## Table of Contents

* Introduction
  + [How Does a Blockchain Application Work?](https://www.dappuniversity.com/articles/blockchain-app-tutorial#how)
  + Concepts
* Tutorial Steps

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

# **Introduction**

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.

##### First of all, we need to know some initial concepts including what is a blockchain, how does it work, what is contract and smart contract, how to connect to your own wallet with contract, and so on…

Let me first explain how a blockchain application works

### - How Does a Blockchain Application Work?

I've chosen a pet shop application for this tutorial because it is one of the most common ways to learn any new programming language. It will teach us how to read and write data from the blockchain, as well as execute business logic that will govern the behavior of our pet shot application. It will teach you the fundamentals about how a blockchain works and how to write Ethereum smart contracts.

In order to understand how a blockchain application works, let's first look at how a pet shop might work as a web application. To access the pet shop, you would use a web browser that would communicate with a web server over the Internet. The server contains all of the code and data for the pet shop.

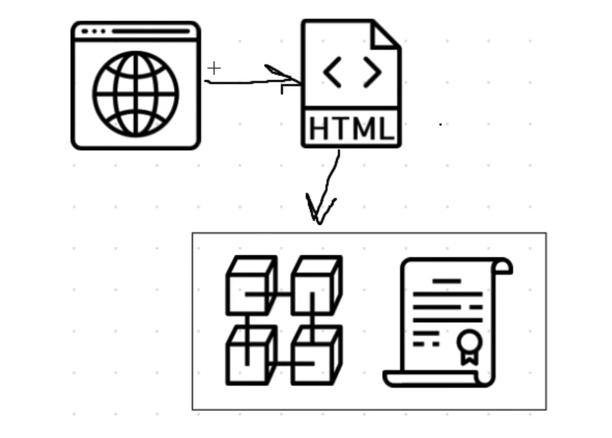
##### 

Here is a list of what you would find on the server:

* Client side files in HTML, CSS, and JavaScript
* Back end code responsible for the application's business logic
* Database that stores the tasks in the pet shop

This server is a centralized entity that full control over every aspect of the application. Anyone with full access to the server can change any part of the code or the data at any time. A blockchain application works quite differently. All of the code and the data to the pet shop does not lie on a centralized server. Instead, it is distributed across the blockchain. All of the code and the data is shared and unchangeable on the blockchain.

To illustrate this, let's examine how our blockchain-based pet shop will work.



To access the blockchain pet shop, we'll use a web browser to talk to the client side application, which will be written in HTML, CSS, and JavaScript. Instead of talking to a back end web server, the client side application will talk directly to the blockchain.

* What is a Blockchain?

A blockchain is a peer-to-peer network of computers, or nodes, that talk to one another. It's a distributed network where all of the participants share the responsibility of running the network. Each network participant maintains a copy of the code and the data on the blockchain. All of this data is contained in bundles of records called "blocks" which are "chained together" to make up the blockchain. All of the nodes on the network ensure that this data is secure and unchangeable, unlike a centralized application where the code and data can be changed at any time. That's what makes the blockchain so powerful! Because the blockchain is responsible for storing data, it fundamentally is a database. And because it's a network of computers that talk to one another, it's a network. You can think of it as a network and a database all in one.

I should also highlight another fundamental distinction between traditional web applications and blockchain applications: instead of being a user of the application itself, you are a user of the blockhain network. The application does not manage any user data. That is the responsibility of the blockchain!

- What is Smart Contract?

All of the code on the blockchain is contained in smart contracts, which are programs that run on the blockchain. They are the building blocks of blockchain applications. I'll write a smart contract in this tutorial to power our pet shop. It will be responsible for fetching all of the tasks in our pet shop from the blockchain, adding new tasks, and completing tasks.

Smart contracts are written in a programming language called Solidity, which looks a lot like JavaScript. All of the code in the smart contract is immutable, or unchangeable. Once we deploy the smart contract to the blockchain, we won't be able to change or update any of the code. This is a design feature that ensures that the code is trustless and secure. I often compare smart contracts to microservices on the web. They act as an interface for reading and writing data from the blockchain, as well as executing business logic. They're publicly accessible, meaning anyone with access to the blockchain can access their interface.

* What is Solidity?

Solidity is a contract-oriented, high-level programming language for implementing smart contracts. Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the Ethereum Virtual Machine (EVM).

* What is Ethereum?

Ethereum is a decentralized ie. blockchain platform that runs smart contracts i.e. applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference.

* The Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine, also known as EVM, is the runtime environment for smart contracts in Ethereum. The Ethereum Virtual Machine focuses on providing security and executing untrusted code by computers all over the world.

The EVM specialised in preventing Denial-of-service attacks and ensures that programs do not have access to each other's state, ensuring communication can be established without any potential interference.

The Ethereum Virtual Machine has been designed to serve as a runtime environment for smart contracts based on Ethereum.

- Truffle

A world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier.

- dApp(Decentralized Applications)

Decentralized applications—also known as "dApps" or "dapps"—are digital applications that run on a blockchain network of computers instead of relying on a single computer. Because dApps are decentralized, they are free from the control and interference of a single authority.

- MetaMask

MetaMask is a software cryptocurrency wallet used to interact with the Ethereum blockchain. It allows users to access their Ethereum wallet through a browser extension or mobile app, which can then be used to interact with decentralized applications.

# **Tutorial Steps**

I am gonna explain you how to write the smart contract and front-end logic for its usage.

Tutorial Steps:

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

## 1. Setting up the development environment

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

* [Node.js v8+ 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://trufflesuite.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 https://trufflesuite.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.

mkdir pet-shop-tutorial

cd pet-shop-tutorial

2. We've created a special [Truffle Box](https://trufflesuite.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.

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://trufflesuite.com/docs/truffle/getting-started/creating-a-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-config.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:

pragma solidity ^0.5.0;

contract Adoption {

}

Things to notice:

* The minimum version of Solidity required is noted at the top of the contract: pragma solidity ^0.5.0;. 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.

### **Your first function: Adopting a pet**[**¶**](https://trufflesuite.com/tutorial/index.html#your-first-function-adopting-a-pet)

Let's allow users to make adoption requests.

Add the following function to the smart contract after the variable declaration we set up above.

// Adopting a pet

function adopt(uint petId) public returns (uint) {

require(petId >= 0 && petId <= 15);

adopters[petId] = msg.sender;

return petId;

}

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**[**¶**](https://trufflesuite.com/tutorial/index.html#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.

Add the following getAdopters() function to the smart contract, after the adopt() function we added above:

// Retrieving the adopters

function getAdopters() public view returns (address[16] memory) {

return adopters;

}

Things to notice:

* 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] memory. memory gives the data location for the variable.
* The view keyword in the function declaration means that the function will not modify the state of the contract. Further information about the exact limits imposed by view is available [here](https://solidity.readthedocs.io/en/latest/contracts.html#view-functions).

## 4. Compiling and migrating the smart contract[¶](https://trufflesuite.com/tutorial/index.html#compiling-and-migrating-the-smart-contract)

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

### **Compilation**[**¶**](https://trufflesuite.com/tutorial/index.html#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.

In a terminal, make sure you are in the root of the directory that contains the dapp and type:

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://trufflesuite.com/docs/truffle/reference/configuration#resolving-naming-conflicts-on-windows)*.*

You should see output similar to the following:

Compiling your contracts...

===========================

> Compiling ./contracts/Adoption.sol

> Compiling ./contracts/Migrations.sol

> Artifacts written to /Users/cruzmolina/Code/truffle-projects/metacoin/build/contracts

> Compiled successfully using:

- solc: 0.5.0+commit.1d4f565a.Emscripten.clang

### **Migration**[**¶**](https://trufflesuite.com/tutorial/index.html#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://trufflesuite.com/docs/truffle/getting-started/running-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:

var Adoption = artifacts.require("Adoption");

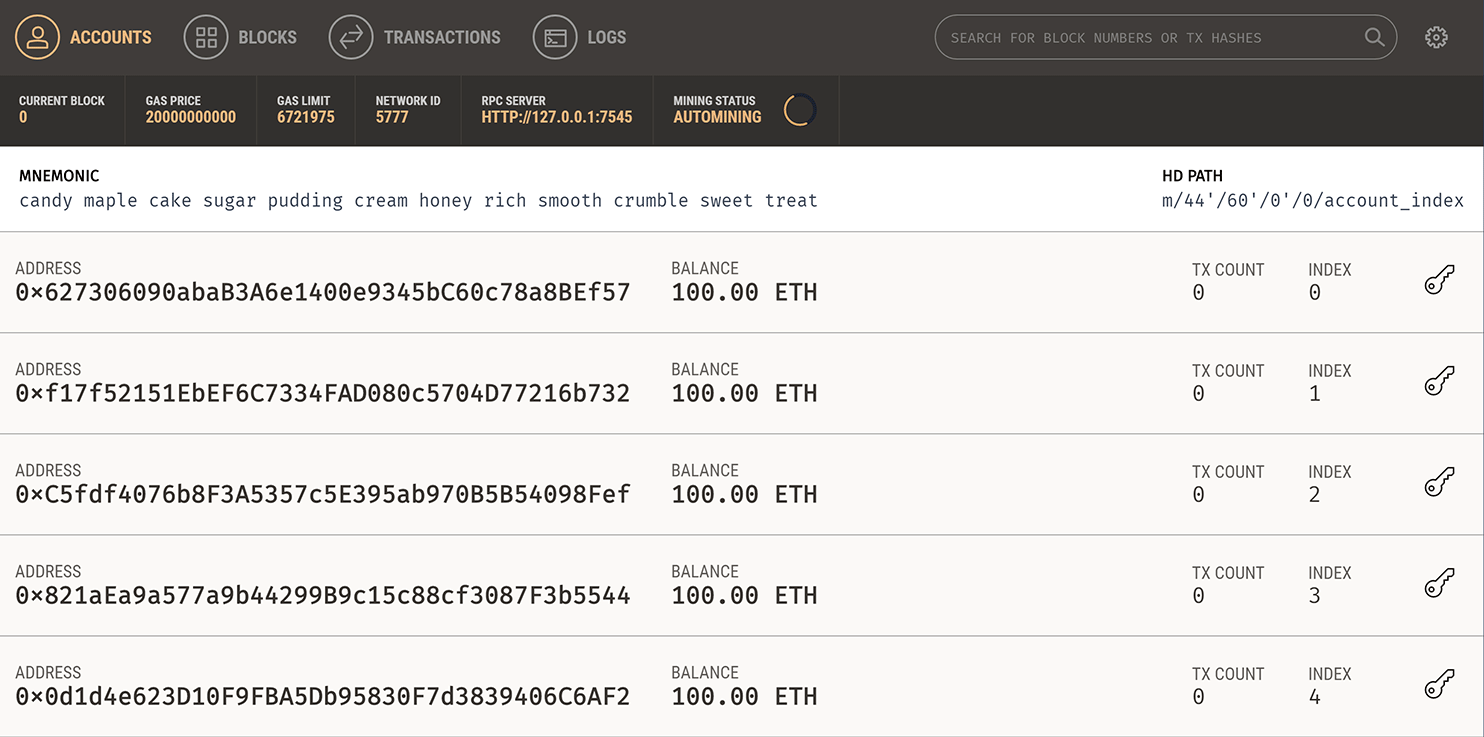
module.exports = function(deployer) {

deployer.deploy(Adoption);

};

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://trufflesuite.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://trufflesuite.com/ganache) and double click the icon to launch the application. This will generate a blockchain running locally on port 7545.

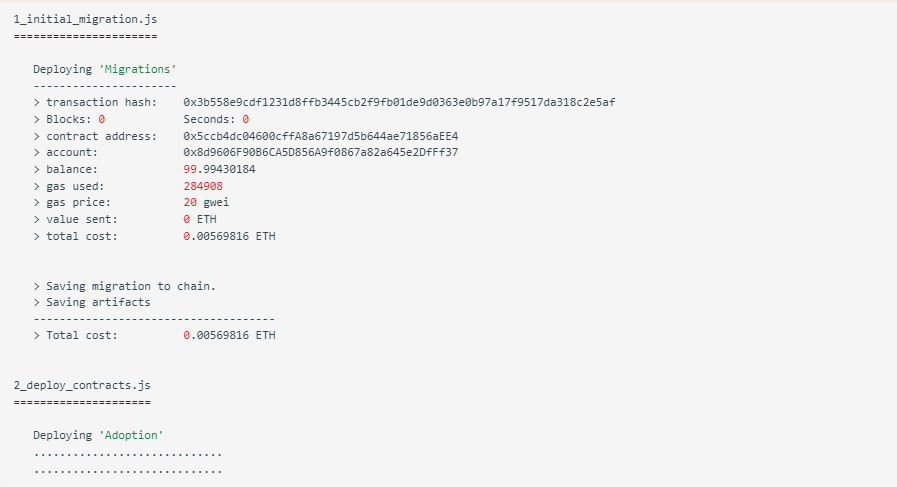
**Note**:Read more about Ganache in the [Ganache documentation](https://trufflesuite.com/docs/ganache/quickstart).



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

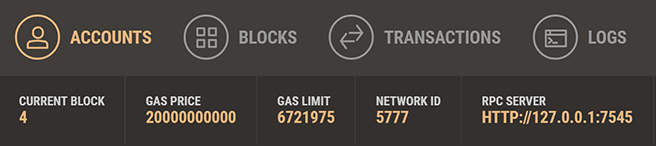
truffle migrate

You should see output similar to the following:



You can see the migrations being executed in order, followed by some information related to each migration. (Your information will differ.)

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. Creating a user interface to interact with the smart contract[¶](https://trufflesuite.com/tutorial/index.html#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**[**¶**](https://trufflesuite.com/tutorial/index.html#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:



Things to notice:

* First, we check if we are using modern dapp browsers or the more recent versions of [MetaMask](https://github.com/MetaMask) where an ethereum provider is injected into the window object. If so, we use it to create our web3 object, but we also need to explicitly request access to the accounts with ethereum.enable().
* If the ethereum object does not exist, we then check for an injected web3 instance. If it exists, this indicates that we are using an older dapp browser (like [Mist](https://github.com/ethereum/mist) or an older version of MetaMask). If so, 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.)

### **Instantiating the contract**[**¶**](https://trufflesuite.com/tutorial/index.html#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.

Still in /src/js/app.js, remove the multi-line comment from within initContract and replace it with the following:



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.

### **Handling the adopt() Function**[**¶**](https://trufflesuite.com/tutorial/index.html#handling-the-adopt-function)

Still in /src/js/app.js, remove the multi-line comment from handleAdopt and replace it with the following:



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.

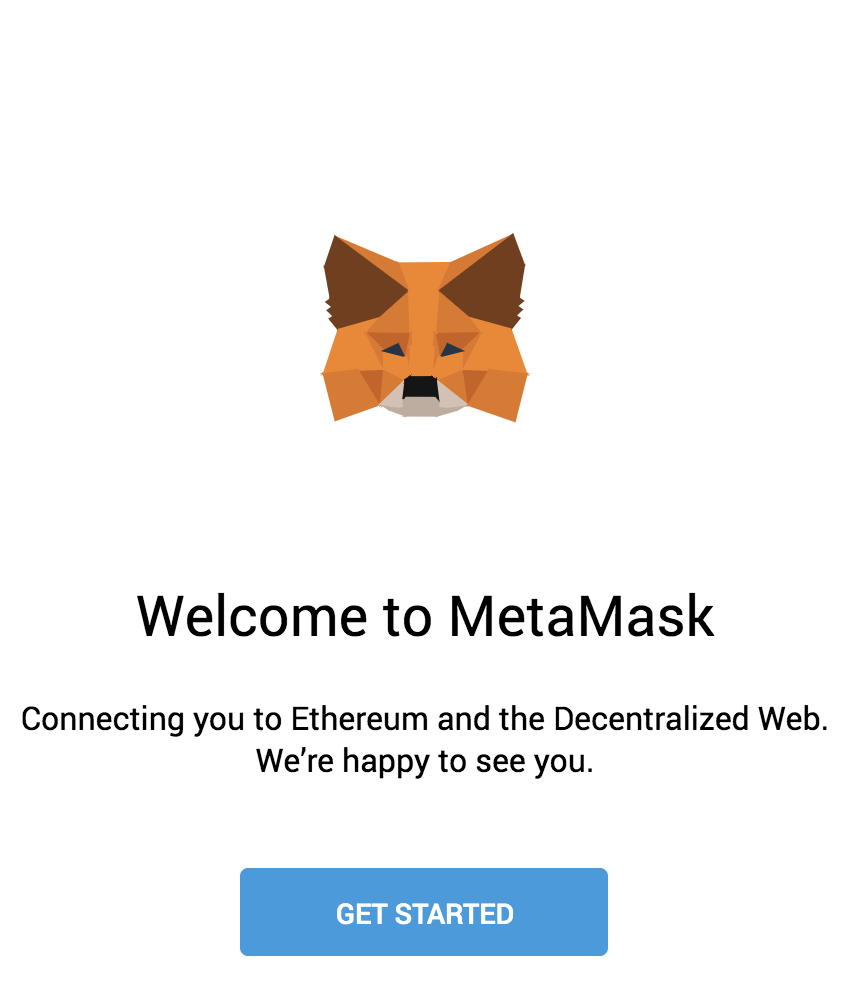
## 6. Interacting with the dapp in a browser[¶](https://trufflesuite.com/tutorial/index.html#interacting-with-the-dapp-in-a-browser)

Now we're ready to use our dapp!

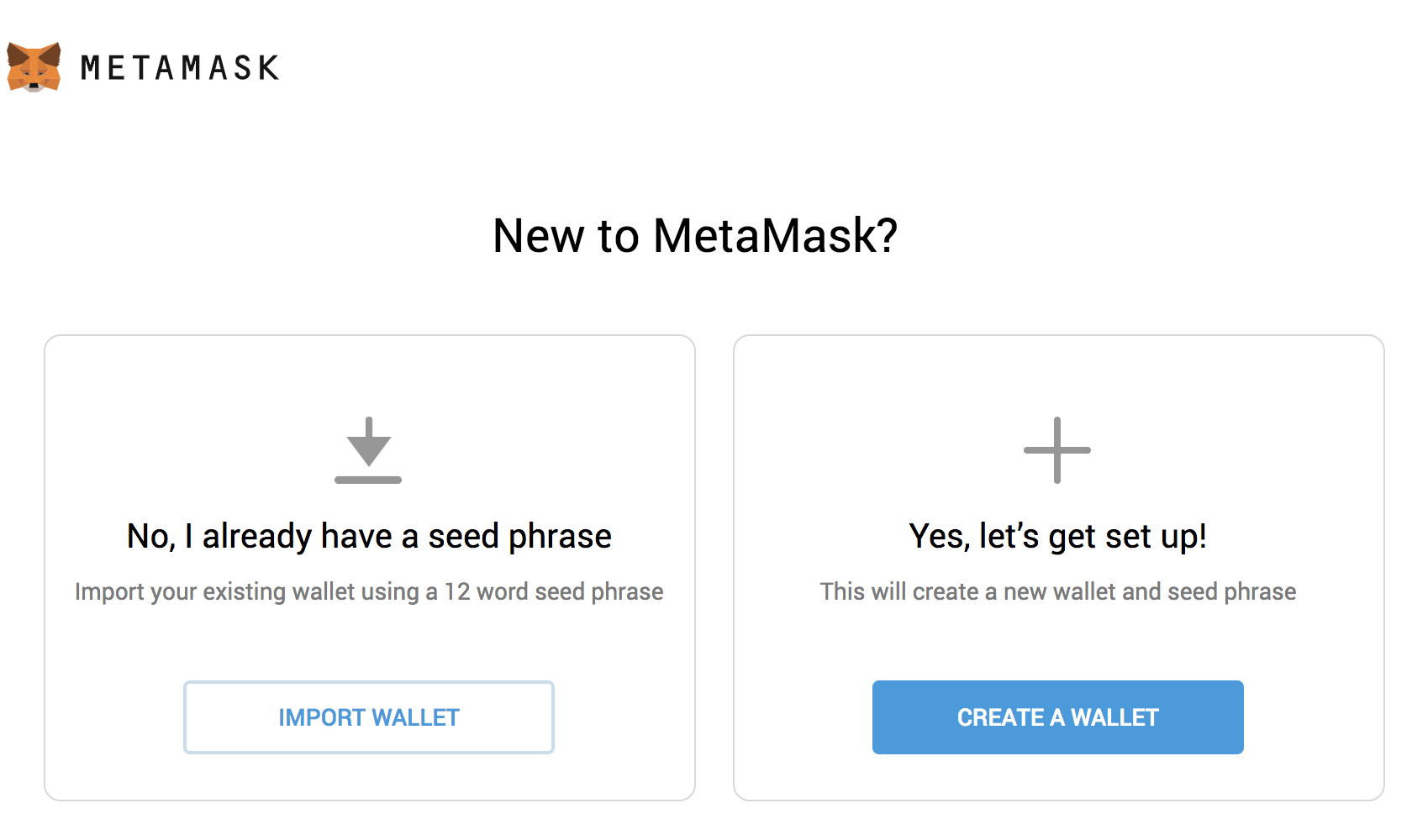
### **Installing and configuring MetaMask[¶](https://trufflesuite.com/tutorial/index.html" \l "installing-and-configuring-metamask" \o "Permanent link)**

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.

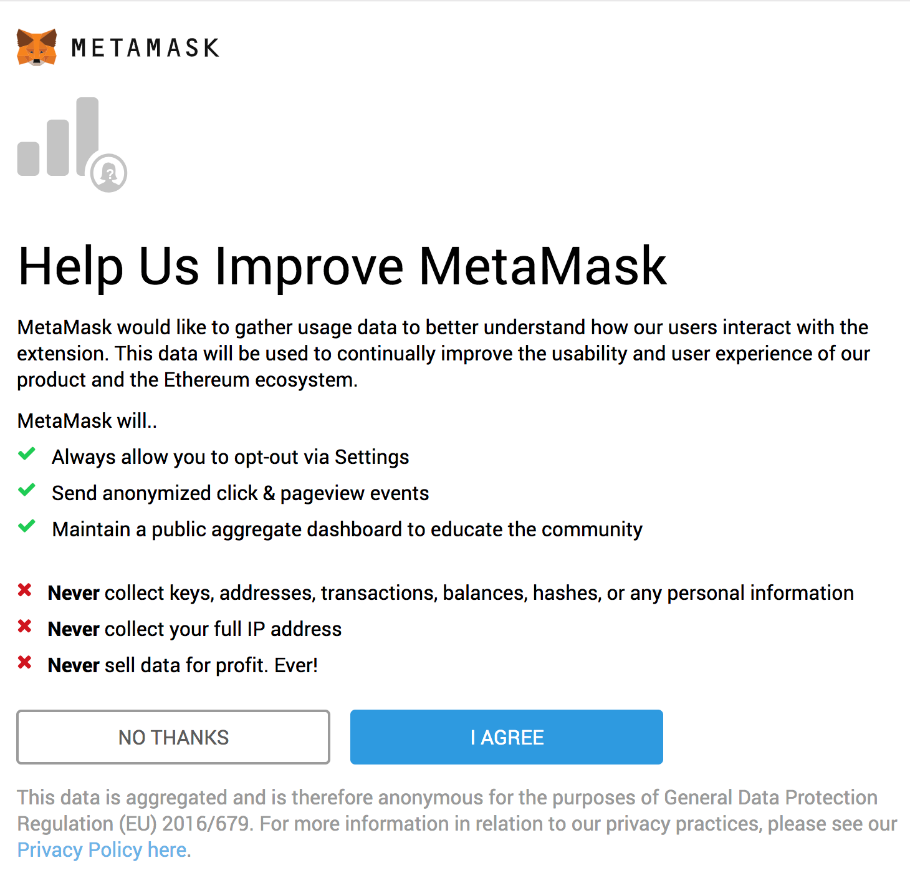
1. Install MetaMask in your browser.
2. Once installed, a tab in your browser should open displaying the following:



After clicking Getting Started, you should see the initial MetaMask screen. Click Import Wallet.



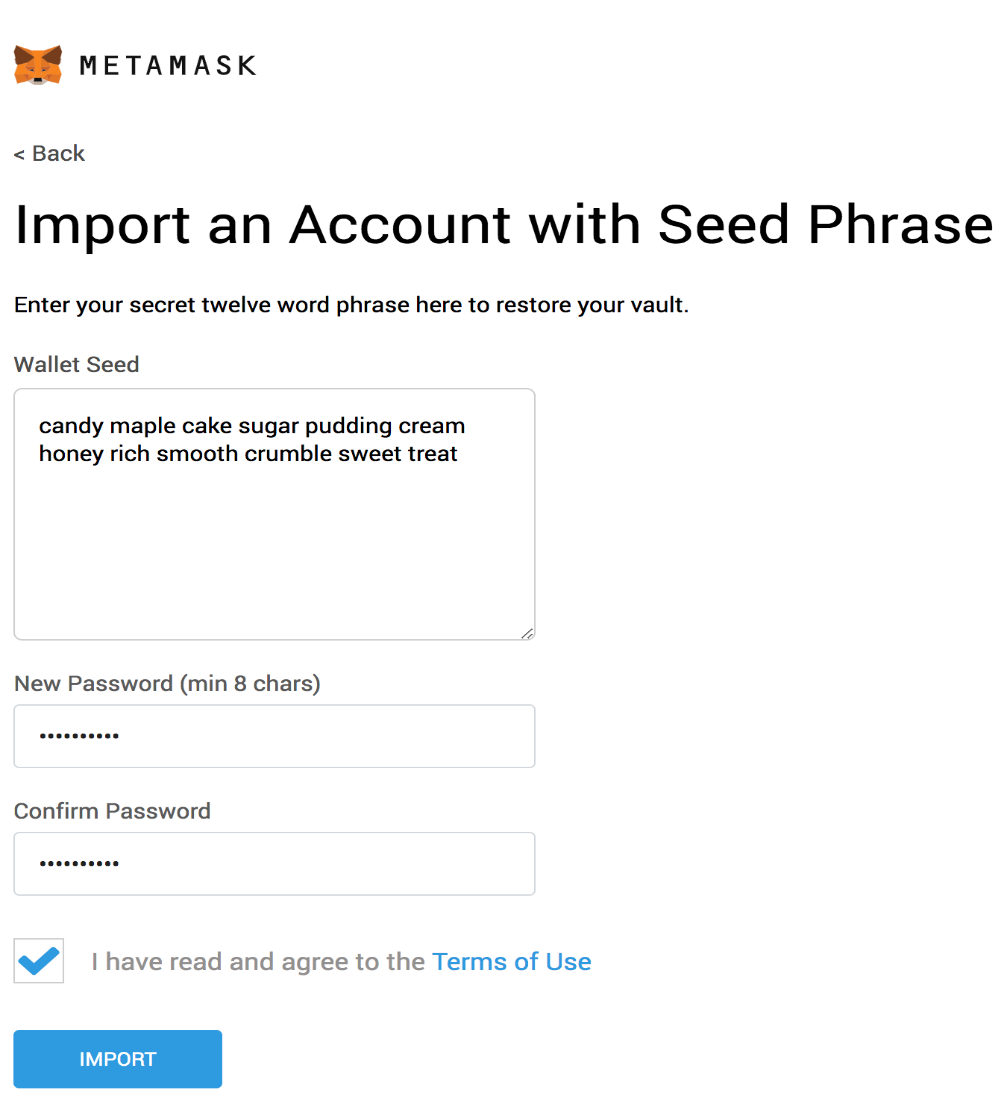
Next, you should see a screen requesting anonymous analytics. Choose to decline or agree.



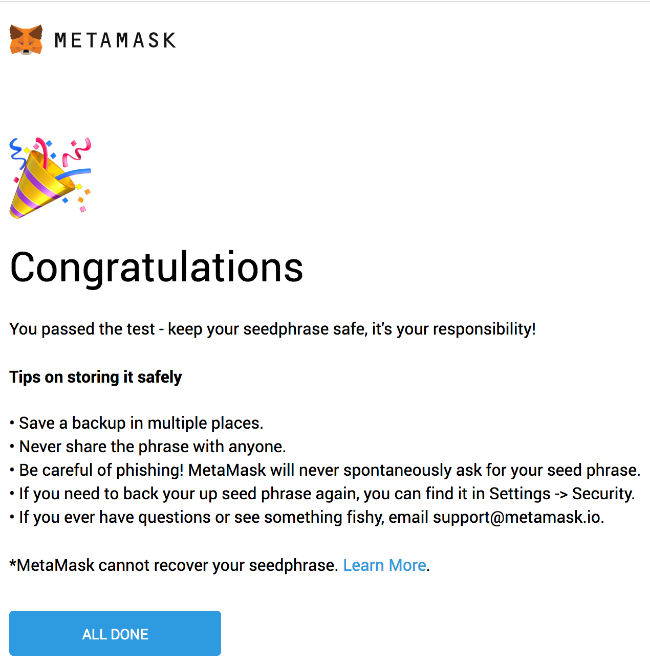
In the box marked Wallet Seed, enter the mnemonic that is displayed in Ganache.

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!

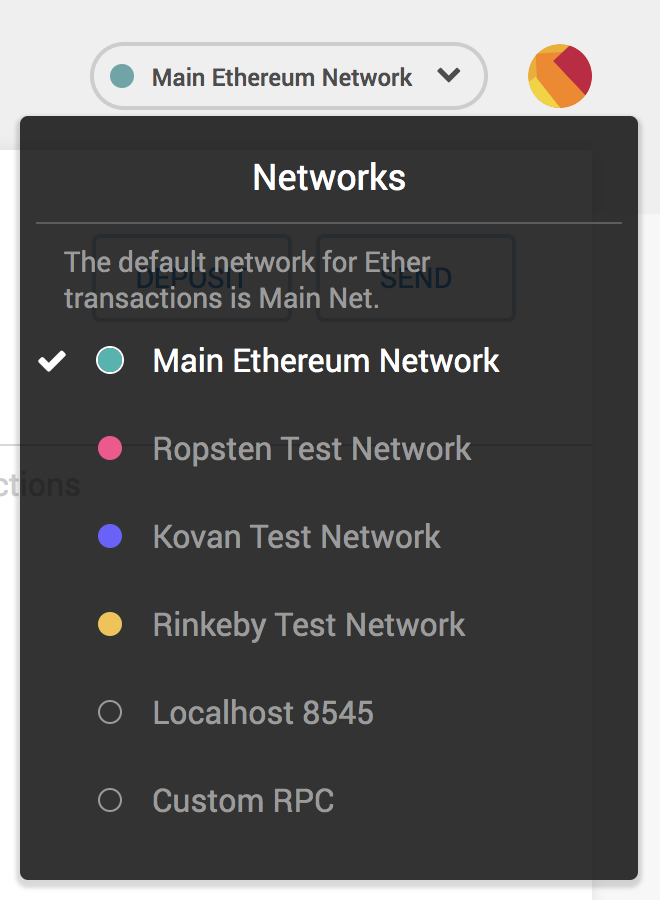
Enter a password below that and click **OK**.



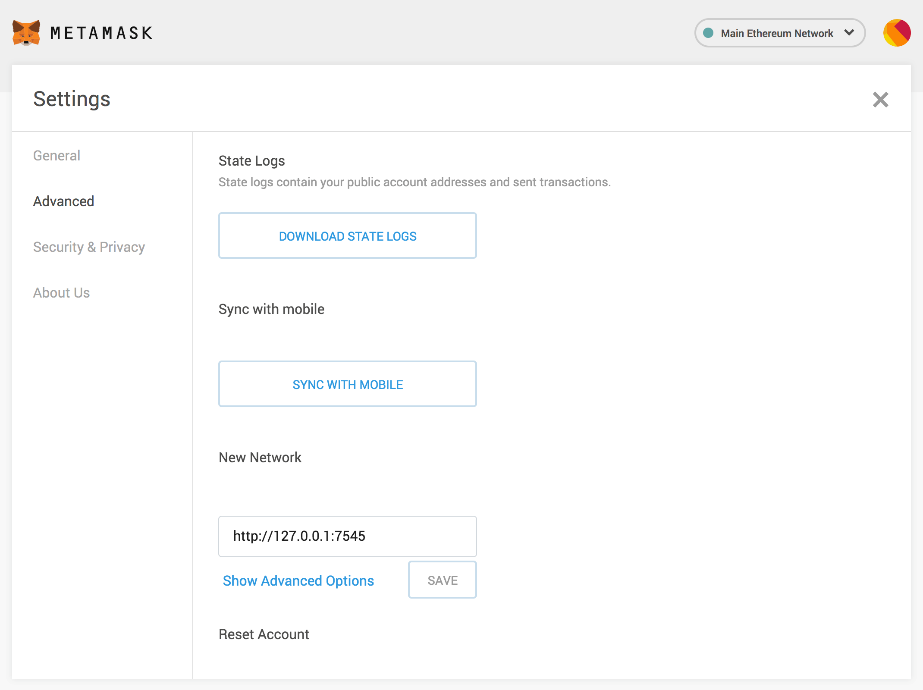
If all goes well, MetaMask should display the following screen. Click **All Done**.



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



In the box titled "New Network" enter http://127.0.0.1:7545, in the box titled "Chain ID" enter 1337 (Default Chain ID for Ganache) and click Save.

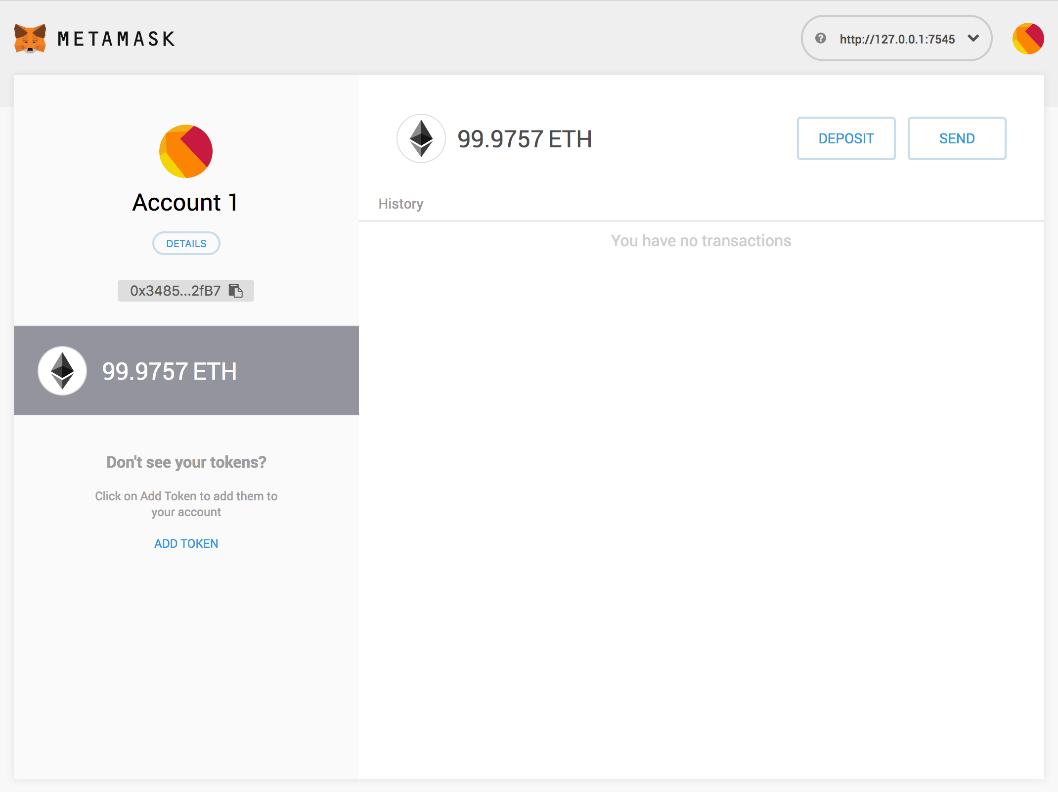


The network name at the top will switch to say <http://127.0.0.1:7545>.

The network name at the top will switch to say http://127.0.0.1:7545.

Click the top-right X to close out of Settings 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**[**¶**](https://trufflesuite.com/tutorial/index.html#installing-and-configuring-lite-server)

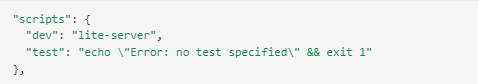
We can now start a local web server and use the dapp. We're using the lite-server library to serve our static files. This shipped with the pet-shop Truffle Box, but let's take a look at how it works.

Open bs-config.json in a text editor (in the project's root directory) and examine the contents:



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:



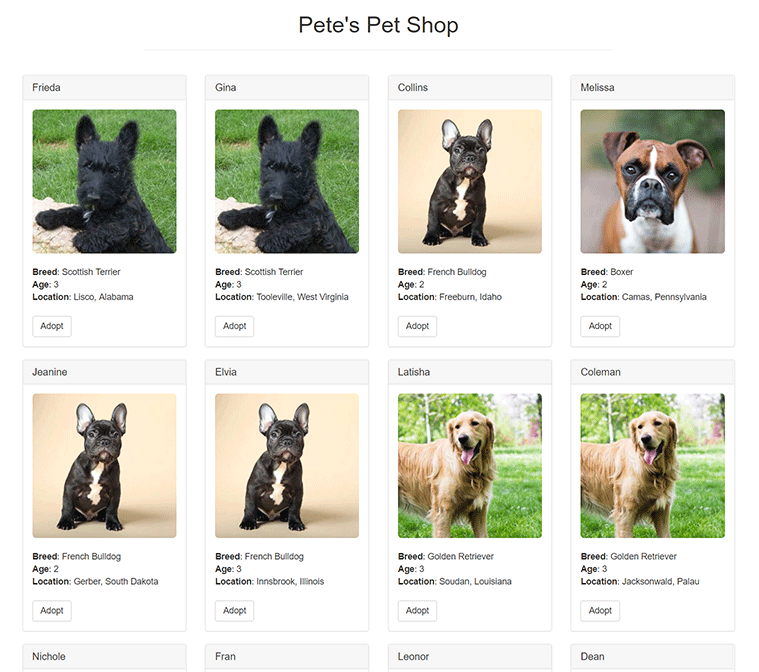
This tells npm to run our local install of lite-server when we execute npm run dev from the console.

### **Using the dapp[¶](https://trufflesuite.com/tutorial/index.html" \l "using-the-dapp" \o "Permanent link)**

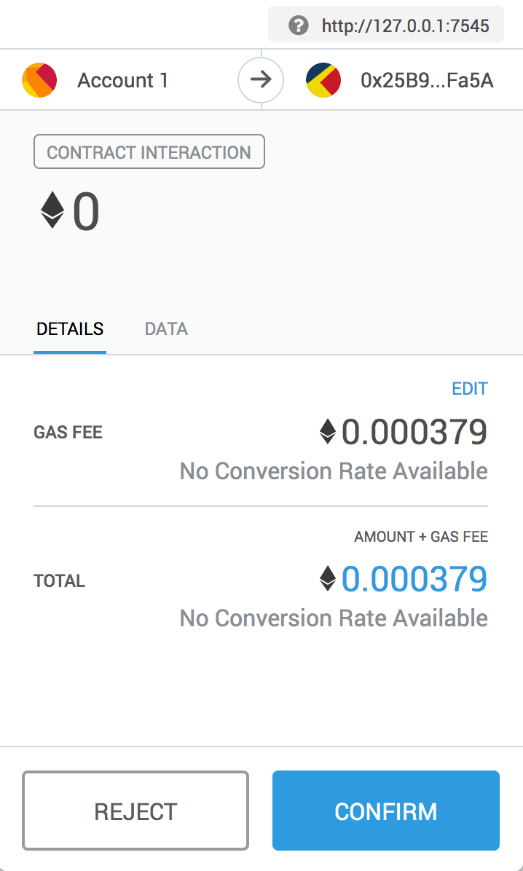
Start the local web server:

npm run dev

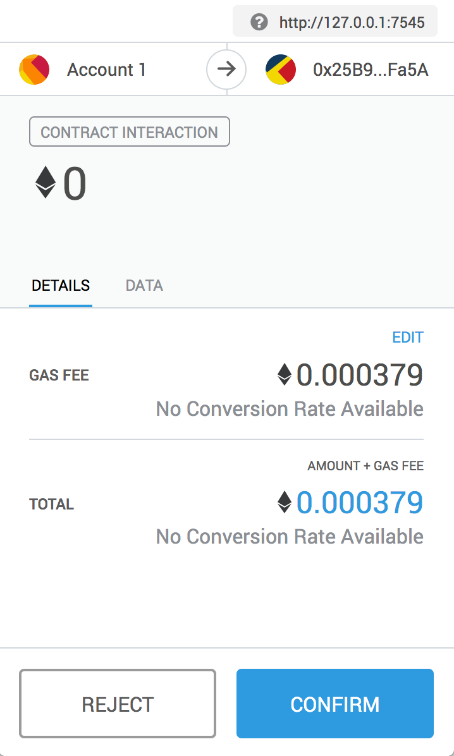
The dev server will launch and automatically open a new browser tab containing your dapp.



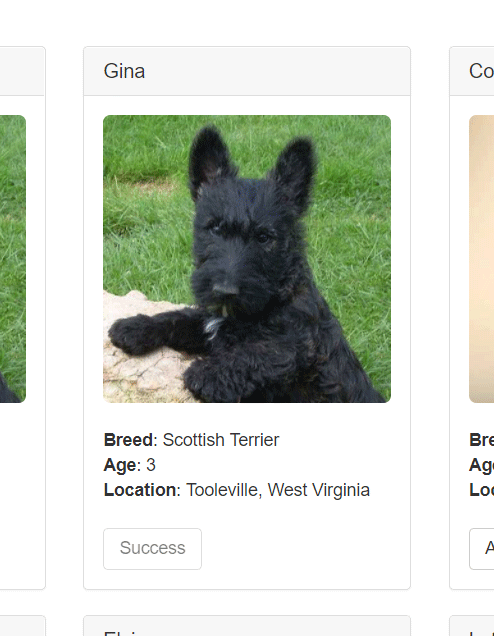
A MetaMask pop-up should appear requesting your approval to allow Pete's Pet Shop to connect to your MetaMask wallet. Without explicit approval, you will be unable to interact with the dapp. Click **Connect**.



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.

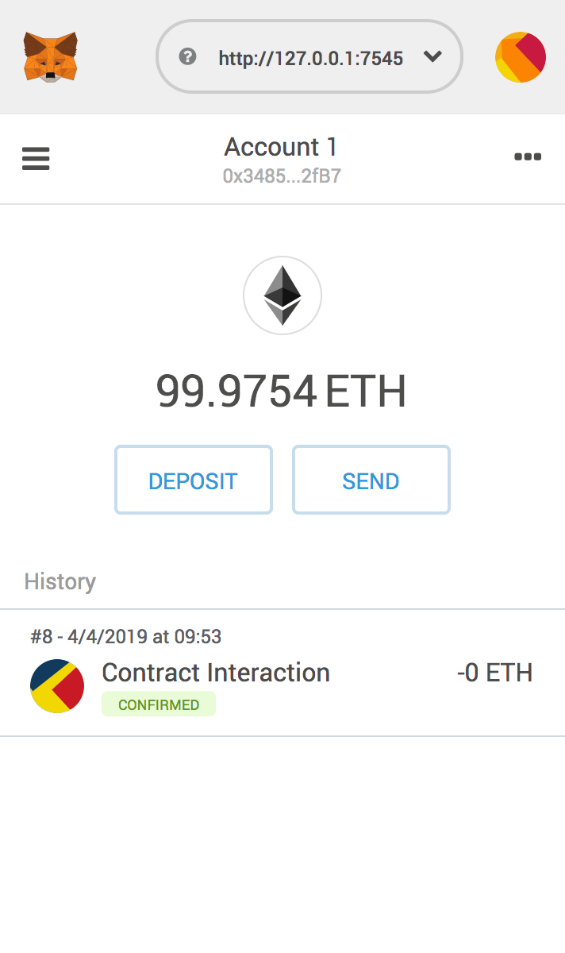


You'll see the button next to the adopted pet change to say "Success" and become disabled, just as we specified, because the pet has now been adopted.



**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.