**Ethereum Pet Shop**

This tutorial will take you through the process of building your first dapp—an adoption tracking system for a pet shop!

This tutorial is meant for those with a basic knowledge of Ethereum and smart contracts, who have some knowledge of HTML and JavaScript, but who are new to dapps.

**Note**: For Ethereum basics, please read the Truffle [Ethereum Overview](https://truffleframework.com/tutorials/ethereum-overview) tutorial before proceeding.

In this tutorial we will be covering:

1. Setting up the development environment
2. Creating a Truffle project using a Truffle Box
3. Writing the smart contract
4. Compiling and migrating the smart contract
5. Testing the smart contract
6. Creating a user interface to interact with the smart contract
7. Interacting with the dapp in a browser

**Background**

Pete Scandlon of Pete's Pet Shop is interested in using Ethereum as an efficient way to handle their pet adoptions. The store has space for 16 pets at a given time, and they already have a database of pets. As an initial proof of concept, **Pete wants to see a dapp which associates an Ethereum address with a pet to be adopted.**

The website structure and styling will be supplied. **Our job is to write the smart contract and front-end logic for its usage.**

**1. Setting up the development environment**

There are a few technical requirements before we start. Please install the following:

* [Node.js v6+ LTS and npm](https://nodejs.org/en/) (comes with Node)
* [Git](https://git-scm.com/)

Once we have those installed, we only need one command to install Truffle:

npm install -g truffle

To verify that Truffle is installed properly, type truffle version on a terminal. If you see an error, make sure that your npm modules are added to your path.

We also will be using [Ganache](https://truffleframework.com/ganache), a personal blockchain for Ethereum development you can use to deploy contracts, develop applications, and run tests. You can download Ganache by navigating to <http://truffleframework.com/ganache> and clicking the "Download" button.

**Note**: If you are developing in an environment without a graphical interface, you can also use Truffle Develop, Truffle's built-in personal blockchain, instead of Ganache. You will need to change some settings—such as the port the blockchain runs on—to adapt the tutorial for Truffle Develop.

**2.****Creating a Truffle project using a Truffle Box**

1. Truffle initializes in the current directory, so first create a directory in your development folder of choice and then moving inside it.
2. mkdir pet-shop-tutorial
3. cd pet-shop-tutorial
4. We've created a special [Truffle Box](https://truffleframework.com/boxes) just for this tutorial called pet-shop, which includes the basic project structure as well as code for the user interface. Use the truffle unbox command to unpack this Truffle Box.
5. truffle unbox pet-shop

**Note**: Truffle can be initialized a few different ways. Another useful initialization command is truffle init, which creates an empty Truffle project with no example contracts included. For more information, please see the documentation on [Creating a project](https://truffleframework.com/docs/getting_started/project).

**Directory structure**

The default Truffle directory structure contains the following:

* contracts/: Contains the [Solidity](https://solidity.readthedocs.io/) source files for our smart contracts. There is an important contract in here called Migrations.sol, which we'll talk about later.
* migrations/: Truffle uses a migration system to handle smart contract deployments. A migration is an additional special smart contract that keeps track of changes.
* test/: Contains both JavaScript and Solidity tests for our smart contracts
* truffle.js: Truffle configuration file

The pet-shop Truffle Box has extra files and folders in it, but we won't worry about those just yet.

**3.****Writing the smart contract**

We'll start our dapp by writing the smart contract that acts as the back-end logic and storage.

1. Create a new file named Adoption.sol in the contracts/ directory.
2. Add the following content to the file:
3. pragma solidity ^0.4.17;
4. contract Adoption {
5. }

Things to notice:

* The minimum version of Solidity required is noted at the top of the contract: pragma solidity ^0.4.17;. The pragma command means "*additional information that only the compiler cares about*", while the caret symbol (^) means "*the version indicated or higher*".
* Like JavaScript or PHP, statements are terminated with semicolons.

**Variable setup**

Solidity is a statically-typed language, meaning data types like strings, integers, and arrays must be defined. **Solidity has a unique type called an address**. Addresses are Ethereum addresses, stored as 20 byte values. Every account and smart contract on the Ethereum blockchain has an address and can send and receive Ether to and from this address.

1. Add the following variable on the next line after contract Adoption {.
2. address[16] public adopters;

Things to notice:

* We've defined a single variable: adopters. This is an **array** of Ethereum addresses. Arrays contain one type and can have a fixed or variable length. In this case the type is address and the length is 16.
* You'll also notice adopters is public. **Public** variables have automatic getter methods, but in the case of arrays a key is required and will only return a single value. Later, we'll write a function to return the whole array for use in our UI.

**Your first function: Adopting a pet**

Let's allow users to make adoption requests.

1. Add the following function to the smart contract after the variable declaration we set up above.
2. // Adopting a pet
3. function adopt(uint petId) public returns (uint) {
4. require(petId >= 0 && petId <= 15);
5. adopters[petId] = msg.sender;
6. return petId;
7. }

Things to notice:

* In Solidity the types of both the function parameters and output must be specified. In this case we'll be taking in a petId (integer) and returning an integer.
* We are checking to make sure petId is in range of our adopters array. Arrays in Solidity are indexed from 0, so the ID value will need to be between 0 and 15. We use the require() statement to ensure the ID is within range.
* If the ID is in range, we then add the address that made the call to our adopters array. **The address of the person or smart contract who called this function is denoted by msg.sender**.
* Finally, we return the petId provided as a confirmation.

**Your second function: Retrieving the adopters**

As mentioned above, array getters return only a single value from a given key. Our UI needs to update all pet adoption statuses, but making 16 API calls is not ideal. So our next step is to write a function to return the entire array.

1. Add the following getAdopters() function to the smart contract, after the adopt() function we added above:
2. // Retrieving the adopters
3. function getAdopters() public view returns (address[16]) {
4. return adopters;
5. }

Since adopters is already declared, we can simply return it. Be sure to specify the return type (in this case, the type for adopters) as address[16].

**4.****Compiling and migrating the smart contract**

Now that we have written our smart contract, the next steps are to compile and migrate it.

Truffle has a built-in developer console, which we call Truffle Develop, which generates a development blockchain that we can use to test deploy contracts. It also has the ability to run Truffle commands directly from the console. We will use Truffle Develop to perform most of the actions on our contract in this tutorial.

**Compilation**

Solidity is a compiled language, meaning we need to compile our Solidity to bytecode for the Ethereum Virtual Machine (EVM) to execute. Think of it as translating our human-readable Solidity into something the EVM understands.

1. In a terminal, make sure you are in the root of the directory that contains the dapp and type:
2. truffle compile

**Note**: If you're on Windows and encountering problems running this command, please see the documentation on [resolving naming conflicts on Windows](https://truffleframework.com/docs/advanced/configuration#resolving-naming-conflicts-on-windows).

You should see output similar to the following:

Compiling ./contracts/Migrations.sol...

Compiling ./contracts/Adoption.sol...

Writing artifacts to ./build/contracts

**Migration**

Now that we've successfully compiled our contracts, it's time to migrate them to the blockchain!

**A migration is a deployment script meant to alter the state of your application's contracts**, moving it from one state to the next. For the first migration, you might just be deploying new code, but over time, other migrations might move data around or replace a contract with a new one.

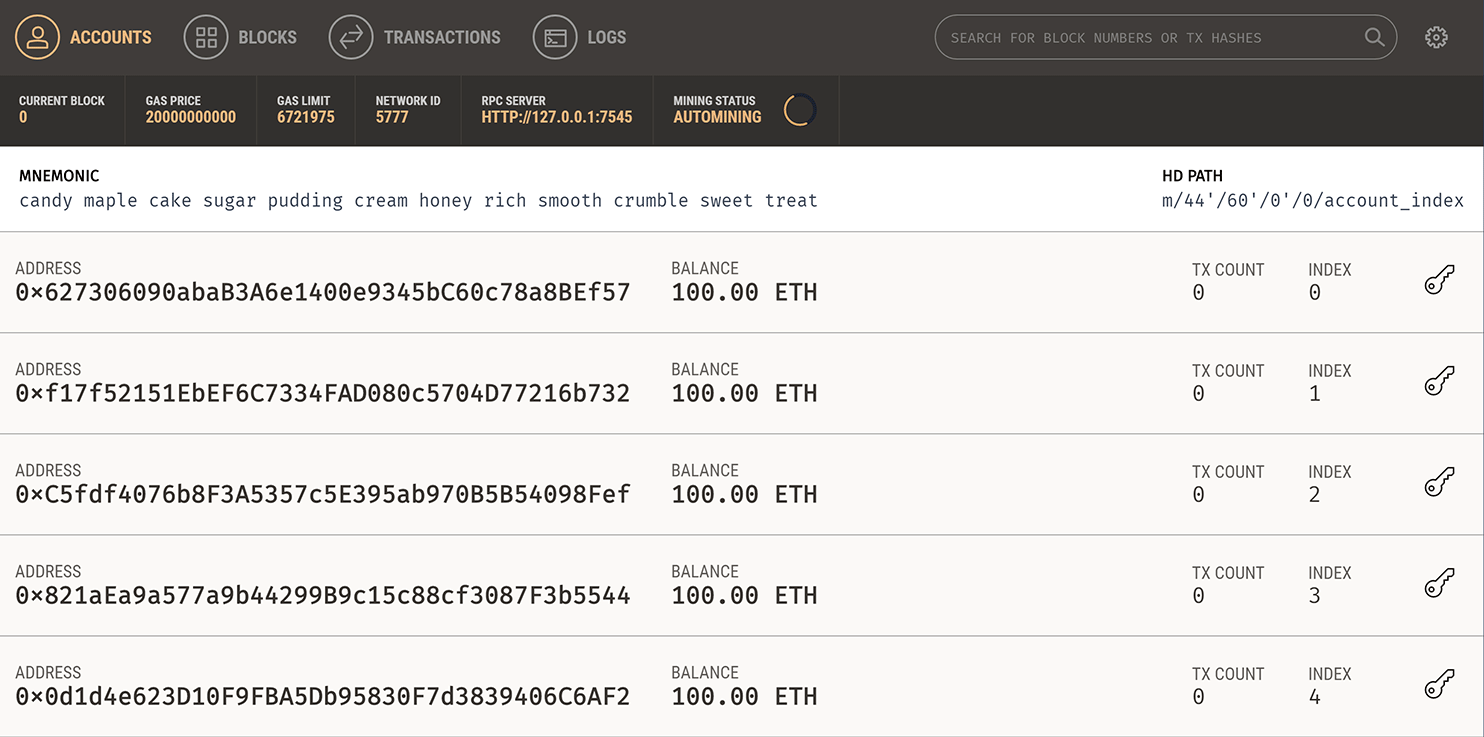
**Note**: Read more about migrations in the [Truffle documentation](https://truffleframework.com/docs/getting_started/migrations).

You'll see one JavaScript file already in the migrations/ directory: 1\_initial\_migration.js. This handles deploying the Migrations.sol contract to observe subsequent smart contract migrations, and ensures we don't double-migrate unchanged contracts in the future.

Now we are ready to create our own migration script.

1. Create a new file named 2\_deploy\_contracts.js in the migrations/ directory.
2. Add the following content to the 2\_deploy\_contracts.js file:
3. var Adoption = artifacts.require("Adoption");
4. module.exports = function(deployer) {
5. deployer.deploy(Adoption);
6. };
7. Before we can migrate our contract to the blockchain, we need to have a blockchain running. For this tutorial, we're going to use [Ganache](https://truffleframework.com/ganache), a personal blockchain for Ethereum development you can use to deploy contracts, develop applications, and run tests. If you haven't already, [download Ganache](https://truffleframework.com/ganache) and double click the icon to launch the application. This will generate a blockchain running locally on port 7545.

**Note**: Read more about Ganache in the [Truffle documentation](https://truffleframework.com/docs/ganache/using).



1. Back in our terminal, migrate the contract to the blockchain.

truffle migrate

You should see output similar to the following:

Using network 'development'.

Running migration: 1\_initial\_migration.js

Deploying Migrations...

... 0xcc1a5aea7c0a8257ba3ae366b83af2d257d73a5772e84393b0576065bf24aedf

Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0

Saving successful migration to network...

... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956

Saving artifacts...

Running migration: 2\_deploy\_contracts.js

Deploying Adoption...

... 0x43b6a6888c90c38568d4f9ea494b9e2a22f55e506a8197938fb1bb6e5eaa5d34

Adoption: 0x345ca3e014aaf5dca488057592ee47305d9b3e10

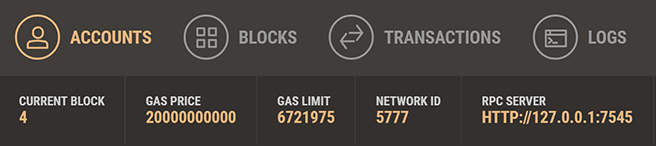
Saving successful migration to network...

... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0

Saving artifacts...

You can see the migrations being executed in order, followed by the blockchain address of each deployed contract. (Your addresses will differ.)

1. In Ganache, note that the state of the blockchain has changed. The blockchain now shows that the current block, previously 0, is now 4. In addition, while the first account originally had 100 ether, it is now lower, due to the transaction costs of migration. We'll talk more about transaction costs later.



You've now written your first smart contract and deployed it to a locally running blockchain. It's time to interact with our smart contract now to make sure it does what we want.

**5.****Testing the smart contract**

Truffle is very flexible when it comes to smart contract testing, in that tests can be written either in JavaScript or Solidity. In this tutorial, we'll be writing our tests in Solidity.

1. Create a new file named TestAdoption.sol in the test/ directory.
2. Add the following content to the TestAdoption.sol file:
3. pragma solidity ^0.4.17;
4. import "truffle/Assert.sol";
5. import "truffle/DeployedAddresses.sol";
6. import "../contracts/Adoption.sol";
7. contract TestAdoption {
8. Adoption adoption = Adoption(DeployedAddresses.Adoption());
9. }

We start the contract off with 3 imports:

* Assert.sol: Gives us various assertions to use in our tests. In testing, **an assertion checks for things like equality, inequality or emptiness to return a pass/fail** from our test. [Here's a full list of the assertions included with Truffle](https://github.com/trufflesuite/truffle-core/blob/master/lib/testing/Assert.sol).
* DeployedAddresses.sol: When running tests, Truffle will deploy a fresh instance of the contract being tested to the blockchain. This smart contract gets the address of the deployed contract.
* Adoption.sol: The smart contract we want to test.

**Note**: The first two imports are referring to global Truffle files, not a truffle directory. You should not see a truffle directory inside your test/ directory.

Then we define a contract-wide variable containing the smart contract to be tested, calling the DeployedAddresses smart contract to get its address.

**Testing the adopt() function**

To test the adopt() function, recall that upon success it returns the given petId. We can ensure an ID was returned and that it's correct by comparing the return value of adopt() to the ID we passed in.

1. Add the following function within the TestAdoption.sol smart contract, after the declaration of Adoption:
2. // Testing the adopt() function
3. function testUserCanAdoptPet() public {
4. uint returnedId = adoption.adopt(8);
5. uint expected = 8;
6. Assert.equal(returnedId, expected, "Adoption of pet ID 8 should be recorded.");
7. }

Things to notice:

* We call the smart contract we declared earlier with the ID of 8.
* We then declare an expected value of 8 as well.
* Finally, we pass the actual value, the expected value and a failure message (which gets printed to the console if the test does not pass) to Assert.equal().

**Testing retrieval of a single pet's owner**

Remembering from above that public variables have automatic getter methods, we can retrieve the address stored by our adoption test above. Stored data will persist for the duration of our tests, so our adoption of pet 8 above can be retrieved by other tests.

1. Add this function below the previously added function in TestAdoption.sol.
2. // Testing retrieval of a single pet's owner
3. function testGetAdopterAddressByPetId() public {
4. // Expected owner is this contract
5. address expected = this;
6. address adopter = adoption.adopters(8);
7. Assert.equal(adopter, expected, "Owner of pet ID 8 should be recorded.");
8. }

Since the TestAdoption contract will be sending the transaction, we set the expected value to **this**, a contract-wide variable that gets the current contract's address. From there we assert equality as we did above.

**Testing retrieval of all pet owners**

Since arrays can only return a single value given a single key, we create our own getter for the entire array.

1. Add this function below the previously added function in TestAdoption.sol.
2. // Testing retrieval of all pet owners
3. function testGetAdopterAddressByPetIdInArray() public {
4. // Expected owner is this contract
5. address expected = this;
6. // Store adopters in memory rather than contract's storage
7. address[16] memory adopters = adoption.getAdopters();
8. Assert.equal(adopters[8], expected, "Owner of pet ID 8 should be recorded.");
9. }

Note the **memory** attribute on adopters. The memory attribute tells Solidity to temporarily store the value in memory, rather than saving it to the contract's storage. Since adopters is an array, and we know from the first adoption test that we adopted pet 8, we compare the testing contracts address with location 8 in the array.

**Running the tests**

1. Back in the terminal, run the tests:
2. truffle test
3. If all the tests pass, you'll see console output similar to this:

Using network 'development'.

Compiling ./contracts/Adoption.sol...

Compiling ./test/TestAdoption.sol...

Compiling truffle/Assert.sol...

Compiling truffle/DeployedAddresses.sol...

TestAdoption

✓ testUserCanAdoptPet (91ms)

✓ testGetAdopterAddressByPetId (70ms)

✓ testGetAdopterAddressByPetIdInArray (89ms)

3 passing (670ms)

**6.****Creating a user interface to interact with the smart contract**

Now that we've created the smart contract, deployed it to our local test blockchain and confirmed we can interact with it via the console, it's time to create a UI so that Pete has something to use for his pet shop!

Included with the pet-shop Truffle Box was code for the app's front-end. That code exists within the src/ directory.

The front-end doesn't use a build system (webpack, grunt, etc.) to be as easy as possible to get started. The structure of the app is already there; we'll be filling in the functions which are unique to Ethereum. This way, you can take this knowledge and apply it to your own front-end development.

**Instantiating web3**

1. Open /src/js/app.js in a text editor.
2. Examine the file. Note that there is a global App object to manage our application, load in the pet data in init() and then call the function initWeb3(). The [web3 JavaScript library](https://github.com/ethereum/web3.js/) interacts with the Ethereum blockchain. It can retrieve user accounts, send transactions, interact with smart contracts, and more.
3. Remove the multi-line comment from within initWeb3 and replace it with the following:
4. // Is there an injected web3 instance?
5. if (typeof web3 !== 'undefined') {
6. App.web3Provider = web3.currentProvider;
7. } else {
8. // If no injected web3 instance is detected, fall back to Ganache
9. App.web3Provider = new Web3.providers.HttpProvider('http://localhost:7545');
10. }
11. web3 = new Web3(App.web3Provider);

Things to notice:

* First, we check if there's a web3 instance already active. (Ethereum browsers like [Mist](https://github.com/ethereum/mist) or Chrome with the [MetaMask](https://metamask.io/) extension will inject their own web3 instances.) If an injected web3 instance is present, we get its provider and use it to create our web3 object.
* If no injected web3 instance is present, we create our web3 object based on our local provider. (This fallback is fine for development environments, but insecure and not suitable for production.)

**7.****Instantiating the contract**

Now that we can interact with Ethereum via web3, we need to instantiate our smart contract so web3 knows where to find it and how it works. Truffle has a library to help with this called truffle-contract. It keeps information about the contract in sync with migrations, so you don't need to change the contract's deployed address manually.

1. Still in /src/js/app.js, remove the multi-line comment from within initContract and replace it with the following:
2. $.getJSON('Adoption.json', function(data) {
3. // Get the necessary contract artifact file and instantiate it with truffle-contract
4. var AdoptionArtifact = data;
5. App.contracts.Adoption = TruffleContract(AdoptionArtifact);
6. // Set the provider for our contract
7. App.contracts.Adoption.setProvider(App.web3Provider);
8. // Use our contract to retrieve and mark the adopted pets
9. return App.markAdopted();
10. });

Things to notice:

* We first retrieve the artifact file for our smart contract. **Artifacts are information about our contract such as its deployed address and Application Binary Interface (ABI)**. **The ABI is a JavaScript object defining how to interact with the contract including its variables, functions and their parameters.**
* Once we have the artifacts in our callback, we pass them to TruffleContract(). This creates an instance of the contract we can interact with.
* With our contract instantiated, we set its web3 provider using the App.web3Provider value we stored earlier when setting up web3.
* We then call the app's markAdopted() function in case any pets are already adopted from a previous visit. We've encapsulated this in a separate function since we'll need to update the UI any time we make a change to the smart contract's data.

**Getting The Adopted Pets and Updating The UI**

1. Still in /src/js/app.js, remove the multi-line comment from markAdopted and replace it with the following:
2. var adoptionInstance;
3. App.contracts.Adoption.deployed().then(function(instance) {
4. adoptionInstance = instance;
5. return adoptionInstance.getAdopters.call();
6. }).then(function(adopters) {
7. for (i = 0; i < adopters.length; i++) {
8. if (adopters[i] !== '0x0000000000000000000000000000000000000000') {
9. $('.panel-pet').eq(i).find('button').text('Success').attr('disabled', true);
10. }
11. }
12. }).catch(function(err) {
13. console.log(err.message);
14. });

Things to notice:

* We access the deployed Adoption contract, then call getAdopters() on that instance.
* We first declare the variable adoptionInstance outside of the smart contract calls so we can access the instance after initially retrieving it.
* Using **call()** allows us to read data from the blockchain without having to send a full transaction, meaning we won't have to spend any ether.
* After calling getAdopters(), we then loop through all of them, checking to see if an address is stored for each pet. Since the array contains address types, Ethereum initializes the array with 16 empty addresses. This is why we check for an empty address string rather than null or other falsey value.
* Once a petId with a corresponding address is found, we disable its adopt button and change the button text to "Success", so the user gets some feedback.
* Any errors are logged to the console.

**Handling the adopt() Function**

1. Still in /src/js/app.js, remove the multi-line comment from handleAdopt and replace it with the following:
2. var adoptionInstance;
3. web3.eth.getAccounts(function(error, accounts) {
4. if (error) {
5. console.log(error);
6. }
7. var account = accounts[0];
8. App.contracts.Adoption.deployed().then(function(instance) {
9. adoptionInstance = instance;
10. // Execute adopt as a transaction by sending account
11. return adoptionInstance.adopt(petId, {from: account});
12. }).then(function(result) {
13. return App.markAdopted();
14. }).catch(function(err) {
15. console.log(err.message);
16. });
17. });

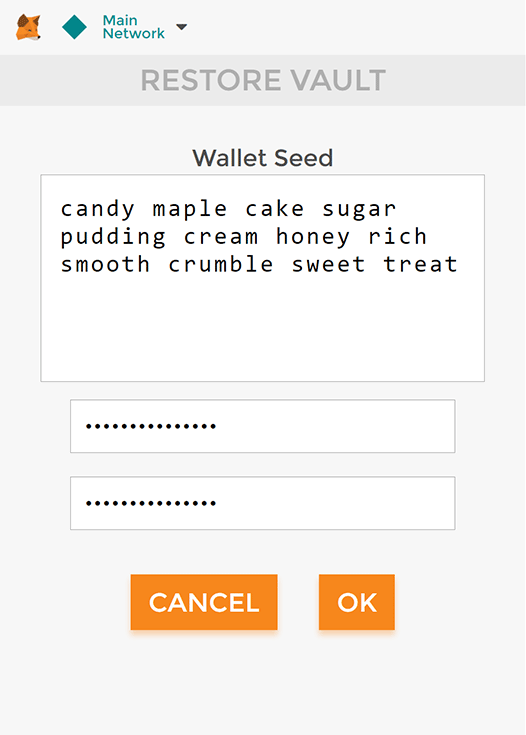
Things to notice:

* We use web3 to get the user's accounts. In the callback after an error check, we then select the first account.
* From there, we get the deployed contract as we did above and store the instance in adoptionInstance. This time though, we're going to send a **transaction** instead of a call. Transactions require a "from" address and have an associated cost. This cost, paid in ether, is called **gas**. The gas cost is the fee for performing computation and/or storing data in a smart contract. We send the transaction by executing the adopt() function with both the pet's ID and an object containing the account address, which we stored earlier in account.
* The result of sending a transaction is the transaction object. If there are no errors, we proceed to call our markAdopted() function to sync the UI with our newly stored data.

**Installing and configuring MetaMask**

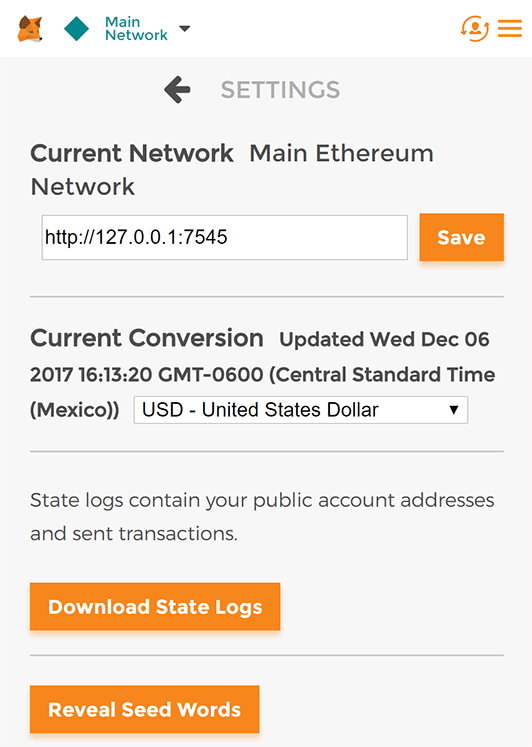
The easiest way to interact with our dapp in a browser is through [MetaMask](https://metamask.io/), a browser extension for both Chrome and Firefox.

1. Install MetaMask in your browser.
2. Once installed, you'll see the MetaMask fox icon next to your address bar. Click the icon and you'll see this screen appear:
3. Click Accept to accept the Privacy Notice.
4. Then you'll see the Terms of Use. Read them, scrolling to the bottom, and then click **Accept** there too.
5. Now you'll see the initial MetaMask screen. Click **Import Existing DEN**.
6. In the box marked **Wallet Seed**, enter the mnemonic that is displayed in Ganache.

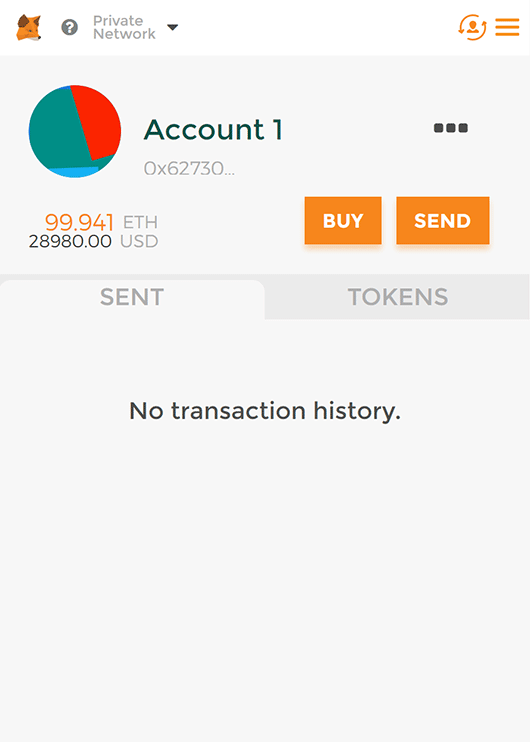
**Warning**: Do not use this mnemonic on the main Ethereum network (mainnet). If you send ETH to any account generated from this mnemonic, you will lose it all!

Enter a password below that and click **OK**.

1. Now we need to connect MetaMask to the blockchain created by Ganache. Click the menu that shows "Main Network" and select **Custom RPC**.



1. In the box titled "New RPC URL" enter http://127.0.0.1:7545 and click **Save**.

The network name at the top will switch to say "Private Network".

1. Click the left-pointing arrow next to "Settings" to close out of the page and return to the Accounts page.

Each account created by Ganache is given 100 ether. You'll notice it's slightly less on the first account because some gas was used when the contract itself was deployed and when the tests were run.

Configuration is now complete.

**Installing and configuring lite-server**

We can now start a local web server and use the dapp. We're using the lite-server library to serve our static files. This shipped with the pet-shop Truffle Box, but let's take a look at how it works.

1. Open bs-config.json in a text editor (in the project's root directory) and examine the contents:
2. {
3. "server": {
4. "baseDir": ["./src", "./build/contracts"]
5. }
6. }

This tells lite-server which files to include in our base directory. We add the ./src directory for our website files and ./build/contracts directory for the contract artifacts.

We've also added a dev command to the scripts object in the package.json file in the project's root directory. The scripts object allows us to alias console commands to a single npm command. In this case we're just doing a single command, but it's possible to have more complex configurations. Here's what yours should look like:

"scripts": {

"dev": "lite-server",

"test": "echo \"Error: no test specified\" && exit 1"

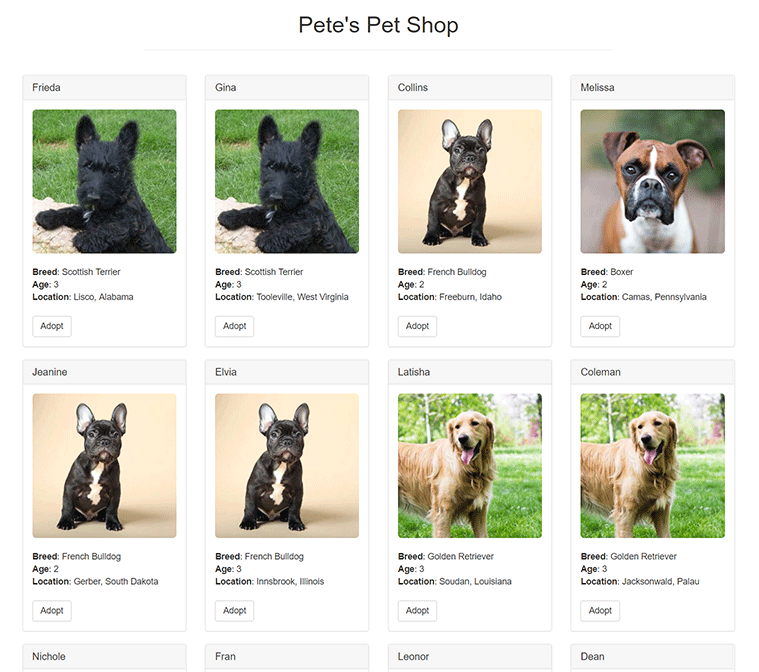
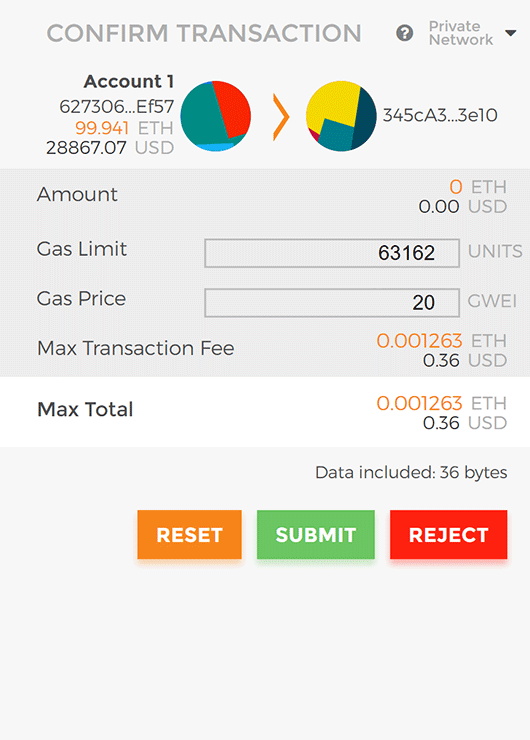
},

This tells npm to run our local install of lite-server when we execute npm run dev from the console.

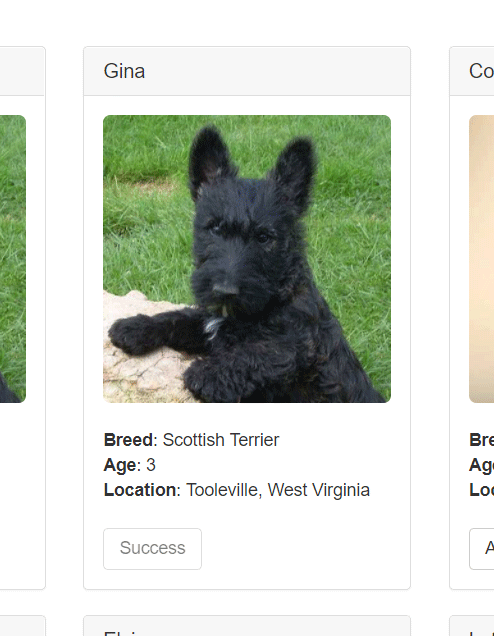
**Using the dapp**

1. Start the local web server:
2. npm run dev

The dev server will launch and automatically open a new browser tab containing your dapp.

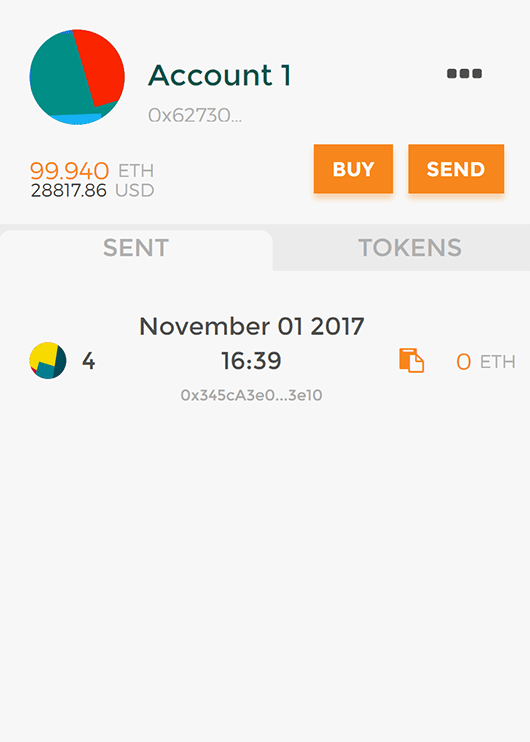


1. To use the dapp, click the **Adopt** button on the pet of your choice.
2. You'll be automatically prompted to approve the transaction by MetaMask. Click **Submit** to approve the transaction.
3. You'll see the button next to the adopted pet change to say "Success" and become disabled, just as we specified, because the pet has now been adopted.



*Adoption success*

**Note**: If the button doesn't automatically change to say "Success", refreshing the app in the browser should trigger it.

And in MetaMask, you'll see the transaction listed:

You'll also see the same transaction listed in Ganache under the "Transactions" section.

Congratulations! You have taken a huge step to becoming a full-fledged dapp developer. For developing locally, you have all the tools you need to start making more advanced dapps. If you'd like to make your dapp live for others to use, stay tuned for our future tutorial on deploying to the Ropsten testnet.