dapp Documentation

Release 0.1

DappHub

Jul 22, 2017

Contents:

Dapp is a simple command line tool for smart contract development. It supports these common usecases:

- Package management
- Source code building
- Unit testing
- Simple contract deployments

CHAPTER 1

Installing Dapp

Installing with Nix

Soon dapp will be completely distributed via the Nix Package Manager, which allows us to provide a consistent experience across all Unix-based operating systems (i.e. various Linux distros and OSX/Mac OS). Getting started with Nix is very easy and you will be glad you did it. Its purely functional approach is unique to the domain of package management and it makes Nix quite powerful.

In the meantime, you will need to perform some manual steps to get dapp. This section will be updated as the installation process gets streamlined.

First download ethrun, rename it to `ethrun`, and move it to somewhere on your $PATH:

```
$ cp ethrun-v0.1.0-osx /usr/local/bin/ethrun
```

The rest of the installation process looks like this:

```
$ curl https://nixos.org/nix/install | sh
$ nix-channel --add https://nix.dapphub.com/pkgs/dapphub
$ nix-channel --update
$ nix-env -i seth solc
$ git clone https://github.com/dapphub/dapp
$ make link -C dapp
```

Dapp depends on some blockchain development tools that are very useful in their own right. In particular, you may find seth convenient when working with any Ethereum client's JSON RPC API. You can explore the full list of dependencies below.

Installing without Nix

The program will expect you to have these dependencies installed:

- bc

- curl
- git
- node
- solc
- ethabi
- ethrun
- seth
- jshon

None of these programs are too difficult to install, and it's likely that you will have the top half of the list on your computer already. Once you have them, installing dapp is as simple as:

```
$ make link
```

This will create a symlink to your repository in `/usr/local`.

CHAPTER 2

Getting Started

Learning Solidity

Without a doubt, the official documentation website is the most up-to-date and useful resource for learning Solidity. The language subtly changes fairly often, so being up-to-date is particularly valuable when working in this domain. Start from the first page and read to the last. It may take a while, but you will have a good understanding of Solidity by the end.

Initializing A Workspace

These are the commands to create a brand new project using dapp:

```
$ mkdir dapp-tutorial
$ cd dapp-tutorial
$ dapp init
```

You will see that dapp does a few things for you:

1. Initializes a git repository if you don't have one.
2. Creates a `Makefile` for convenience.
3. Creates a `Dappfile` for configuration info:

```
name            dapp-tutorial
description
version         0.0.1

author          amilenius
license         Apache-2.0
```

4. Installs a package called `ds-test`, which is used when *testing your contracts*. This package is installed in the `lib` folder.

5. Creates a `src` directory and `test.sol`
6. Runs a sanity test with `dapp test`. This causes the `Test` contract from `test.sol` to be built. The compiler's output is put in the `out` directory.
7. Commits the changes that were made.

CHAPTER 3

Package Management

Dapp's package management feature allows you to both download and publish smart contracts for the purposes of code reuse and discovery. Recently, a new standard was created called the Ethereum Smart Contract Packaging Specification, or EthPM for short. It uses IPFS addresses to identify and distribute smart contract code in a decentralized manner.

In the near future, a registry contract will be deployed to the Ethereum blockchain for easy lookup of package addresses by unique human-friendly names. This will allow for a user experience identical to other package managers you might be more familiar with (e.g. `npm install my-cool-package`). In the meantime, dapp uses git submodules to mimic the experience of installing EthPM packages.

Note: A good resource for working with git submodules can be found at Chris Jean's blog

Installing Packages

Dapp will install a git submodule for you based on the github path you specify. If you don't specify a github user, dapp will choose dapphub as the default. Thus both of these commands will install code in your `lib` folder:

```
# installing a submodule from  https://github.com/dapphub/ds-auth

$ dapp install ds-auth
```

and:

```
# installing a submodule from  https://github.com/apmilen/my-cool-package

$ dapp install apmilen/my-cool-package
```

Files in your installed packages can be imported in your source code by referencing the git submodule name:

```
pragma solidity ^0.4.0;

import "ds-auth/auth.sol";

contract TestContract is DSAuth {}
```

Publishing Packages

Since dapp package management currently works off git submodules, publishing code to the default branch of your repository is equivalent to releasing a new package.

Discovering Packages

Coming soon!

CHAPTER 4

Building Source Code

Building source code is easy with Dapp:

```
$ dapp build
```

This will take all the code in the `src` and `lib` directories and feed it to the `solc` compiler. There are a few build artifacts that are deposited in the `build` directory, and it's worth going over each one specifically.

Build Artifacts

Suppose you had this abridged token contract:

```
contract DSTokenBase {

    event Transfer( address indexed from, address indexed to, uint value);

    mapping( address => uint ) _balances;
    uint _supply;

    function DSTokenBase( uint initial_balance ) {
        _balances[msg.sender] = initial_balance;
        _supply = initial_balance;
    }

    function balanceOf( address who ) constant returns (uint value) {
        return _balances[who];
    }

    function transfer( address to, uint value) returns (bool ok) {

        if( _balances[msg.sender] < value ) {
            throw;
        }
```

```
        _balances[msg.sender] -= value;
        _balances[to] += value;

        Transfer( msg.sender, to, value );

        return true;
    }
}
```

Running `dapp build` will produce three output files:

```
$ dapp build
$ ls out
DSTokenBase.abi          DSTokenBase.bin          DSTokenBase.bin-runtime
```

The `abi` file is a JSON representation of your contract's Application Binary Interface and in this case it will look like this:

```
[
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "name": "from",
        "type": "address"
      },
      {
        "indexed": true,
        "name": "to",
        "type": "address"
      },
      {
        "indexed": false,
        "name": "value",
        "type": "uint256"
      }
    ],
    "name": "Transfer",
    "type": "event"
  },
  {
    "constant": true,
    "inputs": [
      {
        "name": "who",
        "type": "address"
      }
    ],
    "name": "balanceOf",
    "outputs": [
      {
        "name": "value",
        "type": "uint256"
      }
    ],
    "payable": false,
    "type": "function"
```

```
  },
  {
    "constant": false,
    "inputs": [
      {
        "name": "to",
        "type": "address"
      },
      {
        "name": "value",
        "type": "uint256"
      }
    ],
    "name": "transfer",
    "outputs": [
      {
        "name": "ok",
        "type": "bool"
      }
    ],
    "payable": false,
    "type": "function"
  }
]
```

This file specifies the public functions of your contract, which your user interface and other applications on the blockchain use to interact with it.

The other two files that were created carry the `bin` and `bin-runtime` file extensions. The `bin` file is the actual bytecode that gets combined with your encoded constructor arguments and included in the `calldata` of any contract creating transactions.

When the EVM evaluates your contract creating transaction, your contract's constructor function is called. The return value of this constructor function is the actual contract that you will interact with on the blockchain, and it is somewhat different that the bytecode that was originally found in your `bin` file. For instance, the return value of the constructor function does not in turn include bytecode instructions for a constructor function. This return value can be known in advance when compiling your contract, and its value is included in the `bin-runtime` file.

CHAPTER 5

Unit Testing

Dapp uses ethrun and ds-test to test your code natively in Solidity. Writing both source code and unit tests in Solidity reduces context-switching, creating a more pleasant experience for the smart contract developer.

Getting Started

First, make sure you are comfortable *installing packages using dapp*. Next, you will need to install the `ds-test` Solidity package in your project:

```
$ dapp install ds-test
```

Note: Running `dapp init` will install `ds-test` for you, making this step unnecessary in most cases.

To create a test contract, you simply import `test.sol` from `ds-test` and make your contract inherit from `DSTest` like so:

```
pragma solidity ^0.4.0;

import "ds-test/test.sol";

contract TestContract is DSTest {}
```

General Testing

Unit testing with dapp should feel familiar to anyone who has used a test harness in other high-level programming languages. The `DSTest` parent class provides assertion functions for validating correctness and events for logging data in the console. These are detailed in the API reference guide *below*. Other than those functions, there are only a few basic conventions that you will need to learn to run your tests:

setUp Function

The `setUp` function is inherited as a placeholder from `DSTest`. You can override it and add configuration steps that will be executed before each of your test cases.

```
pragma solidity ^0.4.0;

import "ds-test/test.sol";
import "token.sol";

contract AppTokenTest is DSTest {

    AppToken token;

    // token will be instantiated before each test case
    function setUp() {
        token = new AppToken();
    }
}
```

Testcase Naming

Any functions that begin with the prefix `test` will be evaluated for correctness. In the general case, a testcase is considered correct if it doesn't throw an exception. Additionally, functions that begin with the prefix `testFail` are considered correct if they *do* throw an exception.

```
pragma solidity ^0.4.0;

import "ds-test/test.sol";
import "token.sol";

// A contract that can receive and transfer tokens.
// Useful for testing a system with multiple users.
contract TokenUser {
    AppToken  token;

    function TokenUser(AppToken token_) {
        token = token_;
    }

    function doTransfer(address to, uint amount)
        returns (bool)
    {
        return token.transfer(to, amount);
    }
}

contract AppTokenTest is DSTest {

    TokenUser user;
    AppToken token;

    // token and user will be instantiated before each test case
    function setUp() {
        token = new AppToken();
        user = new TokenUser(token);
```

```
    }

    function testTokenTransfer() {

        // inflate the supply of AppTokens
        token.mint(100);

        // test transfer to user
        assert(token.transfer(user, 100));

        // test transfer back
        assert(user.doTransfer(this, 100));
    }

    function testFailTokenTransfer() {

        // test transfering tokens with a balance of 0.
        // this should throw an exception, making this testcase correct.
        token.transfer(user, 10);
    }
}
```

Testing Events

Dapp can also help test the correctness of an emitted sequence of events. To use this feature, call the `expectEventsExact` function with the `address` of the contract you are testing. Then you simply emit the events that you're expecting, and dapp will ensure that they are called by your contract in the same order and with the same data. This means that in your source code, you will need to split out your event definitions into their own contracts so that your test contracts can inherit from them and have access to the right events.

Here is an example `token.sol` file:

```
pragma solidity ^0.4.0;

import "ds-token/token.sol";

contract AppTokenEvents {

    event BuyerSet(address account, bool value);
    event SellerSet(address account, bool value);
}

contract AppToken is DSToken, AppTokenEvents {

    mapping(address => bool) buyers;
    mapping(address => bool) sellers;

    function setBuyer(address buyer, bool value) {
        buyers[buyer] = value;
        BuyerSet(buyer, value);
    }

    function setSeller(address seller, bool value) {
        sellers[seller] = value;
        SellerSet(seller, value);
```

```
    }
}
```

And here is a contract to test it

```
pragma solidity ^0.4.0;

import "ds-test/test.sol";
import "token.sol";

contract TokenUser {}

contract AppTokenTest is DSTest, AppTokenEvents {

    TokenUser user;
    AppToken token;

    function setUp() {
        user = new TokenUser();
        token = new AppToken();
    }

    // This test will pass
    function testEvents() {
        expectEventsExact(token);
        BuyerSet(user, true);
        BuyerSet(user, false);
        SellerSet(this, true);

        token.setBuyer(user, true);
        token.setBuyer(user, false);

        token.setSeller(this, true);
    }

    // This test will fail
    function testEventsIncorrect() {
        expectEventsExact(token);
        BuyerSet(user, true);
        SellerSet(this, true);

        token.setBuyer(user, true);
        token.setBuyer(user, false);

        token.setSeller(this, true);
    }
}
```

API Reference

These are the modifiers, functions, and events that are available when your test contract inherits from `DSTest`, and the different flags available when running `dapp test`

Function Modifiers

Modifier	Description
logs_gas( )	Logs the amount of gas that your test case consumes

Assertions Functions

Function
assert(bool condition)
assertEq(address a, address b)
assertEq(bytes32 a, bytes32 b)
assertEq(int a, int b)
assertEq(uint a, uint b)
assertEq0(bytes a, bytes b)
expectEventsExact(address target)

Logging Events

These events will log information to the console, making them useful for debugging.

Event
logs(bytes)
log_bytes32(bytes32)
log_named_bytes32(bytes32 key, bytes32 val)
log_named_address(bytes32 key, address val)
log_named_int(bytes32 key, int val)
log_named_uint(bytes32 key, uint val)
log_named_decimal_int(bytes32 key, int val, uint decimals)
log_named_decimal_uint(bytes32 key, uint val, uint decimals)

Dapp Test Flags

Flag	Description
-v	Logs events to the command line for failing tests
-r <REGEX>	Only run tests who's name matches the provided regular expression

CHAPTER 6

Creating Contracts

Creating new contracts is kept very simple in dapp. The command is invoked with `dapp create` followed by the contract's type and any constructor parameters.

So if you had this contract:

```
pragma solidity ^0.4.0;

contract AppToken {

    mapping( address => uint ) _balances;
    string public name;

    function AppToken(uint initial_balance, string name_) {
        _balances[msg.sender] = initial_balance;
        _supply = initial_balance;

        name = name_;
    }
}
```

You would first turn on an Ethereum client and then create it like this:

```
$ dapp create AppToken 10000 "MyCoolToken"
+ seth send --create out/AppToken.bin 'AppToken(uint256,string)' 10000 MyCoolToken
seth-send: Published transaction with 1688 bytes of calldata.
seth-send: 0x4df5fdad614a88702439b10dd3002adb7869b339413011ba8ac69b4f07d9f10d
seth-send: Waiting for transaction receipt...
seth-send: Transaction included in block .
```

And that's it! Dapp will read the correct bytecode file from your `out` directory, bundle it up into a transaction with your encoded constructor parameters, and broadcast it to the network by curling your Ethereum client.

Storing Your Contract Addresses

Coming soon!