**TRUFFLE OVERVIEW**

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. With Truffle, you get:

* Built-in smart contract compilation, linking, deployment and binary management.
* Automated contract testing for rapid development.
* Scriptable, extensible deployment & migrations framework.
* Network management for deploying to any number of public & private networks.
* Package management with EthPM & NPM, using the [ERC190](https://github.com/ethereum/EIPs/issues/190) standard.
* Interactive console for direct contract communication.
* Configurable build pipeline with support for tight integration.
* External script runner that executes scripts within a Truffle environment

**INSTALLATION**

* npm install -g truffle

**Requirements**

* NodeJS 5.0+ recommended.
* Windows, Linux or Mac OS X
* Truffle also requires that you have a running Ethereum client which supports the standard JSON RPC API.

**COMPILING CONTRACTS**

**Location**

All of our contracts are located in your project's contracts/ directory. As contracts are written in [Solidity](https://solidity.readthedocs.io/en/develop/), all files containing contracts will have a file extension of .sol. Associated Solidity [libraries](http://solidity.readthedocs.org/en/latest/contracts.html#libraries) will also have a .sol extension.

With a bare Truffle [project](https://truffleframework.com/docs/truffle/quickstart) (created through **truffle** **init**), we're given a single Migrations.sol file that helps in the deployment process.

**Command**

To compile a Truffle project, change to the root of the directory where the project is located and then type the following into a terminal:

* **truffle compile**

Upon first run, all contracts will be compiled. Upon subsequent runs, Truffle will compile **only the contracts that have been changed since the last compile**. If you'd like to override this behavior, run the above command with the **--all**option.

**Build artifacts**

Artifacts of our compilation will be placed in the build/contracts/ directory, relative to our project root. (This directory will be created if it does not exist.)

These artifacts are integral to the inner workings of Truffle, and they play an important part in the successful deployment of your application.

**Dependencies**

We can declare contract dependencies using Solidity's [import](http://solidity.readthedocs.org/en/latest/layout-of-source-files.html#importing-other-source-files) command. Truffle will compile contracts in the correct order and ensure all dependencies are sent to the compiler. Dependencies can be specified in two ways:

* import "./AnotherContract.sol";
* import "somepackage/SomeContract.sol";

**RUNNING** **MIGRATIONS**

Migrations are JavaScript files that help us deploy contracts to the Ethereum network. These files are responsible for staging your deployment tasks, and they're written under the assumption that our deployment needs will change over time. As your project evolves, you'll create new migration scripts to further this evolution on the blockchain. A history of previously run migrations is recorded on-chain through a special Migrations contract, detailed below. For successful migration the **maximum bytecode size** of the contract getting deployed should not more than **24kB.**

**Command**

To run your migrations, run the following:

* **truffle migrate**

This will run all migrations located within our project's migrations directory. At their simplest, migrations are simply a set of managed deployment scripts. If **our migrations were previously run successfully**, truffle migrate will start execution from the last migration that was run, running **only newly created migrations**. **If no new migrations exists, truffle migrate won't perform any action at all**. You can use the **--reset** option to run all your migrations from the beginning.

**Migration Files**

1. Filename: migrations/1\_initial\_migration.js

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

module.exports = function(deployer) { *// Deploy the Migrations contract as our only task*

deployer.deploy(Migrations);};

1. Filename: migrations/2\_deploy\_contracts.js

var Echoupal1 = artifacts.require("Echoupal1");var Echoupal2 = artifacts.require("Echoupal2");module.exports = function (deployer) {

*//Deploy Echoupal1 and when it gets mined, deploy Echoupal2* deployer.deploy(Echoupal1).then(function () {

return deployer.deploy(Echoupal2)

});};

**CONFIGURATION**

**Location**

Our configuration file is called **truffle.js** and is located at the root of our project directory. This file is a Javascript file and can execute any code necessary to create your configuration. It must export an object representing our project configuration like the example below.

module.exports = {

networks: {

development: {

host: "localhost",

port: 8545,

gas: 8000000,

gasPrice: 0,

network\_id: "2018"

}

},

solc: {

optimizer: {

enabled: true,

runs: 200

}}

**truffle-contract**

Better Ethereum contract abstraction, for Node and the browser.

**Install**

* **npm install truffle-contract**

**Features**

* Synchronized transactions for better control flow (i.e., transactions won't finish until you're guaranteed they've been mined).
* Promises. No more callback hell. Works well with ES6 and async/await.
* Default values for transactions, like from address or gas.
* Returning logs, transaction receipt and transaction hash of every synchronized transaction.

**Usage**

First, set up a new web3 provider instance and initialize your contract, then require("truffle-contract"). The input to the contract function is a JSON blob defined by [truffle-contract-schema](https://github.com/trufflesuite/truffle-contract-schema). This JSON blob is structured in a way that can be passed to all truffle-related projects.

var provider = new Web3.providers.HttpProvider("**http://localhost:8545**");

var contract = require("truffle-contract");

var MyContract = contract({

abi: ...,

unlinked\_binary: ...,

address: ..., // optional

// many more

})

MyContract.setProvider(provider);

We now have access to the following functions on MyContract, as well as many others:

* **at():** Create an instance of MyContract that represents your contract at a specific address.
* **deployed():** Create an instance of MyContract that represents the default address managed by MyContract.
* **new():** Deploy a new version of this contract to the network, getting an instance of MyContract that represents the newly deployed instance.

**Executing contract functions**

Using the abstraction we can easily execute contract functions on the Ethereum network.

**Sample contract**

pragma solidity ^0.4.2;

import "./ConvertLib.sol";

// This is just a simple example of a coin-like contract.

// It is not standards compatible and cannot be expected to talk to other

// coin/token contracts. If you want to create a standards-compliant

// token, see: https://github.com/ConsenSys/Tokens. Cheers!

contract MetaCoin {

mapping (address => uint) balances;

event Transfer(address indexed \_from, address indexed \_to, uint256 \_value);

function MetaCoin() {

balances[tx.origin] = 10000;

}

function sendCoin(address receiver, uint amount) returns(bool sufficient) {

if (balances[msg.sender] < amount) return false;

balances[msg.sender] -= amount;

balances[receiver] += amount;

Transfer(msg.sender, receiver, amount);

return true;

}

function getBalanceInEth(address addr) returns(uint){

return ConvertLib.convert(getBalance(addr),2);

}

function getBalance(address addr) returns(uint) {

return balances[addr];

}}

**Making a transaction**

There are three functions on the MetaCoin(sample) contract that we can execute. If we analyze each of them, we’ll see that sendCoin is the only function that aims to make changes to the network. The goal of sendCoin is to "send" some Meta coins from one account to the next, and these changes should persist.

When calling sendCoin, we'll execute it as a transaction. In the following example, we'll send 10 Meta coin from one account to another, in a way that persists changes on the network:

var account\_one = "0x1234..."; // an address

var account\_two = "0xabcd..."; // another address

var meta;

MetaCoin.deployed().then(function(instance) {

meta = instance;

return meta.sendCoin(account\_two, 10, {from: account\_one});

}).then(function(result) {

// If this callback is called, the transaction was successfully processed.

alert("Transaction successful!")

}).catch(function(e) {

// There was an error! Handle it.

})

There are a few things interesting about the above code:

* We called the **abstraction's sendCoin function directly**. This will result in a transaction by default (i.e., writing data) instead of call.
* **When the transaction is successful, the callback function isn't fired until the transaction is processed.** This makes life easy and means you don't have to check the status of the transaction yourself.
* We passed an object as the third parameter to sendCoin. Note that the sendCoin function in our Solidity contract doesn't have a third parameter. What you see above is a special object that can always be passed as the last parameter to a function that lets you edit specific details about the transaction. Here, we set the fromaddress ensuring this transaction came from account\_one.

**Making a call**

Continuing with MetaCoin, notice the getBalance function is a great candidate for reading data from the network. It doesn't need to make any changes, as it just returns the MetaCoin balance of the address passed to it. Let's give it a shot:

var account\_one = "0x1234..."; // an address

var meta;

MetaCoin.deployed().then(function(instance) {

meta = instance;

return meta.getBalance.call(account\_one, {from: account\_one});

}).then(function(balance) {

// If this callback is called, the call was successfully executed.

// Note that this returns immediately without any waiting.

// Let's print the return value.

console.log(balance.toNumber());

}).catch(function(e) {

// There was an error! Handle it.

})

What's interesting here:

We had to execute the**.call()**function explicitly to let the Ethereum network know we're not intending to persist any changes.

We received a return value instead of a transaction id on success. Note that since the Ethereum network can handle very large numbers, we're given a [BigNumber](https://github.com/MikeMcl/bignumber.js/) object which we then convert to a number.

Warning: We convert the return value to a number because in this example the numbers are small. However, if you try to convert a BigNumber that's larger than the largest integer supported by Javascript, you'll likely run into errors or unexpected behavior.

**Catching events**

Our contracts can fire events that we can catch to gain more insight into what our contracts are doing. The easiest way to handle events is by processing the result object of the transaction that triggered the event, like so:

var account\_one = "0x1234..."; // an address

var account\_two = "0xabcd..."; // another address

var meta;

MetaCoin.deployed().then(function(instance) {

meta = instance;

return meta.sendCoin(account\_two, 10, {from: account\_one});

}).then(function(result) {

// result is an object with the following values:

//

// result.tx => transaction hash, string

// result.logs => array of decoded events that were triggered within this transaction

// result.receipt => transaction receipt object, which includes gas used

// We can loop through result.logs to see if we triggered the Transfer event.

for (var i = 0; i < result.logs.length; i++) {

var log = result.logs[i];

if (log.event == "Transfer") {

// We found the event!

break;

}

}

}).catch(function(err) {

// There was an error! Handle it.

});

**Processing transaction results**

When we make a transaction, we're given a result object that gives you a wealth of information about the transaction. Specifically, you get the following:

* **result.tx (string)** - Transaction hash
* **result.logs (array)** - Decoded events (logs)
* **result.receipt (object)** - Transaction receipt

**Add a new contract to the network**

In all of the above cases, we've been using a contract abstraction that has already been deployed. We can deploy our own version to the network using the **.new()** function:

MetaCoin.new().then(function(instance) {

// Print the new address

console.log(instance.address);

}).catch(function(err) {

// There was an error! Handle it.

});

**Use a contract at a specific address**

If we already have an address for a contract, we can create a new abstraction to represent the contract at that address.

var instance = MetaCoin.**at("0x1234...");**

**Sending ether to a contract**

We may simply want to send Ether directly to a contract, or trigger a contract's [fallback function](http://solidity.readthedocs.io/en/develop/contracts.html#fallback-function). You can do so using one of the following two options.

Option 1: Send a transaction directly to a contract via instance.sendTransaction(). This is promisified like all available contract instance functions, and has the same API as web3.eth.sendTransaction but without the callback. The to value will be automatically filled in for you if not specified.

instance.sendTransaction({...}).then(function(result) {

// Same transaction result object as above.

});

Option 2: There's also shorthand for just sending Ether directly:

instance.send(web3.toWei(1, "ether")).then(function(result) {

// Same result object as above.});

**NETWORKS AND APP DEPLOYMENT**

Even the smallest project will interact with at the very least two blockchain nodes: One on the developer's machine, like [Ganache](https://truffleframework.com/ganache) or Truffle Develop, and the other representing the network where the developer will eventually deploy their application (such as the main public Ethereum network or a private consortium network, for instance). Truffle provides a system for managing the compilation and deployment artifacts for each network, and does so in a way that simplifies final application deployment.

**Specifying a network**

Most Truffle commands will behave differently based on the network specified, and will use that network's contracts and configuration. You can specify a network using the --network option, like below:

* **truffle migrate --network live**

In this example, Truffle will run your migrations on the "**live**" network, is associated with the public Ethereum blockchain.

**Build artifacts**

As mentioned in the [Compiling contracts](https://truffleframework.com/docs/truffle/getting-started/compiling-contracts) section, build artifacts are stored in the ./build/contracts directory as .json files. When you compile your contracts or run your migrations using a specific network, Truffle will update those .json files so they contain the information related to that network. When those artifacts are used later -- such as within your frontend or application via [truffle-contract](https://github.com/trufflesuite/truffle/tree/master/packages/truffle-contract) -- they'll automatically detect which network the Ethereum client is connected to and use the correct contract artifacts accordingly.

**Application deployment**

Because the network is auto-detected by the contract artifacts at runtime, this means that we only need to deploy your application or frontend once. When you run your application, the running Ethereum client will determine which artifacts are used, and this will make your application very flexible.

**General Options**

**contracts\_build\_directory**

The default output directory for compiled contracts is ./build/contracts relative to the project root. This can be changed with the contracts\_build\_directory key.

Examples:

To place the built contract artifacts in ./output/contracts:

module.exports = {

contracts\_build\_directory: "./output",

networks: {

development: {

host: "127.0.0.1",

port: 8545,

network\_id: "\*",

}

}

};

The built contract artifacts do not need to be inside the project root:

module.exports = {

contracts\_build\_directory: "../../../output",

networks: {

development: {

host: "127.0.0.1",

port: 8545,

network\_id: "\*",

}

}

};

Absolute paths will also work.

**networks**

Specifies which networks are available for deployment during migrations, as well as specific transaction parameters when interacting with each network (such as gas price, from address, etc.). When compiling and running migrations on a specific network, contract artifacts will be saved and recorded for later use. When your contract abstractions detect that your Ethereum client is connected to a specific network, they'll use the contract artifacts associated that network to simplify app deployment. Networks are identified through Ethereum's net\_version RPC call, as well as blockchain URIs.

The networks object, shown below, is keyed by a network name and contains a corresponding object that defines the parameters of the network. The networks option is required, as if you have no network configuration, Truffle will not be able to deploy your contracts. The default network configuration provided by truffle init gives you a development network that matches any network it connects to -- this is useful during development, but not suitable for production deployments. To configure Truffle to connect to other networks, simply add more named networks and specify the corresponding network id.

The network name is used for user interface purposes, such as when running your migrations on a specific network:

* **truffle migrate --network live**

Example:

networks: {

development: {

host: "127.0.0.1",

port: 8545,

network\_id: "\*" // match any network

},

live: {

host: "178.25.19.88", // Random IP for example purposes (do not use)

port: 80,

network\_id: 1, // Ethereum public network

// optional config values:

// gas

// gasPrice

// from - default address to use for any transaction Truffle makes during migrations

// provider - web3 provider instance Truffle should use to talk to the Ethereum network.

// - function that returns a web3 provider instance (see below.)

// - if specified, host and port are ignored.

}

}

For each network, if unspecified, transaction options will default to the following values:

* **gas:** Gas limit used for deploys. Default is 4712388.
* **gasPrice:** Gas price used for deploys. Default is 100000000000 (100 Shannon).
* **from:** From address used during migrations. Defaults to the first available account provided by your Ethereum client.
* **provider:** Default web3 provider using host and port options: new Web3.providers.HttpProvider("http://<host>:<port>")

For each network, you can specify either host / port or provider, but not both. If you need an HTTP provider, we recommend using host and port, while if you need a custom provider such as HDWalletProvider, you must use provider.

**Providers**

The following network list consists of a local test network and an Infura-hosted Ropsten network, both provided by HDWalletProvider. Make sure you wrap truffle-hdwallet providers in a function closure as shown below to ensure that only one network is ever connected at a time.

networks: {

ropsten: {

provider: function() {

return new HDWalletProvider(mnemonic, "https://ropsten.infura.io/");

},

network\_id: '3',

},

test: {

provider: function() {

return new HDWalletProvider(mnemonic, "http://127.0.0.1:8545/");

},

network\_id: '\*',

},

}

If we specify host and port instead of provider, Truffle will create its own default HTTP provider using that host and port, and no minimal network connection will be opened, so there is no need to do the function wrapping workaround. That said, you wouldn't be able to use a custom provider in this case.

**Solidity compiler configuration**

Solidity compiler settings. Supports optimizer settings for solc.

**solc**

Configuration options to pass to the Solidity compiler.

Example:

solc: {

optimizer: {

enabled: true,

runs: 200

}

}

**TRUFFLE COMMANDS**

**Available commands**

**build**

Execute build pipeline (if configuration present).

* **truffle build [deprecated]**

Requires the build key to be present in the configuration.

**compile**

Compile contract source files.

* **truffle compile [--all] [--network <name>]**

This will only compile contracts that have changed since the last compile, unless otherwise specified.

Optional parameters:

* **--all:** Compile all contracts instead of only the contracts changed since last compile.
* **--network <name>:** Specify the network to use, saving artifacts specific to that network. Network name must exist in the configuration.

**console**

Run a console with contract abstractions and commands available.

* **truffle console [--network <name>] [--verbose-rpc]**

Spawns an interface to interact with contracts via the command line. Additionally, many Truffle commands are available within the console (without the truffle prefix).

Requires an external Ethereum client, such as [Ganache](https://truffleframework.com/docs/ganache/using) or geth. For a console that creates a development and test environment, use **truffle develop.**

Optional parameters:

* **--network <name>:** Specify the network to use. Network name must exist in the configuration.
* **--verbose-rpc:** Log communication between Truffle and the Ethereum client.

**create**

Helper to create new contracts, migrations and tests.

* **truffle create (contract|migration|test) <ArtifactName>**

Parameters:

* **contract:** Create a new contract definition and file contracts/ArtifactName.sol.
* **migration:** Create a new migration and file migrations/###########\_artifact\_name.js.
* **test:** Create a new test and file tests/artifact\_name.js.

<ArtifactName>: Name of new artifact.

Camel case names of artifacts will be converted to underscore-separated file names for the migrations and tests. Number prefixes for migrations are automatically generated.

**debug**

Interactively debug any transaction on the blockchain.

* **truffle debug <transaction\_hash> [experimental]**

Will start an interactive debugging session on a particular transaction. Allows you to step through each action and replay.

Parameters:

<transaction\_hash>: Transaction ID to use for debugging.

**deploy**

Alias for migrate. See migrate for details.

**develop**

Open a console with a development blockchain

* **truffle develop**

Spawns a local development blockchain, and allows you to interact with contracts via the command line. Additionally, many Truffle commands are available within the console (without the truffle prefix).

If we want an interactive console but want to use an existing blockchain, use truffle console.

**exec**

Execute a JS module within the Truffle environment.

* **truffle exec <script.js> [--network <name>]**

This will include web3, set the default provider based on the network specified (if any), and include our contracts as global objects while executing the script. Our script must export a function that Truffle can run.

Parameters:

* **<script.js>:** JavaScript file to be executed. Can include path information if the script does not exist in the current directory.

Optional parameters:

* **--network <name>:** Specify the network to use, using artifacts specific to that network. Network name must exist in the configuration.

**help**

Display a list of all commands and then exit.

* **truffle help**

**init**

Initialize new and empty Ethereum project

* **truffle init**

Creates a new and empty Truffle project within the current working directory. Takes no arguments.

**install**

Install a package from the Ethereum Package Registry.

* **truffle install [package\_name]<@version>**

Parameters:

* **package\_name:** Name of the package as listed in the Ethereum Package Registry.

Optional parameters:

* **<@version>:** When specified, will install a specific version of the package, otherwise will install the latest version.

**migrate**

Run migrations to deploy contracts.

* **truffle migrate [--reset] [-f <number>] [--network <name>] [--compile-all] [--verbose-rpc]**

Unless specified, this will run from the last completed migration.

Optional parameters:

* **--reset:** Run all migrations from the beginning, instead of running from the last completed migration.
* **-f <number>:** Run contracts from a specific migration. The number refers to the prefix of the migration file.
* **--network <name>:** Specify the network to use, saving artifacts specific to that network. Network name must exist in the configuration.
* **--compile-all:** Compile all contracts instead of intelligently choosing which contracts need to be compiled.
* **--verbose-rpc:** Log communication between Truffle and the Ethereum client.

**networks**

Show addresses for deployed contracts on each network.

* **truffle networks [--clean]**

Use this command before publishing our package to see if there are any extraneous network artifacts we don't want published. With no options specified, this package will simply output the current artifact state.

Optional parameters:

* **--clean:** Remove all network artifacts that aren't associated with a named network.

**opcode**

Print the compiled opcodes for a given contract.

* **truffle opcode <contract\_name>**

Parameters:

* **<contract\_name>:** Name of the contract to print opcodes for. Must be a contract name, not a file name.

**publish**

Publish **a package to the Ethereum Package Registry.**

* **truffle publish**

All parameters are pulled from your project's configuration file. Takes no arguments.

**serve**

Serve the built app from http://127.0.0.1:8080, rebuilding and redeploying changes as needed. Similar to truffle watch, but with the web server component added.

* **truffle serve [-p <port>] [--network <name>]**

Optional parameters:

* -p <port>: Specify the port to serve on. Default is 8080.
* --network <name>: Specify the network to use, using artifacts specific to that network. Network name must exist in the configuration.

**test**

Run JavaScript and Solidity tests.

* **truffle test <test\_file> [--compile-all] [--network <name>] [--verbose-rpc]**

Runs some or all tests within the test/ directory as specified.

Parameters:

* **<test\_file>:** Name of the test file to be run. Can include path information if the file does not exist in the current directory.

Optional parameters:

* **--compile-all:** Compile all contracts instead of intelligently choosing which contracts need to be compiled.
* **--network <name>:** Specify the network to use, using artifacts specific to that network. Network name must exist in the configuration.
* **--verbose-rpc:** Log communication between Truffle and the Ethereum client.

**unbox**

Download a Truffle Box, a pre-built Truffle project.

* **truffle unbox <box\_name>**

Downloads a [Truffle Box](https://truffleframework.com/boxes) to the current working directory.

Parameters:

* **<box\_name>:** Name of the Truffle Box.

**version**

Show version number and exit.

* **truffle version**

**watch**

Watch filesystem for changes and rebuild the project automatically.

* **truffle watch**

This command will initiate a watch for changes to contracts, application, and configuration files. When there's a change, it will rebuild the app as necessary. Similar to truffle serve, but without the web server component.