

Tutorial 2

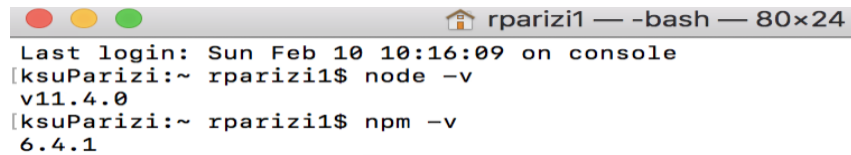
Truffle Framework Tutorial

This document contains step-by-step instructions for developing, testing, and deploying full stack Ethereum Dapps within Truffle framework (one-stop end to end development environment) using an example.

Step 1: Install required tools and dependencies

1. Install Node Package Manager (NPM), which comes with Node.js: download link: <https://nodejs.org/en/> (just update it if you'd installed before)

To check whether it is installed: `node -v` and `npm -v`



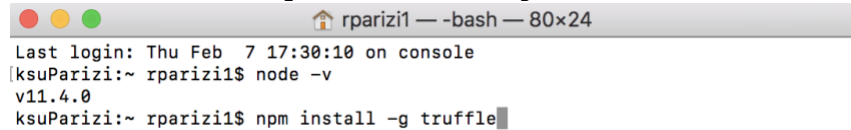
```

Last login: Sun Feb 10 10:16:09 on console
ksuParizi:~ rparizi1$ node -v
v11.4.0
ksuParizi:~ rparizi1$ npm -v
6.4.1

```

2. Install [Truffle](#)

Use this command: `npm install -g truffle`



```

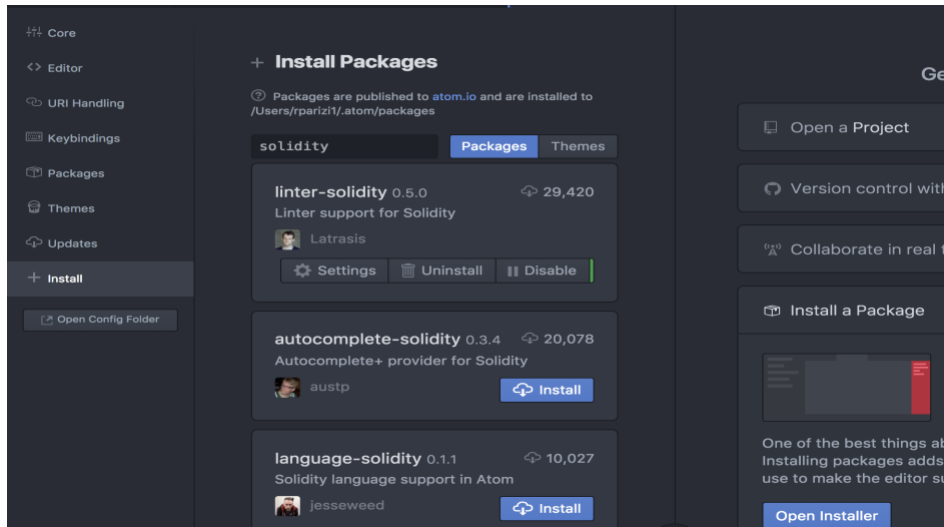
Last login: Thu Feb 7 17:30:10 on console
ksuParizi:~ rparizi1$ node -v
v11.4.0
ksuParizi:~ rparizi1$ npm install -g truffle

```

To check whether it is installed: `truffle version`

If you runs into EACCES permission errors during install, watch [this](#) or follow [this](#) to resolve it.

3. Install **Ganache**: <https://truffleframework.com/ganache>. Ganache is a personal blockchain for Ethereum development you can use to deploy contracts, develop your applications, and run tests. It is available as both a desktop application as well as a command-line tool (formerly known as the TestRPC). [note, you can create your own private chain using `geth`]
4. Install MetaMask (you've done it already!)
5. [Optional but recommended] Install editor, **Atom**: <https://atom.io/> . Once installed, open Atom> install a Package> open installer> search for “linter-solidity” package and install it, for syntax highlighting [if you have your own preferred code editor, skip this part].



Step 2: Building Backend of your Dapp

Now that we have our dependencies installed, let's start building our dApp!

We will be building a voting dApp. The dApp will talk to our smart contract on the blockchain. The application's front end will have a table of candidates that lists each candidate's id, name, and vote count. It will have a form where we can cast a vote for our desired candidate. It also shows the account we're connected to the blockchain with under "your account". Each account address will serve as a unique identifier for each voter in our dapp [a screenshot of final dapp's UI].

Election Results

#	Name	Votes
1	Candidate 1	0
2	Candidate 2	0

Select Candidate

Candidate 1

[Vote](#)

Your Account: 0x2191ef87e392377ec08e7c08eb105ef5448eced5

Now let's create a project directory/folder to hold our dApp in the command line like this [save it in a proper location]:

```
mkdir election
cd election
```

```
ksuParizi:Truffle-Projects rparizi1$ mkdir election
ksuParizi:Truffle-Projects rparizi1$ cd election
ksuParizi:election rparizi1$
```

1. Create a truffle project

You have two choices, either you can create a **bare** truffle project template, or you can use [Truffle Boxes](#), which are example applications with some starter code.

-To create a bare Truffle project, use `truffle init`

```
election — -bash — 80x24
ksuParizi:Truffle-Projects rparizi1$ mkdir election
ksuParizi:Truffle-Projects rparizi1$ cd election
ksuParizi:election rparizi1$ truffle init

✓ Preparing to download
✓ Downloading
✓ Cleaning up temporary files
✓ Setting up box

Unbox successful. Sweet!

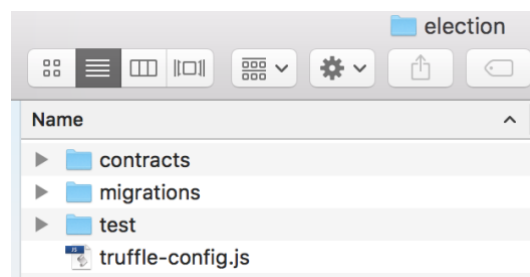
Commands:
  Compile:      truffle compile
  Migrate:      truffle migrate
  Test contracts: truffle test

ksuParizi:election rparizi1$
```

-To use Truffle boxes use the `truffle unbox <box-name>` command

Regardless, once the creation operation is completed, you'll have a project structure with the following items:

- **contracts/**: Directory for [Solidity contracts](#)
- **migrations/**: Directory for [scriptable deployment files](#)
- **test/**: Directory for test files for [testing your application and contracts](#)
- **truffle-config.js**: Truffle [configuration file](#)



In this tutorial, we'll be using the [Pet Shop box](#) —to reuse most of its code for front-end. From within your project directory, install the pet shop box from the command line:

```
truffle unbox pet-shop
```

```
ksuParizi:election rparizi1$ truffle unbox pet-shop

✓ Preparing to download
✓ Downloading
✓ Cleaning up temporary files
✓ Setting up box

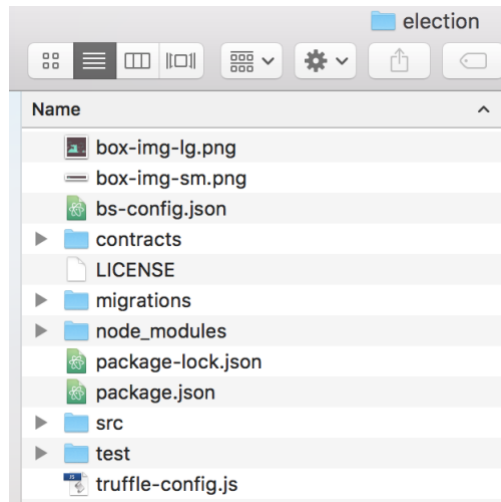
Unbox successful. Sweet!

Commands:

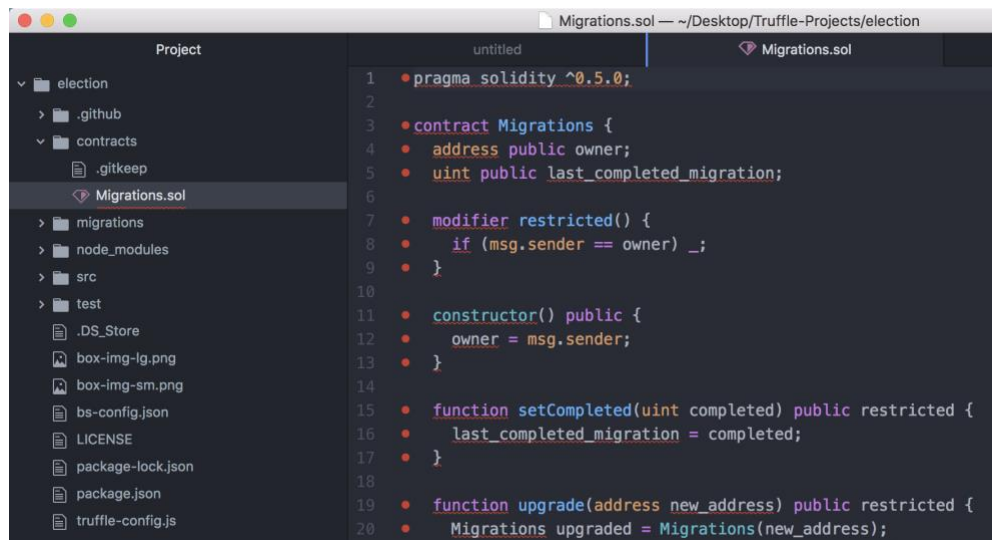
  Compile:      truffle compile
  Migrate:      truffle migrate
  Test contracts: truffle test
  Run dev server: npm run dev

ksuParizi:election rparizi1$
```

Check out your election folder, Let's see what the pet shop box gave us (assuming you had removed your bare truffle project in 'election' before unboxing pet-shop):



For a better experience, you can open your project in Atom editor, and work with it through the tutorial:



2. Write your smart contracts

Now let's start writing our smart contract! The smart contract will contain all the business logic of our dApp. It will be in charge reading from and write to the Ethereum blockchain. It will allow us to list the candidates that will run in the election, and keep track of all the votes and voters. It will also govern all of the rules of the election, like enforcing accounts to only vote once. From the root of your project, go ahead and create a new contract empty file in the contracts directory like this:

```
touch contracts/Election.sol
```

Paste the enclosed code in your created file 'Election. sol'. Link to [download](#). Try to understand the code before you move forward!

Note, by default there will be a contract named Migrations.sol under 'contracts', leave it there (just change its compiler version to pragma solidity 0.5.0)

3. Compile your smart contracts

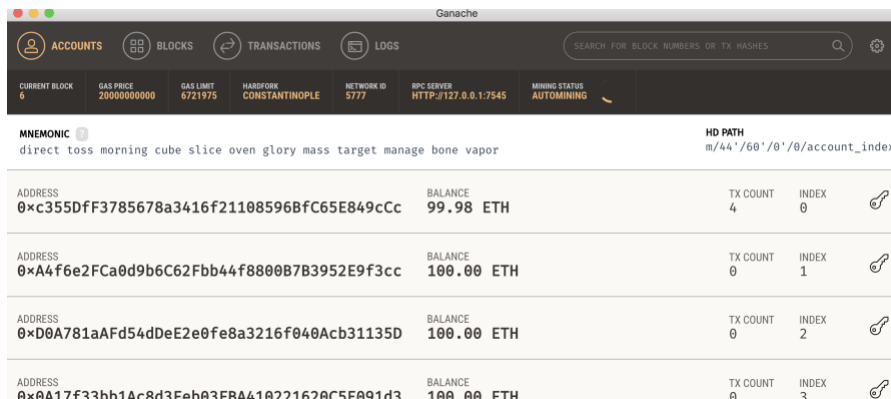
Use the command: `truffle compile`

If successful, you will see a 'build' folder appears. *build/contracts/Election.json* file. You can open it to see what it is inside. It is a json file. There are two keys of this json object that are very important, which are *ABI* and *bytecode*. The *bytecode* is the content that the Ethereum Virtual Machine understands. It's like the binary file. The *ABI* is the interface so we can interact with the smart contract

Note: if you don't see any action after running the command, try to use `sudo` preceding to the command!

4. Deploy your contracts to your blockchain network

Before doing so, make sure you will have a look at your `truffle-config.js` to specify which Ethereum network (or simply blockchain) you would like to deploy your dapp to. By default (in this tutorial), you don't need to make any changes as it already set to match a local Ethereum client. That is, we will use a local blockchain known as *development network* (simulated with 10 nodes). To run it: simply open Ganache GUI--recommended (or use either of these commands: `truffle develop` or `testrpc`). Note: once you are confident that you have created a right application (after testing), you can come back to this file and change it settings to connect it to live networks (for more information about configuration read [here](#)).



Ganache

After this, we need to write the migration code. To some people who are familiar with frameworks like Ruby on Rails, migration in this case is not like database migration. Here, migration means the process to deploy that particular smart contract. In order to do this, we'll need to create a new file in the migrations directory. From your project root, create a new file from the command line like this (you can change the name though):

```
touch migrations/2_deploy_contracts.js
```

Notice that we number all of our files inside the migrations directory with numbers so that Truffle knows which order to execute them in. Copy this content to that file:

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

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

Now you can deploy the smart contract (i.e. run our migrations) from the command line like this:

```
truffle migrate
```

or

```
truffle migrate --network ganache
```

You will get the output like this:

```
Starting migrations...
=====
> Network name:      'development'
> Network id:        5777
> Block gas limit:   6721975

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash:  0x77347833ac43b8654e600125ebc89630cf9fdf7965852a6cee8ec2115fe84b7b
> Blocks: 0         Seconds: 0
> contract address: 0xd5dd95933D718Dd6775884158bfF794425Bd15f8
> account:          0xc355DfF3785678a3416f211085968fC65E849cCc
> balance:          99.97468532
> 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 'Election'
-----
> transaction hash:  0x63fce30301ebd98389cd9c543111757dc35e8d06b57cbfb0d39a1e0a87ba1146
> Blocks: 0         Seconds: 0
> contract address: 0x8Ca8B1fd7aC9D5c52d3913c1F2d10F3138f85726
> account:          0xc355DfF3785678a3416f211085968fC65E849cCc
> balance:          99.96476038
> gas used:         454213
> gas price:        20 gwei
> value sent:       0 ETH
> total cost:       0.00908426 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:       0.00908426 ETH

Summary
=====
> Total deployments: 2
> Final cost:       0.01478242 ETH
```

Your output will not be same 100% with mine. The addresses will be different.

Now, your dapp is ready to use! You can go to **Step 3** if you want to skip testing at this time.

[truffle migrate --reset used for resetting the migration process]

5. Test your contracts

Why testing is so important when you're developing smart contracts. We want to ensure that the contracts are bug free for a few reasons:

1. All of the code on the Ethereum blockchain is immutable; it cannot change. If the contract contains any bugs, we must disable it and deploy a new copy. This new copy will not have the same state as the old contract, and it will have a different address.
2. Deploying contracts costs gas because it creates a transaction and writes data to the blockchain. This costs Ether, and we want to minimize the amount of Ether we ever have to pay.
3. If any of our contract functions that write to the blockchain contain bugs, the account who is calling this function could potentially waste Ether, and it might not behave the way they expect.

Now let's write some tests. Make sure you have your network running (Ganache). Then, create a new test file in the command line from the root of your project like this:

```
touch test/election.js
```

Paste these test cases:

We'll write all our tests in Javascript inside this file with the [Mocha testing framework](#) and the [Chai assertion library](#). These come bundled with the Truffle framework. We'll write all these tests in Javascript to simulate client-side interaction with our smart contract, much like we did in the console. Here is all the code for the tests:

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

contract("Election", function(accounts) {
  var electionInstance;

  it("initializes with two candidates", function() {
    return Election.deployed().then(function(instance) {
      return instance.candidatesCount();
    }).then(function(count) {
      assert.equal(count, 2);
    });
  });

  it("it initializes the candidates with the correct values", function() {
    return Election.deployed().then(function(instance) {
```

```

    electionInstance = instance;
    return electionInstance.candidates(1);
  }).then(function(candidate) {
    assert.equal(candidate[0], 1, "contains the correct id");
    assert.equal(candidate[1], "Candidate 1", "contains the correct name");
    assert.equal(candidate[2], 0, "contains the correct votes count");
    return electionInstance.candidates(2);
  }).then(function(candidate) {
    assert.equal(candidate[0], 2, "contains the correct id");
    assert.equal(candidate[1], "Candidate 2", "contains the correct name");
    assert.equal(candidate[2], 0, "contains the correct votes count");
  });
});
});

```

Let me explain this code. First, we require the contract and assign it to a variable, like we did in the migration file. Next, we call the "contract" function, and write all our tests within the callback function. This callback function provides an "accounts" variable that represents all the accounts on our blockchain, provided by Ganache.

The first test checks that the contract was initialized with the correct number of candidates by checking the candidates count is equal to 2.

The next test inspects the values of each candidate in the election, ensuring that each candidate has the correct id, name, and vote count.

Now let's add a test to our "election.js" test file:

```

it("allows a voter to cast a vote", function() {
  return Election.deployed().then(function(instance) {
    electionInstance = instance;
    candidateld = 1;
    return electionInstance.vote(candidateld, { from: accounts[0] });
  }).then(function(receipt) {
    return electionInstance.voters(accounts[0]);
  }).then(function(voted) {

```

```

    assert(voted, "the voter was marked as voted");
    return electionInstance.candidates(candidateId);
  }).then(function(candidate) {
    var voteCount = candidate[2];
    assert.equal(voteCount, 1, "increments the candidate's vote count");
  })
});

```

We want to test two things here:

1. Test that the function increments the vote count for the candidate.
2. Test that the voter is added to the mapping whenever they vote.

Next we can write a few test for our function's requirements. Let's write a test to ensure that our vote function throws an exception for double voting:

```

it("throws an exception for invalid candidates", function() {
  return Election.deployed().then(function(instance) {
    electionInstance = instance;
    return electionInstance.vote(99, { from: accounts[1] })
  }).then(assert.fail).catch(function(error) {
    assert(error.message.indexOf('revert') >= 0, "error message must contain revert");
    return electionInstance.candidates(1);
  }).then(function(candidate1) {
    var voteCount = candidate1[2];
    assert.equal(voteCount, 1, "candidate 1 did not receive any votes");
    return electionInstance.candidates(2);
  }).then(function(candidate2) {
    var voteCount = candidate2[2];
    assert.equal(voteCount, 0, "candidate 2 did not receive any votes");
  });
});

```

We can assert that the transaction failed and that an error message is returned. We can dig into this error message to ensure that the error message contains the "revert" substring. Then we can ensure that our contract's state was unaltered by ensuring that the candidates did not receive any votes.

Now let's write a test to ensure that we prevent double voting:

```

it("throws an exception for double voting", function() {
  return Election.deployed().then(function(instance) {
    electionInstance = instance;
    candidateId = 2;
    electionInstance.vote(candidateId, { from: accounts[1] });
    return electionInstance.candidates(candidateId);
  }).then(function(candidate) {
    var voteCount = candidate[2];
    assert.equal(voteCount, 1, "accepts first vote");
    // Try to vote again
    return electionInstance.vote(candidateId, { from: accounts[1] });
  }).then(assert.fail).catch(function(error) {
    assert(error.message.indexOf('revert') >= 0, "error message must contain revert");
    return electionInstance.candidates(1);
  }).then(function(candidate1) {
    var voteCount = candidate1[2];
    assert.equal(voteCount, 1, "candidate 1 did not receive any votes");
    return electionInstance.candidates(2);
  }).then(function(candidate2) {
    var voteCount = candidate2[2];
    assert.equal(voteCount, 1, "candidate 2 did not receive any votes");
  });
});

```

First, we'll set up a test scenario with a fresh account that hasn't voted yet. Then we'll cast a vote on their behalf. Then we'll try to vote again. We'll assert that an error has occurred here. We can inspect the error message, and ensure that no candidates received votes, just like the previous test.

Now let's run the five tests from the command line like this:

```
truffle test
```

Yay, they passed! 🎉 (or may not)

Step 3: Building Frontend of your Dapp

Now let's start building out the front-end application that will talk to our smart contract. There are many ways to do this. Here, we'll do this by modifying the HTML and Javascript files that came with the Truffle Pet Shop box that we installed in the previous section.

We'll use this existing code to get started. Let's also take note of a few other things that came with the Truffle Pet Shop box like the Bootstrap framework that will keep us from having to write any CSS in this tutorial. We also got lite-server, which will serve our assets for development purposes.

You do not have to be a front-end expert to follow along with this part of the tutorial. I have intentionally kept the HTML and Javascript code very simple, and we will not spend much time focusing on it.

Go ahead to src folder and replace all of the content of your "**index.html**" file with this code: [download](#).

Next, replace all of the content of your "**app.js**" file with this code: [download](#).
[Web3.js is used]

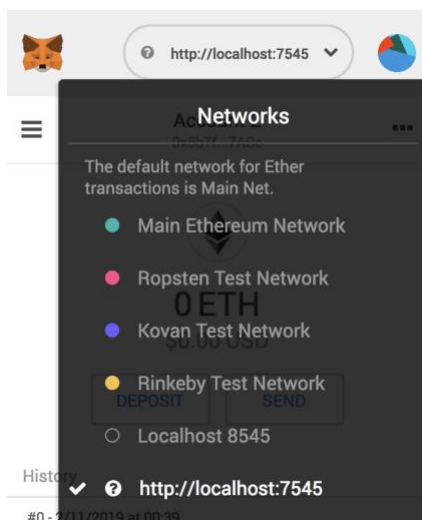
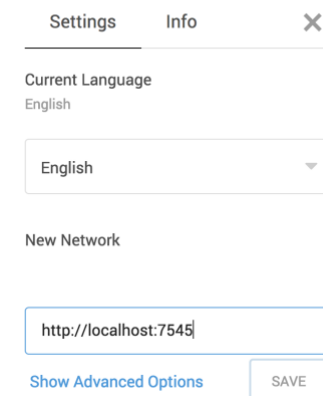
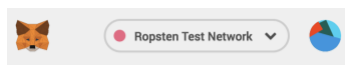
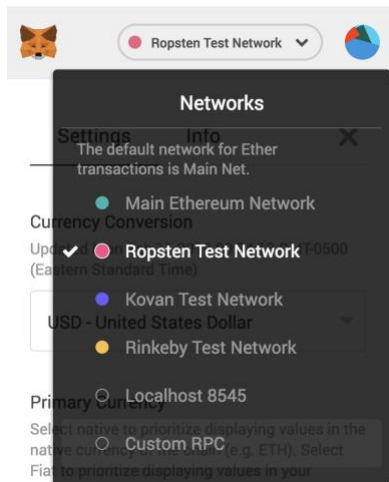
Next, start your development server from the command line like this:

```
npm run dev
```

This should automatically open a new browser window with your application (<http://localhost:3000/>).

Notice that your application says "Loading...". That's because we're not logged in to the blockchain network yet! In order to connect to the blockchain, we need to set up MetaMask. To set up MetaMask: 1) from drop-down list of networks choose **Custom RPC** 2) type in "<http://localhost:portnumber>" under network and save. See screenshots!

Try to vote for candidates with different addresses (note, refresh chrome every time you vote with an address, and change the address (voter) to see the form and vote button—one address can only vote once).



Congratulations! 🎉 You have successfully built a full stack decentralized application on the Ethereum blockchain!