Steps:

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

## Setting up the development environment:

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**

truffle version

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://archive.trufflesuite.com/ganache), a personal blockchain for Ethereum development you can use to deploy contracts, develop applications, and run tests.

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**

1. We've created a special [Truffle Box](https://archive.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.

**truffle init**

Directory structure

The default Truffle directory structure contains the following:

* Contracts:
  + Contains the Solidity source files for our smart contracts.
  + There is an important contract in here called Migrations.sol.
* 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

1. Writing the smart contract
2. Write the smart contract that acts as the **back-end logic and storage.**
3. Create a new file named “Adoption.sol” in the **contracts/** directory.
4. Add the following content to the file:

**pragma solidity** ^0.5.0;

contract Adoption {

}

* 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.

**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 block chain 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.

**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.

### 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.

// 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

* 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:

// 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.

1. 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**

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

1. 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.

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");

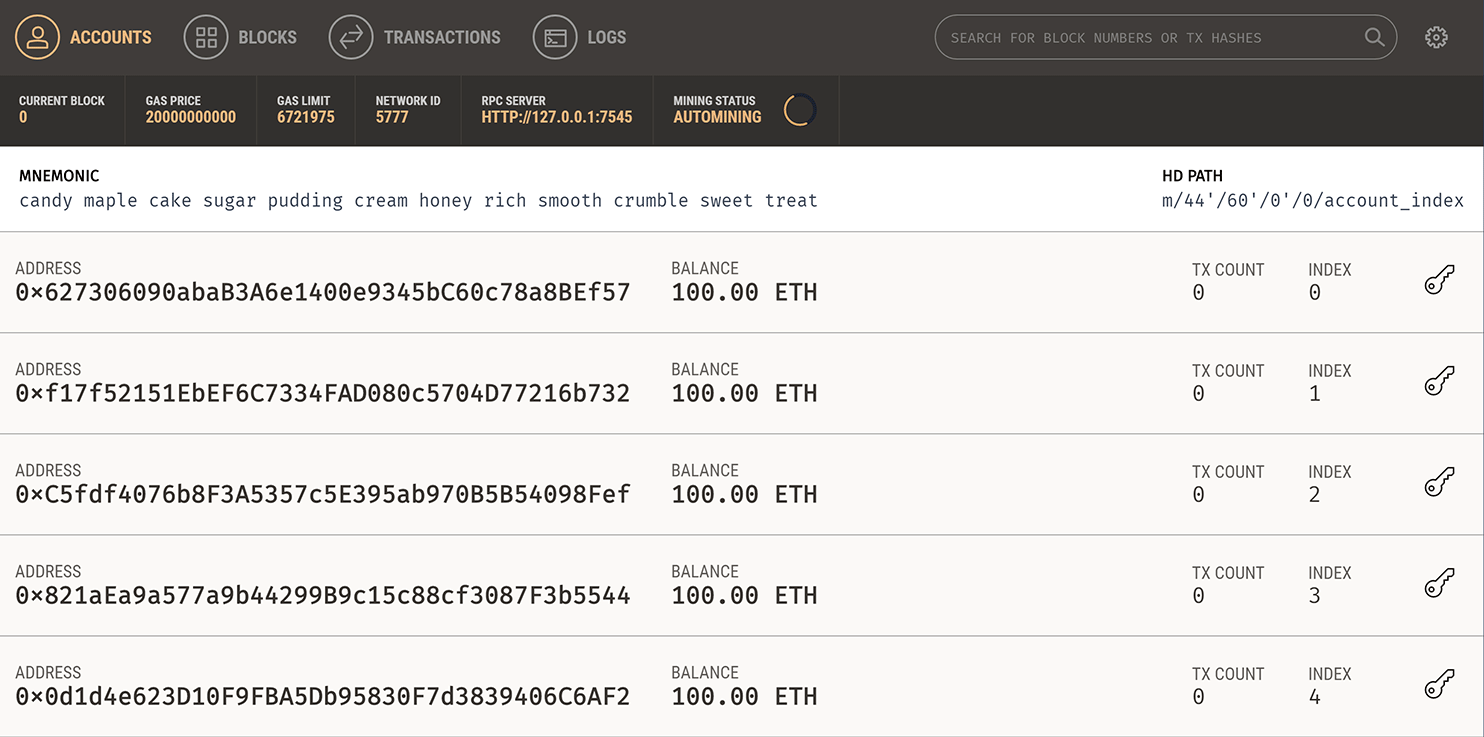
module.exports = function(deployer)

{

deployer.deploy(Adoption);

};

1. Before we can migrate our contract to the blockchain, we need to have a blockchain running.
2. For this tutorial, we're going to use [Ganache](https://archive.trufflesuite.com/ganache), a personal blockchain for Ethereum development you can use to deploy contracts, develop applications, and run tests.
3. This will generate a blockchain running locally on **port 7545**.



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

**truffle migrate**

You should see output similar to the following:

**1\_initial\_migration.js**

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

Deploying 'Migrations'

----------------------

> transaction hash: 0x3b558e9cdf1231d8ffb3445cb2f9fb01de9d0363e0b97a17f9517da318c2e5af

> Blocks: 0 Seconds: 0

> contract address: 0x5ccb4dc04600cffA8a67197d5b644ae71856aEE4

> account: 0x8d9606F90B6CA5D856A9f0867a82a645e2DfFf37

> balance: 99.99430184

> gas used: 284908

> gas price: 20 gwei

> value sent: 0 ETH

> total cost: 0.00569816 ETH

> Saving migration to chain.

> Saving artifacts

-------------------------------------

> Total cost: 0.00569816 ETH

2\_deploy\_contracts.js

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

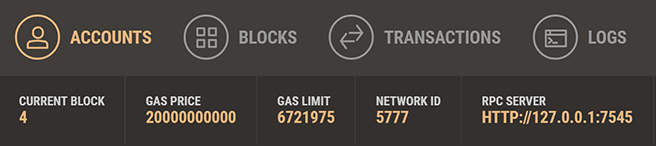
Deploying 'Adoption'

.............................

.............................

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.



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.

1. Testing the smart contract using Solidity

Truffle is very flexible when it comes to smart contract testing, in that tests can be written either in JavaScript or Solidity.

* 1. Create a new file named **TestAdoption.sol** in the **test/ directory**.
  2. Add the following content to the TestAdoption.sol file:

pragma solidity ^0.5.0;

import "truffle/Assert.sol";

import "truffle/DeployedAddresses.sol";

import "../contracts/Adoption.sol";

contract TestAdoption

{

// The address of the adoption contract to be tested

**Adoption adoption = Adoption (DeployedAddresses.Adoption())**;

// The id of the pet that will be used for testing

**uint expectedPetId = 8;**

//The expected owner of adopted pet is this contract

**address expectedAdopter = address(this);**

}

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.
* **DeployedAddresses.sol**: When running tests, Truffle will deploy a fresh instance of the contract being tested to the block chain. This smart contract gets the address of the deployed contract.
* **Adoption**: The smart contract we want to test.

**Note**: The first two imports are referring to global Truffle files, not a `truffle` directory.

Then we define three contract-wide variables:

* First, one containing the smart contract to be tested, calling the DeployedAddresses smart contract to get its address.
* Second, the id of the pet that will be used to test the adoption functions.
* Third, since the TestAdoption contract will be sending the transaction, we set the expected adopter address to **this**, a contract-wide variable that gets the current contract's 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:

// Testing the adopt() function

function testUserCanAdoptPet() public

{

uint returnedId = adoption.adopt(expectedPetId);

Assert.equal(returnedId, expectedPetId, "Adoption of the expected pet should match what is returned.");

}

Things to notice:

* We call the smart contract we declared earlier with the ID of expectedPetId.
* 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 expectedPetId above can be retrieved by other tests.

1. Add this function below the previously added function in TestAdoption.sol.

// Testing retrieval of a single pet's owner

function testGetAdopterAddressByPetId () public

{

address adopter = adoption.adopters(expectedPetId);

Assert.equal(adopter, expectedAdopter, "Owner of the expected pet should be this contract");

}

After getting the adopter address stored by the adoption contract, 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.

// Testing retrieval of all pet owners

function testGetAdopterAddressByPetIdInArray() public

{

// Store adopters in memory rather than contract's storage

address[16] memory adopters = adoption.getAdopters();

Assert.equal(adopters[expectedPetId], expectedAdopter, "Owner of the expected pet should be this contract");

}

* 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 expectedPetId, we compare the testing contracts address with location expectedPetId in the array.