMask Chrome extension](#).



Figure 1. The detail page of the MetaMask Chrome extension

It's important to verify that you are downloading the real MetaMask extension, as sometimes people are able to sneak malicious extensions past Google's filters. The real one:

- Shows the ID `nkbihfbeogaeaoehlefnkodbefgpgknn` in the address bar
- Is offered by <https://metamask.io>
- Has more than 1,400 reviews
-

CREATING A WALLET

Once MetaMask is installed you should see a new icon (the head of a fox) in your browser's toolbar. Click on it to get started. You will be asked to accept the terms and conditions and then to create your new Ethereum wallet by entering a password (see [The password page of the MetaMask Chrome extension](#)).

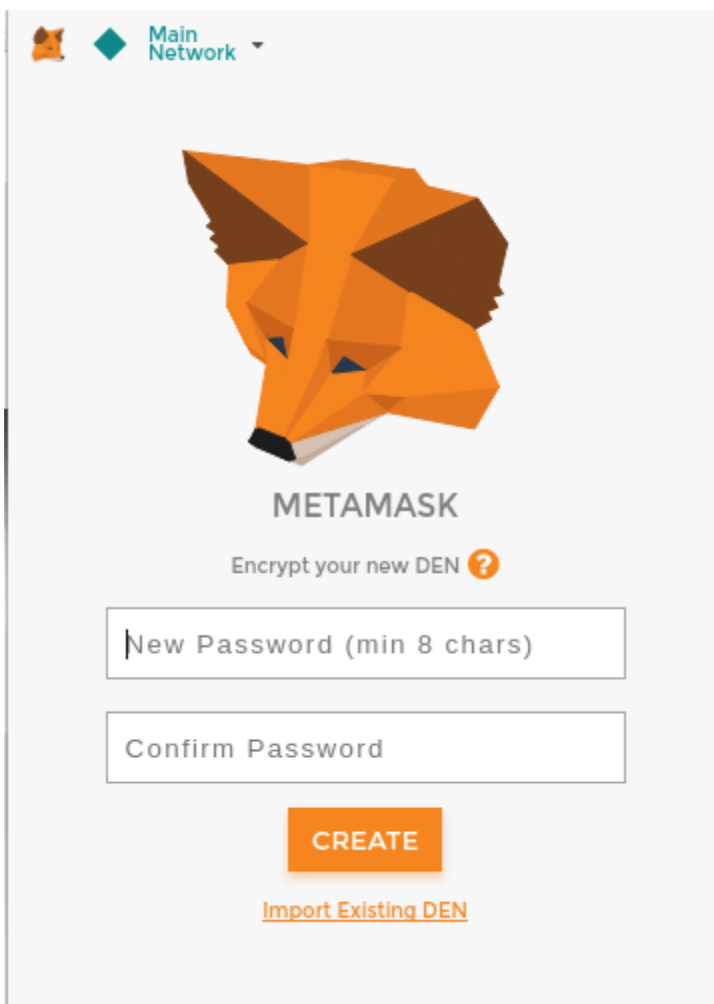


Figure 2. The password page of the MetaMask Chrome extension

Tip The password controls access to MetaMask, so that it can't be used by anyone with access to your browser.

Once you've set a password, MetaMask will generate a wallet for you and show you a *mnemonic backup* consisting of 12 English words (see [The mnemonic backup of your wallet, created by MetaMask](#)). These words can be used in any compatible wallet to recover access to your funds should something happen to MetaMask or your computer. You do not need the password for this recovery; the 12 words are sufficient.

Tip Back up your mnemonic (12 words) on paper, twice. Store the two paper backups in two separate secure locations, such as a fire-resistant safe, a locked drawer, or a safe deposit box. Treat the paper backups like cash of equivalent value to what you store in your Ethereum wallet. Anyone with access to these words can gain access and steal your money.

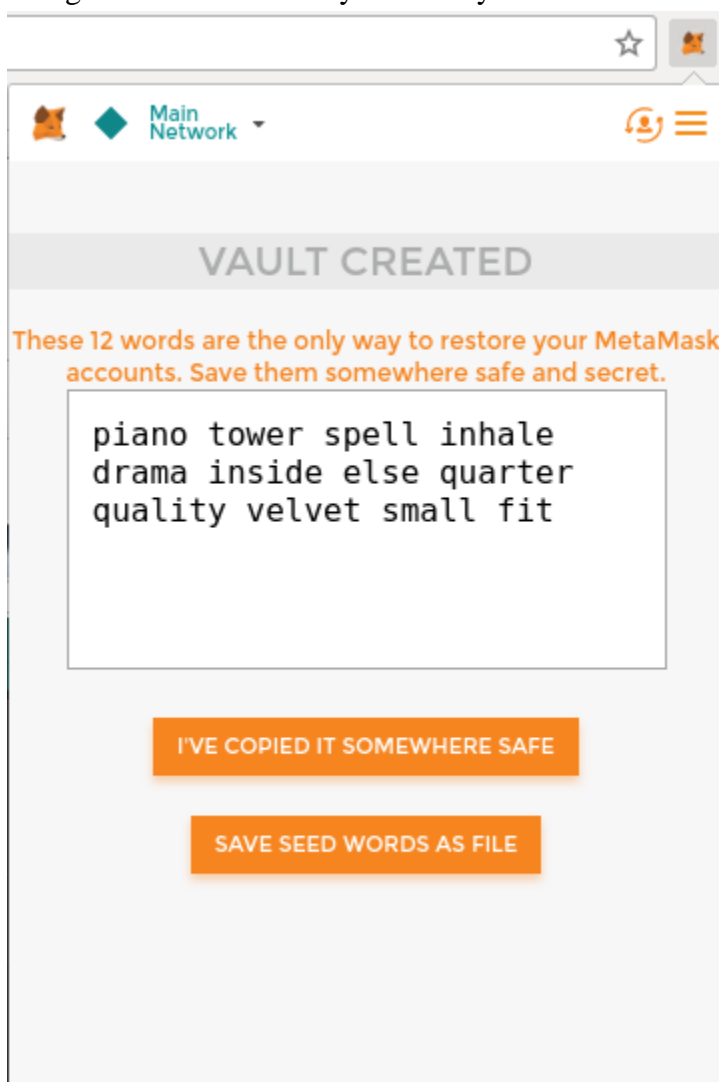


Figure 3. The mnemonic backup of your wallet, created by MetaMask

Once you have confirmed that you have stored the mnemonic securely, you'll be able to see the details of your Ethereum account, as shown in [Your Ethereum account in MetaMask](#).

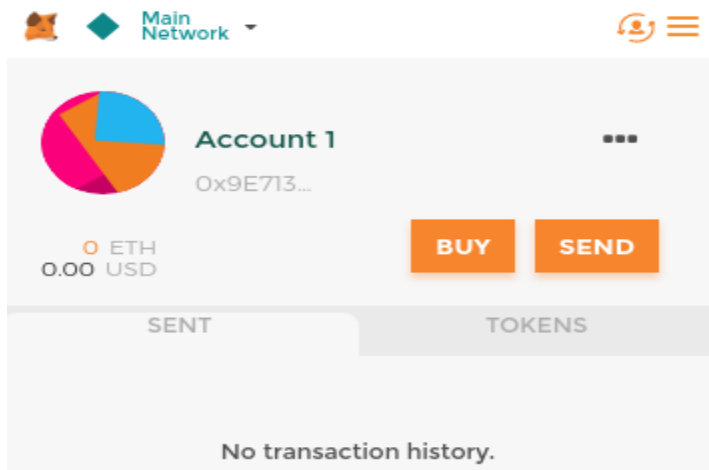


Figure 4. Your Ethereum account in MetaMask

Your account page shows the name of your account ("Account 1" by default), an Ethereum address (0x9E713... in the example), and a colorful icon to help you visually distinguish this account from other accounts. At the top of the account page, you can see which Ethereum network you are currently working on ("Main Network" in the example).

Congratulations! You have set up your first Ethereum wallet.

Switching Networks

As you can see on the MetaMask account page, you can choose between multiple Ethereum networks. By default, MetaMask will try to connect to the main network. The other choices are public testnets, any Ethereum node of your choice, or nodes running private blockchains on your own computer (localhost):

Main Ethereum Network

The main public Ethereum blockchain. Real ETH, real value, and real consequences.

Ropsten Test Network

Ethereum public test blockchain and network. ETH on this network has no value.

Kovan Test Network

Ethereum public test blockchain and network using the Aura consensus protocol with proof of authority (federated signing). ETH on this network has no value. The Kovan test network is supported by Parity only. Other Ethereum clients use the Clique consensus protocol, which was proposed later, for proof of authority-based verification.

Rinkeby Test Network

Ethereum public test blockchain and network, using the Clique consensus protocol with proof of authority (federated signing). ETH on this network has no value.

Localhost 8545

Connects to a node running on the same computer as the browser. The node can be part of any public blockchain (main or testnet), or a private testnet.

Custom RPC

Allows you to connect MetaMask to any node with a Geth-compatible Remote Procedure Call (RPC) interface. The node can be part of any public or private blockchain.

Your MetaMask wallet uses the same private key and Ethereum address on all the networks it connects to. Note to. However, your Ethereum address balance on each Ethereum network will be different. Your keys may control ether and contracts on Ropsten, for example, but not on the main network.

Getting Some Test Ether

Your first task is to get your wallet funded. You won't be doing that on the main network because real ether costs money and handling it requires a bit more experience. For now, you'll load your wallet with some testnet ether.

Switch MetaMask to the *Ropsten Test Network*. Click Buy, then click Ropsten Test Faucet. MetaMask will open a new web page, as shown in [MetaMask Ropsten Test Faucet](#).

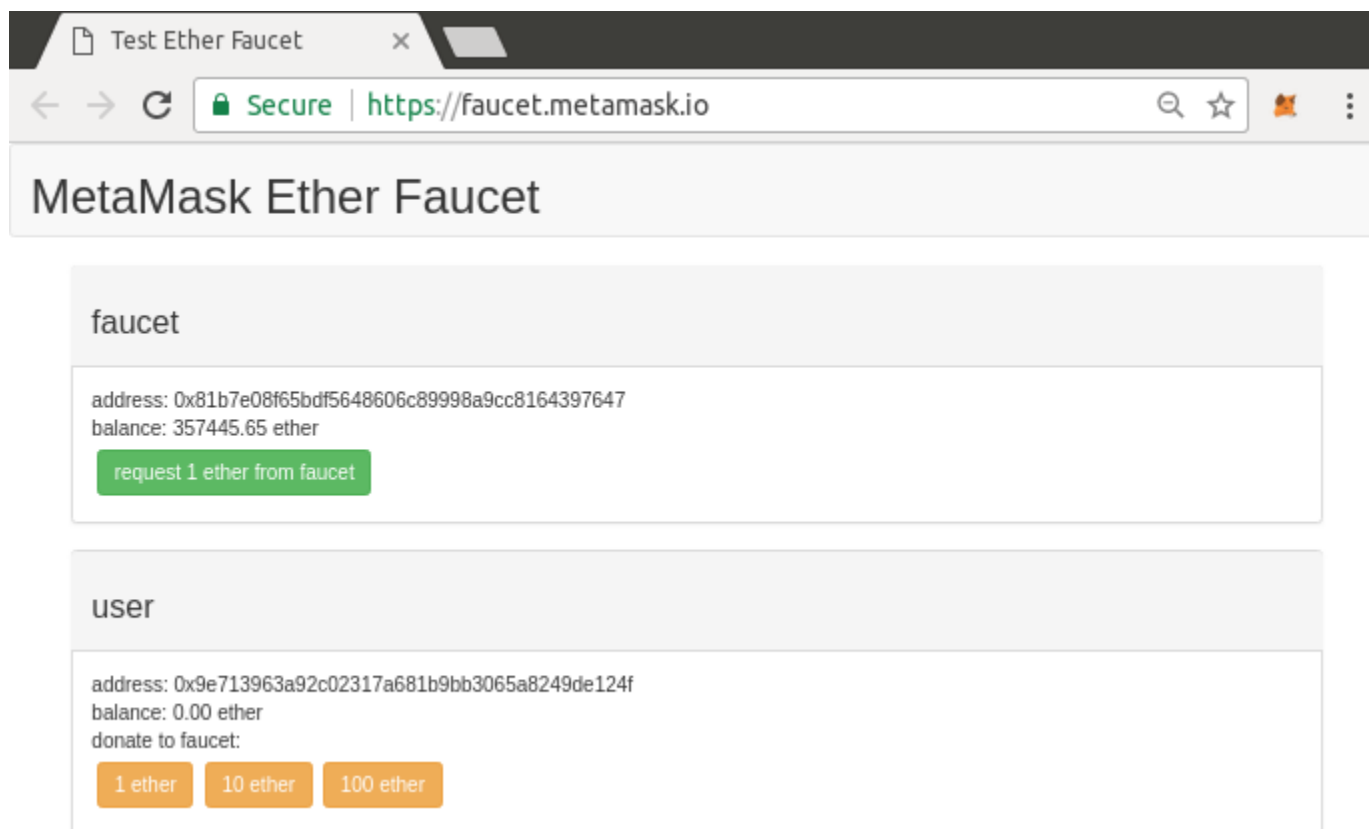


Figure 5. MetaMask Ropsten Test Faucet

You may notice that the web page already contains your MetaMask wallet's Ethereum address. MetaMask integrates Ethereum-enabled web pages with your MetaMask wallet and can "see" Ethereum addresses on the web page, allowing you, for example, to send a payment to an online shop displaying an Ethereum address. MetaMask can also populate the web page with your own wallet's address as a recipient address if the web page requests it. In this page, the faucet application is asking MetaMask for a wallet address to send test ether to.

Click the green "request 1 ether from faucet" button. You will see a transaction ID appear in the lower part of the page. The faucet app has created a transaction—a payment to you. The transaction ID looks like this:

0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57

In a few seconds, the new transaction will be mined by the Ropsten miners and your MetaMask wallet will show a balance of 1 ETH. Click on the transaction ID and your browser will take you to a *block explorer*, which is a website that allows you to visualize and explore blocks, addresses, and transactions. MetaMask uses the [Etherscan block explorer](#), one of the more popular Ethereum block explorers. The transaction containing the payment from the Ropsten Test Faucet is shown in [Etherscan Ropsten block explorer](#).

The screenshot shows the Etherscan Ropsten (Revival) TESTNET interface. The transaction hash 0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57 is entered in the search bar. The transaction details are as follows:

Transaction Information	
TxHash:	0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57
TxReceipt Status:	Success
Block Height:	2546420 (3 block confirmations)
TimeStamp:	1 min ago (Jan-29-2018 05:19:35 PM +UTC)
From:	0x81b7e08f65bdf5648606c89998a9cc8164397647
To:	0x9e713963a92c02317a681b9bb3065a8249de124f
Value:	1 Ether (\$0.00)

Figure 6. Etherscan Ropsten block explorer

The transaction has been recorded on the Ropsten blockchain and can be viewed at any time by anyone, simply by searching for the transaction ID, or [visiting the link](#).

Try visiting that link, or entering the transaction hash into the *ropsten.etherscan.io* website, to see it for yourself.

Sending Ether from MetaMask

Once you've received your first test ether from the Ropsten Test Faucet, you can experiment with sending ether by trying to send some back to the faucet. As you can see on the Ropsten Test Faucet page, there is an option to "donate" 1 ETH to the faucet. This option is available so that once you're done testing, you can return the remainder of your test ether, so that someone else can use it next. Even though test ether has no value, some people hoard it, making it difficult for everyone else to use the test networks. Hoarding test ether is frowned upon!

Fortunately, we are not test ether hoarders. Click the orange "1 ether" button to tell MetaMask to create a transaction paying the faucet 1 ether. MetaMask will prepare a transaction and pop up a window with the confirmation, as shown in [Sending 1 ether to the faucet](#).

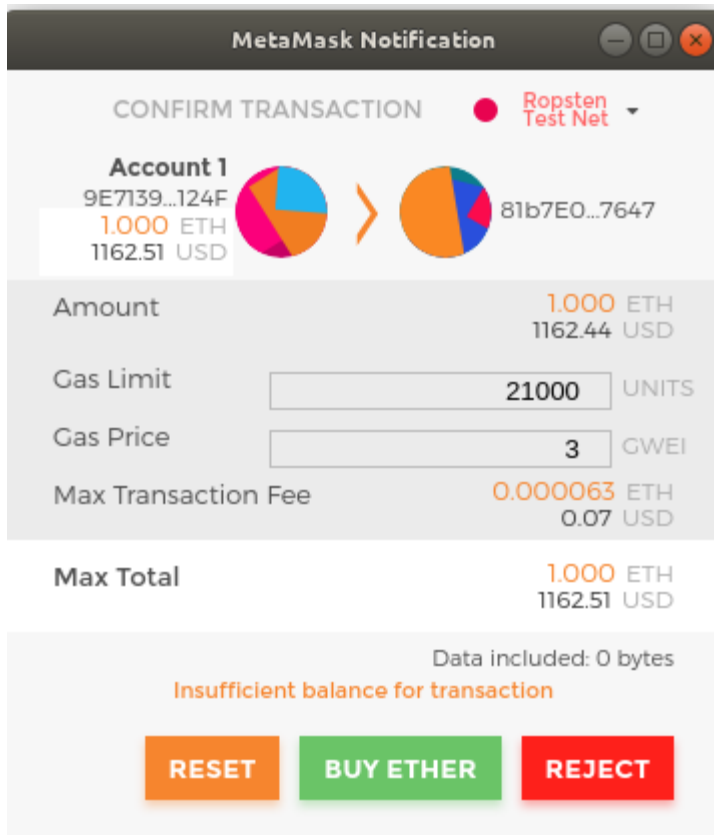


Figure 7. Sending 1 ether to the faucet

Oops! You probably noticed you can't complete the transaction—MetaMask says you have an insufficient balance. At first glance this may seem confusing: you have 1 ETH, you want to send 1 ETH, so why is MetaMask saying you have insufficient funds?

The answer is because of the cost of *gas*. Every Ethereum transaction requires payment of a fee, which is collected by the miners to validate the transaction. The fees in Ethereum are charged in a virtual currency called gas. You pay for the gas with ether, as part of the transaction.

Fees are required on the test networks too. Without fees, a test network would behave differently from the main network, making it an inadequate testing platform. Fees also protect the test networks from DoS attacks and poorly constructed contracts (e.g., infinite loops), much like they protect the main network.

When you sent the transaction, MetaMask calculated the average gas price of recent successful transactions at 3 gwei, which stands for gigawei. Wei is the smallest subdivision of the ether currency, as we discussed in [Ether Currency Units](#). The gas limit is set at the cost of sending a basic transaction, which is 21,000 gas units. Therefore, the maximum amount of ETH you will spend is $3 * 21,000 \text{ gwei} = 63,000 \text{ gwei} = 0.000063 \text{ ETH}$. (Be advised that average gas prices can fluctuate, as they are predominantly determined by miners. We will see in a later chapter how you can increase/decrease your gas limit to ensure your transaction takes precedence if need be.)

All this to say: making a 1 ETH transaction costs 1.000063 ETH. MetaMask confusingly rounds that *down* to 1 ETH when showing the total, but the actual amount you need is 1.000063 ETH and you only have 1 ETH. Click Reject to cancel this transaction.

Let's get some more test ether! Click the green "request 1 ether from the faucet" button again and wait a few seconds. Don't worry, the faucet should have plenty of ether and will give you more if you ask.

Once you have a balance of 2 ETH, you can try again. This time, when you click the orange "1 ether" donation button, you have sufficient balance to complete the transaction. Click Submit when MetaMask pops up the payment window. After all of this, you should see a balance of 0.999937 ETH because you sent 1 ETH to the faucet with 0.000063 ETH in gas.

Exploring the Transaction History of an Address

By now you have become an expert in using MetaMask to send and receive test ether. Your wallet has received at least two payments and sent at least one. You can view all these transactions using the *ropsten.etherscan.io* block explorer. You can either copy your wallet address and paste it into the block explorer's search box, or have MetaMask open the page for you. Next to your account icon in MetaMask, you will see a button showing three dots. Click on it to show a menu of account-related options (see [MetaMask account context menu](#)).

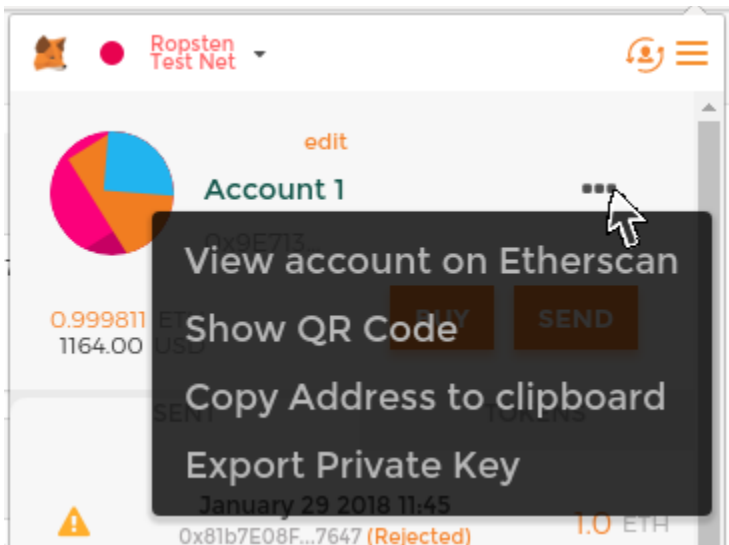


Figure 8. MetaMask account context menu

Select "View account on Etherscan" to open a web page in the block explorer showing your account's transaction history, as shown in [Address transaction history on Etherscan](#).

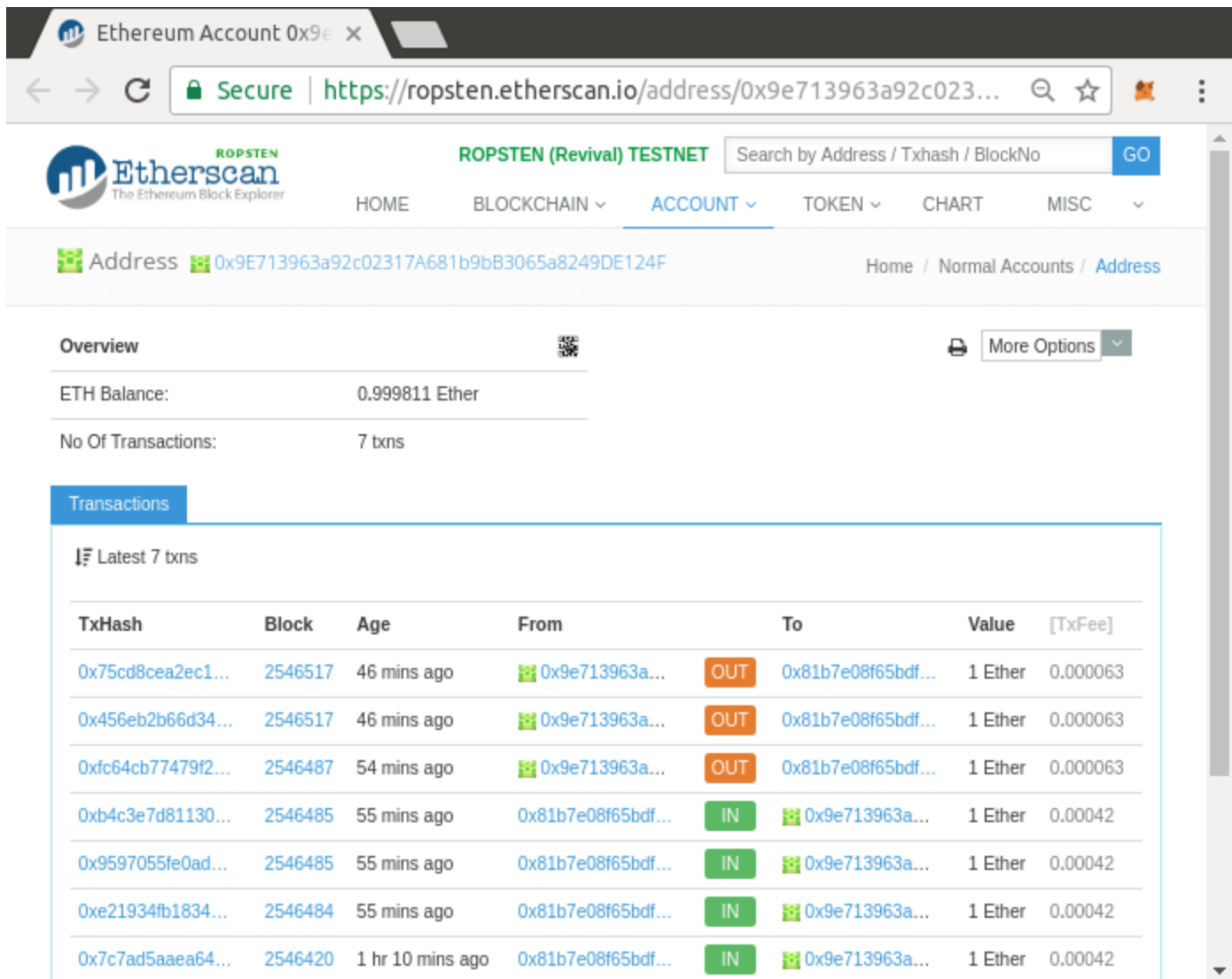


Figure 9. Address transaction history on Etherscan

Here you can see the entire transaction history of your Ethereum address. It shows all the transactions recorded on the Ropsten blockchain where your address is the sender or recipient. Click on a few of these transactions to see more details.

You can explore the transaction history of any address. Take a look at the transaction history of the Ropsten Test Faucet address (hint: it is the "sender" address listed in the oldest payment to your address). You can see all the test ether sent from the faucet to you and to other addresses. Every transaction you see can lead you to more addresses and more transactions. Before long you will be lost in the maze of interconnected data. Public blockchains contain an enormous wealth of information, all of which can be explored programmatically, as we will see in future examples.

Introducing the World Computer

You've now created a wallet and sent and received ether. So far, we've treated Ethereum as a cryptocurrency. But Ethereum is much, much more. In fact, the cryptocurrency function is subservient to Ethereum's function as a decentralized world computer. Ether is meant to be used to pay for running *smart contracts*, which are computer programs that run on an emulated computer called the *Ethereum Virtual Machine* (EVM).

The EVM is a global singleton, meaning that it operates as if it were a global, single-instance computer, running everywhere. Each node on the Ethereum network runs a local copy of the EVM to validate contract execution, while the Ethereum blockchain records the changing *state* of this world computer as it processes transactions and smart contracts. We'll discuss this in much greater detail in [\[evm_chapter\]](#).

Externally Owned Accounts (EOAs) and Contracts

The type of account you created in the MetaMask wallet is called an *externally owned account* (EOA). Externally owned accounts are those that have a private key; having the private key means control over access to funds or contracts. Now, you're probably guessing there is another type of account. That other type of account is a *contract account*. A contract account has smart contract code, which a simple EOA can't have. Furthermore, a contract account does not have a private key. Instead, it is owned (and controlled) by the logic of its smart contract code: the software program recorded on the Ethereum blockchain at the contract account's creation and executed by the EVM.

Contracts have addresses, just like EOAs. Contracts can also send and receive ether, just like EOAs. However, when a transaction destination is a contract address, it causes that contract to *run* in the EVM, using the transaction, and the transaction's data, as its input. In addition to ether, transactions can contain *data* indicating which specific function in the contract to run and what parameters to pass to that function. In this way, transactions can *call* functions within contracts.

Note that because a contract account does not have a private key, it cannot *initiate* a transaction. Only EOAs can initiate transactions, but contracts can *react* to transactions by calling other contracts, building complex execution paths. One typical use of this is an EOA sending a request transaction to a multisignature smart contract wallet to send some ETH on to another address. A typical DApp programming pattern is to have Contract A calling Contract B in order to maintain a shared state across users of Contract A.

In the next few sections, we will write our first contract. You will then learn how to create, fund, and use that contract with your MetaMask wallet and test ether on the Ropsten test network.

A Simple Contract: A Test Ether Faucet

Ethereum has many different high-level languages, all of which can be used to write a contract and produce EVM bytecode. You can read about many of the most prominent and interesting ones in [\[high_level_languages\]](#). One high-level language is by far the dominant choice for smart contract programming: Solidity. Solidity was created by Dr. Gavin Wood, the coauthor of this book, and has become the most widely used language in Ethereum (and beyond). We'll use Solidity to write our first contract.

For our first example ([Faucet.sol: A Solidity contract implementing a faucet](#)), we will write a contract that controls a *faucet*. You've already used a faucet to get test ether on the Ropsten test network. A faucet is a relatively simple thing: it gives out ether to any address that asks, and can be refilled periodically. You can implement a faucet as a wallet controlled by a human or a web server.

Example 1. Faucet.sol: A Solidity contract implementing a faucet

link:code/Solidity/Faucet.sol[]

You will find all the code samples for this book in the *code* subdirectory of [the book's GitHub repository](#).
Note Specifically, our *Faucet.sol* contract is in:

```
code/Solidity/Faucet.sol
```

This is a very simple contract, about as simple as we can make it. It is also a *flawed* contract, demonstrating a number of bad practices and security vulnerabilities. We will learn by examining all of its flaws in later sections. But for now, let's look at what this contract does and how it works, line by line. You will quickly notice that many elements of Solidity are similar to existing programming languages, such as JavaScript, Java, or C++.

The first line is a comment:

```
// Our first contract is a faucet!
```

Comments are for humans to read and are not included in the executable EVM bytecode. We usually put them on the line before the code we are trying to explain, or sometimes on the same line. Comments start with two forward slashes: `//`. Everything from the first slash until the end of that line is treated the same as a blank line and ignored.

The next line is where our actual contract starts:

```
contract Faucet {
```

This line declares a contract object, similar to a class declaration in other object-oriented languages. The contract definition includes all the lines between the curly braces (`{}`), which define a *scope*, much like how curly braces are used in many other programming languages.

Next, we declare the first function of the Faucet contract:

```
function withdraw(uint withdraw_amount) public {
```

The function is named `withdraw`, and it takes one unsigned integer (`uint`) argument named `withdraw_amount`. It is declared as a public function, meaning it can be called by other contracts. The function definition follows, between curly braces. The first part of the `withdraw` function sets a limit on withdrawals:

```
require(withdraw_amount <= 1000000000000000000);
```

It uses the built-in Solidity function `require` to test a precondition, that the `withdraw_amount` is less than or equal to 100,000,000,000,000,000 wei, which is the base unit of ether (see [Ether denominations and unit names](#)) and equivalent to 0.1 ether. If the `withdraw` function is called with a `withdraw_amount` greater than that amount, the `require` function here will cause contract execution to stop and fail with an *exception*. Note that statements need to be terminated with a semicolon in Solidity.

This part of the contract is the main logic of our faucet. It controls the flow of funds out of the contract by placing a limit on withdrawals. It's a very simple control but can give you a glimpse of the power of a programmable blockchain: decentralized software controlling money.

Next comes the actual withdrawal:

```
msg.sender.transfer(withdraw_amount);
```

A couple of interesting things are happening here. The `msg` object is one of the inputs that all contracts can access. It represents the transaction that triggered the execution of this contract. The attribute `sender` is the sender address of the transaction. The function `transfer` is a built-in function that transfers ether from the current contract to the address of the sender. Reading it backward, this means transfer to the sender of the `msg` that

triggered this contract execution. The transfer function takes an amount as its only argument. We pass the `withdraw_amount` value that was the parameter to the `withdraw` function declared a few lines earlier.

The very next line is the closing curly brace, indicating the end of the definition of our `withdraw` function.

Next, we declare one more function:

```
function () public payable {}
```

This function is a so-called *fallback* or *default* function, which is called if the transaction that triggered the contract didn't name any of the declared functions in the contract, or any function at all, or didn't contain data. Contracts can have one such default function (without a name) and it is usually the one that receives ether. That's why it is defined as a public and payable function, which means it can accept ether into the contract. It doesn't do anything, other than accept the ether, as indicated by the empty definition in the curly braces (`{}`). If we make a transaction that sends ether to the contract address, as if it were a wallet, this function will handle it.

Right below our default function is the final closing curly brace, which closes the definition of the contract `Faucet`. That's it!

Compiling the Faucet Contract

Now that we have our first example contract, we need to use a Solidity compiler to convert the Solidity code into EVM bytecode so it can be executed by the EVM on the blockchain itself.

The Solidity compiler comes as a standalone executable, as part of various frameworks, and bundled in Integrated Development Environments (IDEs). To keep things simple, we will use one of the more popular IDEs, called *Remix*.

Use your Chrome browser (with the MetaMask wallet you installed earlier) to navigate to the Remix IDE at <https://remix.ethereum.org>.

When you first load Remix, it will start with a sample contract called *ballot.sol*. We don't need that, so close it by clicking the x on the corner of the tab, as seen in [Close the default example tab](#).

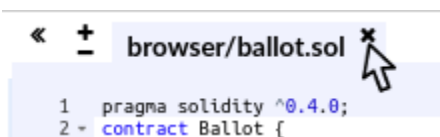


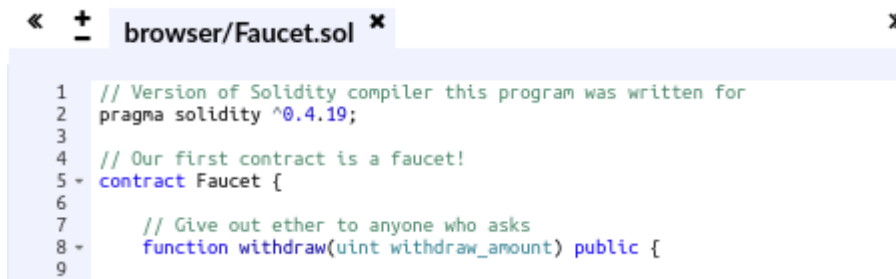
Figure 10. Close the default example tab

Now, add a new tab by clicking on the circular plus sign in the top-left toolbar, as seen in [Click the plus sign to open a new tab](#). Name the new file *Faucet.sol*.



Figure 11. Click the plus sign to open a new tab

Once you have the new tab open, copy and paste the code from our example *Faucet.sol*, as seen in [Copy the Faucet example code into the new tab](#).



```
1 // Version of Solidity compiler this program was written for
2 pragma solidity ^0.4.19;
3
4 // Our first contract is a faucet!
5 contract Faucet {
6
7     // Give out ether to anyone who asks
8     function withdraw(uint withdraw_amount) public {
9
```

Figure 12. Copy the Faucet example code into the new tab

Once you have loaded the *Faucet.sol* contract into the Remix IDE, the IDE will automatically compile the code. If all goes well, you will see a green box with "Faucet" in it appear on the right, under the Compile tab, confirming the successful compilation (see [Remix successfully compiles the Faucet.sol contract](#)).

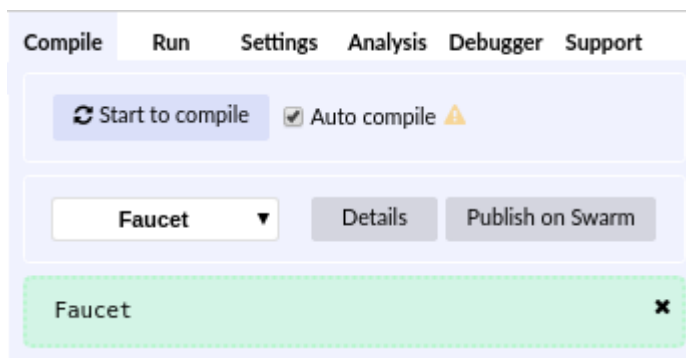


Figure 13. Remix successfully compiles the Faucet.sol contract

If something goes wrong, the most likely problem is that the Remix IDE is using a version of the Solidity compiler that is different from 0.4.19. In that case, our pragma directive will prevent *Faucet.sol* from compiling. To change the compiler version, go to the Settings tab, set the version to 0.4.19, and try again.

The Solidity compiler has now compiled our *Faucet.sol* into EVM bytecode. If you are curious, the bytecode looks like this:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN
STOP PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI
PUSH1 0x0 CALLDATALOAD PUSH29
0x1000000000000000000000000000000000000000000000000000000000000000
SWAP1 DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI
JUMPDEST STOP JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD SWAP1
SWAP2 SWAP1 POP POP PUSH1 0x61 JUMP JUMPDEST STOP JUMPDEST PUSH8
0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AND
PUSH2 0x8FC DUP3 SWAP1 DUP2 ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1
0x40 MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP POP ISZERO
ISZERO PUSH1 0xB6 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75 GASLIMIT CALLVALUE LOG4 0x5f
PUSH24 0x7541F409787592C988A079407FB28B4AD000290000000000
```

Aren't you glad you are using a high-level language like Solidity instead of programming directly in EVM bytecode? Me too!

Creating the Contract on the Blockchain

So, we have a contract. We've compiled it into bytecode. Now, we need to "register" the contract on the Ethereum blockchain. We will be using the Ropsten testnet to test our contract, so that's the blockchain we want to submit it to.

Registering a contract on the blockchain involves creating a special transaction whose destination is the address 0x0000000000000000000000000000000000, also known as the *zero address*. The zero address is a special address that tells the Ethereum blockchain that you want to register a contract. Fortunately, the Remix IDE will handle all of that for you and send the transaction to MetaMask.

First, switch to the Run tab and select Injected Web3 in the Environment drop-down selection box. This connects the Remix IDE to the MetaMask wallet, and through MetaMask to the Ropsten test network. Once you do that, you can see Ropsten under Environment. Also, in the Account selection box it shows the address of your wallet (see [Remix IDE Run tab, with Injected Web3 environment selected](#)).

Compile

Run

Analysis


Testing


Debugger

Settings

Supp

Environment

Injected Web3  Ropsten (3)

Account 

0x9e7...e124f (0.131624865400000002 ethe



Gas limit

3000000

Value

0

wei

Faucet  

Deploy

or

At Address

Load contract from Address

Figure 14. Remix IDE Run tab, with Injected Web3 environment selected

Right below the Run settings you just confirmed is the Faucet contract, ready to be created. Click on the Deploy button shown in [Remix IDE Run tab, with Injected Web3 environment selected](#).

Remix will construct the special "creation" transaction and MetaMask will ask you to approve it, as shown in [MetaMask showing the contract creation transaction](#). You'll notice the contract creation transaction has no ether in it, but it has 258 bytes of data (the compiled contract) and will consume 10 gwei in gas. Click Submit to approve it.

The image shows the MetaMask transaction confirmation interface. At the top, it says "CONFIRM TRANSACTION" and "Ropsten Test Net". Below this, "Account 1" is shown with address "9E7139...124F", "1.001 ETH", and "996.78 USD". A pie chart icon and a document icon labeled "New Contract" are also visible. The transaction details are as follows:

Field	Value	Unit
Amount	0.00	ETH / USD
Gas Limit	113558	UNITS
Gas Price	10	GWEI
Max Transaction Fee	0.001135	ETH / 1.13 USD
Max Total	0.001135	ETH / 1.13 USD

At the bottom, it states "Data included: 258 bytes". There are three buttons: "RESET" (orange), "SUBMIT" (green, with a mouse cursor), and "REJECT" (red).

Figure 15. MetaMask showing the contract creation transaction

Now you have to wait. It will take about 15 to 30 seconds for the contract to be mined on Ropsten. Remix won't appear to be doing much, but be patient.

Once the contract is created, it appears at the bottom of the Run tab (see [The Faucet contract is ALIVE!](#)).



Figure 16. The Faucet contract is ALIVE!

Notice that the Faucet contract now has an address of its own: Remix shows it as "Faucet at 0x72e...c7829" (although your address, the random letters and numbers, will be different). The small clipboard symbol to the right allows you to copy the contract address to your clipboard. We will use that in the next section.

Interacting with the Contract

Let's recap what we've learned so far: Ethereum contracts are programs that control money, which run inside a virtual machine called the EVM. They are created by a special transaction that submits their bytecode to be recorded on the blockchain. Once they are created on the blockchain, they have an Ethereum address, just like wallets. Anytime someone sends a transaction to a contract address it causes the contract to run in the EVM, with the transaction as its input. Transactions sent to contract addresses may have ether or data or both. If they contain ether, it is "deposited" to the contract balance. If they contain data, the data can specify a named function in the contract and call it, passing arguments to the function.

Viewing the Contract Address in a Block Explorer

We now have a contract recorded on the blockchain, and we can see it has an Ethereum address. Let's check it out in the *ropsten.etherscan.io* block explorer and see what a contract looks like. In the Remix IDE, copy the address of the contract by clicking the clipboard icon next to its name (see [Copy the contract address from Remix](#)).

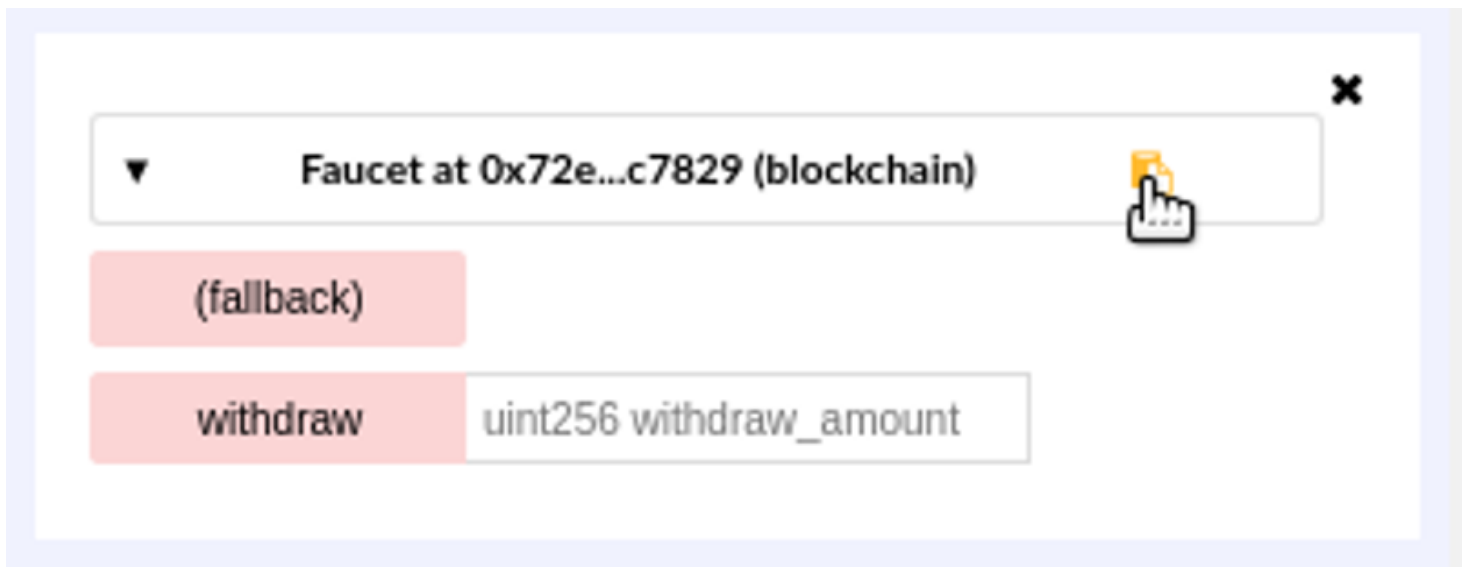


Figure 17. Copy the contract address from Remix

Keep Remix open; we'll come back to it again later. Now, navigate your browser to *ropsten.etherscan.io* and paste the address into the search box. You should see the contract's Ethereum address history, as shown in [View the Faucet contract address in the Etherscan block explorer](#).

Etherscan ROPSTEN (Revival) TESTNET

Search by Address / Txhash / BlockNo GO

HOME BLOCKCHAIN ACCOUNT TOKEN CHART MISC

Contract Address 0x72E9D27f206fD62eaC5B81129aa3e774015c7829 Home / Contract Accounts / Address

Contract Overview

ETH Balance: 0 Ether

No Of Transactions: 1 txn

Misc

Contract Creator 0x9e713963a92c... at txn 0x90333f7ecc9d...

More Options

Transactions

Contract Code

Latest 1 txn

TxHash	Block	Age	From	To	Value	[TxFee]
0x90333f7ecc9d...	2567995	16 hrs 48 mins ago	0x9e713963a92c...	IN Contract Creation	0 Ether	0.00113558

[Download CSV Export]

Figure 18. View the Faucet contract address in the Etherscan block explorer

Funding the Contract

For now, the contract only has one transaction in its history: the contract creation transaction. As you can see, the contract also has no ether (zero balance). That's because we didn't send any ether to the contract in the creation transaction, even though we could have.

Our faucet needs funds! Our first project will be to use MetaMask to send ether to the contract. You should still have the address of the contract in your clipboard (if not, copy it again from Remix). Open MetaMask, and send 1 ether to it, exactly as you would to any other Ethereum address (see [Send 1 ether to the contract address](#)).

Account 1 9E7139...124F 1.000 ETH 947.41 USD

72E9D2...7829

Amount 1.000 ETH 946.78 USD

Gas Limit 31560 UNITS

Gas Price 5 GWEI

Max Transaction Fee 0.000157 ETH 0.15 USD

Max Total 1.000 ETH 946.93 USD

Data included: 0 bytes

RESET SUBMIT REJECT

Figure 19. Send 1 ether to the contract address

In a minute, if you reload the Etherscan block explorer, it will show another transaction to the contract address and an updated balance of 1 ether.

Remember the unnamed default public payable function in our *Faucet.sol* code? It looked like this:

```
function () public payable {}
```

When you sent a transaction to the contract address, with no data specifying which function to call, it called this default function. Because we declared it as payable, it accepted and deposited the 1 ether into the contract's account balance. Your transaction caused the contract to run in the EVM, updating its balance. You have funded your faucet!

Withdrawing from Our Contract

Next, let's withdraw some funds from the faucet. To withdraw, we have to construct a transaction that calls the withdraw function and passes a withdraw_amount argument to it. To keep things simple for now, Remix will construct that transaction for us and MetaMask will present it for our approval.

Return to the Remix tab and look at the contract on the Run tab. You should see a red box labeled withdraw with a field entry labeled uint256 withdraw_amount (see [The withdraw function of Faucet.sol, in Remix](#)).



Figure 20. The withdraw function of Faucet.sol, in Remix

This is the Remix interface to the contract. It allows us to construct transactions that call the functions defined in the contract. We will enter a withdraw_amount and click the withdraw button to generate the transaction.

First, let's figure out the withdraw_amount. We want to try and withdraw 0.1 ether, which is the maximum amount allowed by our contract. Remember that all currency values in Ethereum are denominated in wei internally, and our withdraw function expects the withdraw_amount to be denominated in wei too. The amount we want is 0.1 ether, which is 100,000,000,000,000,000 wei (a 1 followed by 17 zeros).

Tip Due to a limitation in JavaScript, a number as large as 10^{17} cannot be processed by Remix. Instead, we enclose it in double quotes, to allow Remix to receive it as a string and manipulate it as a BigNumber. If we don't enclose it in quotes, the Remix IDE will fail to process it and display "Error encoding arguments: Error: Assertion failed."

Type "100000000000000000" (with the quotes) into the withdraw_amount box and click on the withdraw button (see [Click "withdraw" in Remix to create a withdrawal transaction](#)).

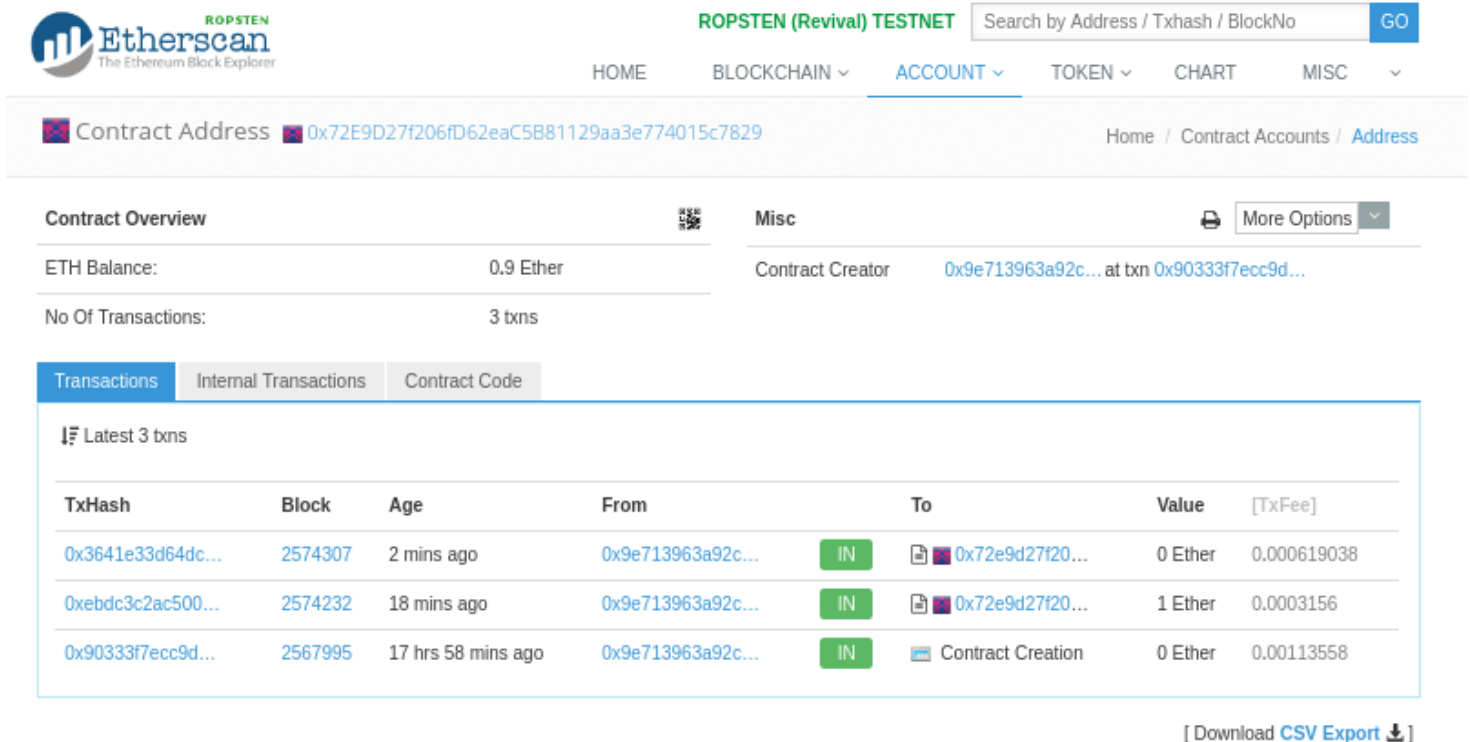


Figure 23. Etherscan shows the transaction calling the withdraw function

We now see a new transaction with the contract address as the destination and a value of 0 ether. The contract balance has changed and is now 0.9 ether because it sent us 0.1 ether as requested. But we don't see an "OUT" transaction in the *contract address history*.

Where's the outgoing withdrawal? A new tab has appeared on the contract's address history page, named Internal Transactions. Because the 0.1 ether transfer originated from the contract code, it is an internal transaction (also called a *message*). Click on that tab to see it (see [Etherscan shows the internal transaction transferring ether out from the contract](#)).

This "internal transaction" was sent by the contract in this line of code (from the `withdraw` function in *Faucet.sol*):

```
msg.sender.transfer(withdraw_amount);
```

To recap: you sent a transaction from your MetaMask wallet that contained data instructions to call the `withdraw` function with a `withdraw_amount` argument of 0.1 ether. That transaction caused the contract to run inside the EVM. As the EVM ran the *Faucet* contract's `withdraw` function, first it called the `require` function and validated that the requested amount was less than or equal to the maximum allowed withdrawal of 0.1 ether. Then it called the `transfer` function to send you the ether. Running the `transfer` function generated an internal transaction that deposited 0.1 ether into your wallet address, from the contract's balance. That's the one shown on the Internal Transactions tab in Etherscan.

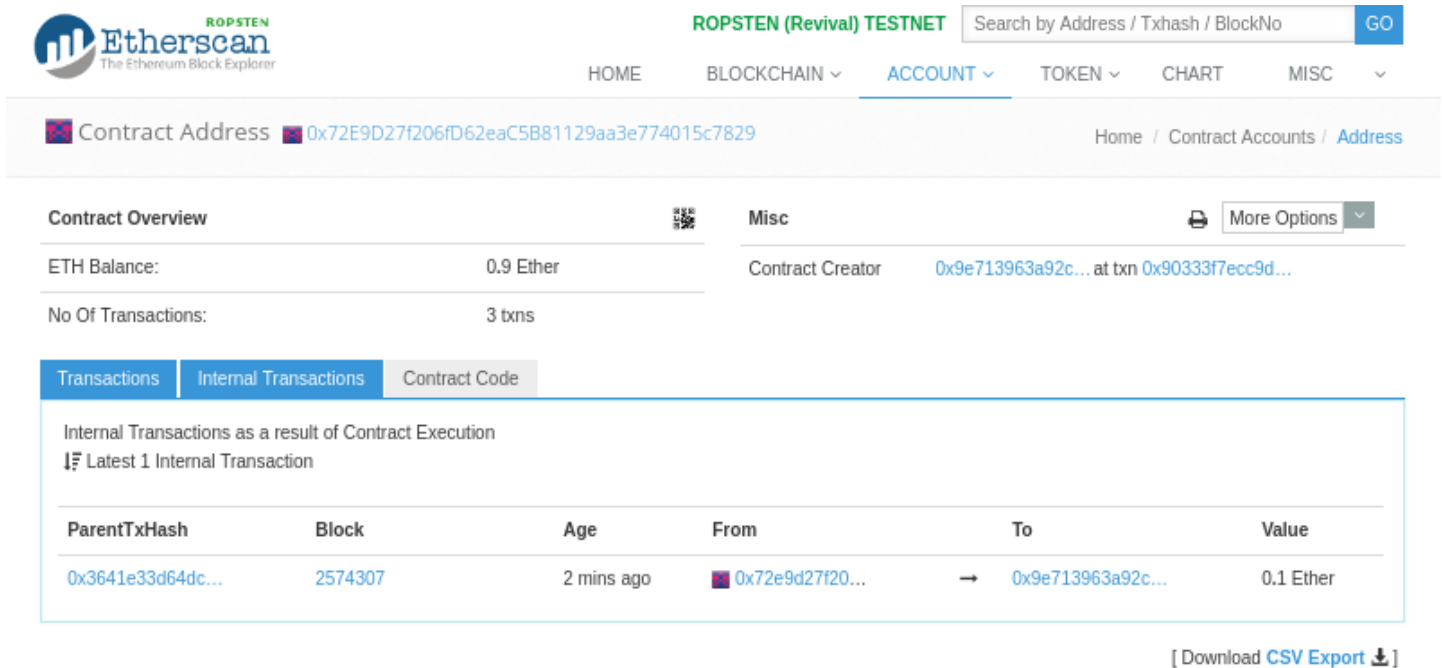


Figure 24. Etherscan shows the internal transaction transferring ether out from the contract

Conclusions

In this chapter, you set up a wallet using MetaMask and funded it using a faucet on the Ropsten test network. You received ether into your wallet's Ethereum address, then you sent ether to the faucet's Ethereum address.

Next, you wrote a faucet contract in Solidity. You used the Remix IDE to compile the contract into EVM bytecode, then used Remix to form a transaction and created the Faucet contract on the Ropsten blockchain. Once created, the Faucet contract had an Ethereum address, and you sent it some ether. Finally, you constructed a transaction to call the withdraw function and successfully asked for 0.1 ether. The contract checked the request and sent you 0.1 ether with an internal transaction.

It may not seem like much, but you've just successfully interacted with software that controls money on a decentralized world computer.

We will do a lot more smart contract programming in [\[smart contracts chapter\]](#) and learn about best practices and security considerations in [\[smart contract security\]](#).