Truffle documentation

Oct 20, 2020

Contents

A world class development environment, testing framework and asset pipeline for blockchains using the PlatON 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.
- Interactive console for direct contract communication.
- Configurable build pipeline with support for tight integration.
- External script runner that executes scripts within a Truffle environment.

CHAPTER 1

Truffle document content

Truffle document contains 5 parts: Quick Start, Basic Functions, Writing Test Cases, Advanced Usage, References, The following is an outline:

1.1 Truffle QuickStart

This page will take you through the basics of creating a platon truffle project and deploying a smart contract to a blockchain.

1.1.1 Creating a project

To use most platon truffle commands, you need to run them against an existing platon truffle project. So the first step is to create a platon truffle project.

You can create a bare project template, but for those just getting started, you can use `platon-truffle Boxes`, which are example applications and project templates. We'll use the `MetaCoin box`, which creates a token that can be transferred between accounts:

1. Create a new directory for your platon truffle project:

```
mkdir MetaCoin
cd MetaCoin
```

2. Download ("unbox") the MetaCoin box:

```
platon-truffle unbox metacoin
```

Note: You can use the *platon-truffle unbox <box-name>* command to download any of the other platon truffle Boxes.

Note: To create a bare platon truffle project with no smart contracts included, use *platon-truffle init*.

Once this operation is completed, you'll now 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*

1.1.2 Exploring the project

Note: This page is just a quickstart, so we're not going to go into much detail here. Please see the rest of the platon truffle documentation to learn more.

1. Open the `contracts/MetaCoin.sol` file in a text editor. This is a smart contract (written in Solidity) that creates a MetaCoin token. Note that this also references another Solidity file `contracts/ConvertLib.sol` in the same directory.
2. Open the `contracts/Migrations.sol` file. This is a separate Solidity file that manages and updates *the status of your deployed smart contract*. This file comes with every platon truffle project, and is usually not edited.
3. Open the `migrations/1_initial_migration.js` file. This file is the migration (deployment) script for the `Migrations` contract found in the `Migrations.sol` file.
4. Open the `migrations/2_deploy_contracts.js` file. This file is the migration script for the `MetaCoin` contract. (Migration scripts are run in order, so the file beginning with `2` will be run after the file beginning with `1`.)
5. Open the `test/TestMetacoin.sol` file. This is a *test file written in Solidity* which ensures that your contract is working as expected.
6. Open the `test/metacoin.js` file. This is a *test file written in JavaScript* which performs a similar function to the Solidity test above.
7. Open the `truffle-config.js` file. This is the platon truffle *configuration file*, for setting network information and other project-related settings. The file is blank, but this is okay, as we'll be using a platon truffle command that has some defaults built-in.

1.1.3 Testing

1. On a terminal, run the Solidity test:

```
platon-truffle test ./test/TestMetacoin.sol
```

You will see the following output

```
TestMetacoin
   testInitialBalanceUsingDeployedContract (71ms)
   testInitialBalanceWithNewMetaCoin (59ms)

2 passing (794ms)
```

Note: If you're on Windows and encountering problems running this command, please see the documentation on resolving naming conflicts on Windows.

These tree tests were run against the contract, with descriptions displayed on what the tests are supposed to do.

1. Run the JavaScript test:

```
platon-truffle test ./test/metacoin.js
```

You will see the following output

```
  Contract: MetaCoin
     should put 10000 MetaCoin in the first account
     should call a function that depends on a linked library (40ms)
     should send coin correctly (129ms)

  3 passing (255ms)
```

1.1.4 Compiling

1. Compile the smart contracts:

```
platon-truffle compile
```

You will see the following output:

```
Compiling .\contracts\ConvertLib.sol...
Compiling .\contracts\MetaCoin.sol...
Compiling .\contracts\Migrations.sol...

Writing artifacts to .\build\contracts
```

1.1.5 Migrating

To deploy our smart contracts, we're going to need to connect to a blockchain. platon truffle has a built-in personal blockchain that can be used for testing. This blockchain is local to your system and does not interact with the main PlatON network.

You can create this blockchain and interact with it using scripts/node/start.sh shell scripts.

1. Run platon truffle migrate:

```
platon-truffle migrate
```

```
Starting migrations...
======================
> Network name:    'develop'
> Network id:      4447
> Block gas limit: 6721975

1_initial_migration.js
======================
```

(continues on next page)

(continued from previous page)

```
   Deploying 'Migrations'
   ----------------------
   > transaction hash:    
↪0x3fd222279dad48583a3320decd0a2d12e82e728ba9a0f19bdaaff98c72a030a2
   > Blocks: 0            Seconds: 0
   > contract address:    0xa0AdaB6E829C818d50c75F17CFCc2e15bfd55a63
   > account:             0x627306090abab3a6e1400e9345bc60c78a8bef57
   > balance:             99.99445076
   > gas used:            277462
   > gas price:           20 gvon
   > value sent:          0 LAT
   > total cost:          0.00554924 LAT

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00554924 LAT

2_deploy_contracts.js
=====================

   Deploying 'ConvertLib'
   ----------------------
   > transaction hash:    
↪0x97e8168f1c05fc40dd8ffc529b9a2bf45cc7c55b07b6b9a5a22173235ee247b6
   > Blocks: 0            Seconds: 0
   > contract address:    0xfb39FeaeF3ac3fd46e2123768e559BCe6bD638d6
   > account:             0x627306090abab3a6e1400e9345bc60c78a8bef57
   > balance:             99.9914458
   > gas used:            108240
   > gas price:           20 gvon
   > value sent:          0 LAT
   > total cost:          0.0021648 LAT

   Linking
   -------
   * Contract: MetaCoin <--> Library: ConvertLib (at address: 
↪0xfb39FeaeF3ac3fd46e2123768e559BCe6bD638d6)

   Deploying 'MetaCoin'
   --------------------
   > transaction hash:    
↪0xee4994097c10e7314cc83adf899d67f51f22e08b920e95b6d3f75c5eb498bde4
   > Blocks: 0            Seconds: 0
   > contract address:    0x6891Ac4E2EF3dA9bc88C96fEDbC9eA4d6D88F768
   > account:             0x627306090abab3a6e1400e9345bc60c78a8bef57
   > balance:             99.98449716
   > gas used:            347432
   > gas price:           20 gvon
   > value sent:          0 LAT
   > total cost:          0.00694864 LAT

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00911344 LAT
```

(continues on next page)

(continued from previous page)

```
Summary
=======
> Total deployments:   3
> Final cost:          0.01466268 LAT
```

This shows the transaction IDs and addresses of your deployed contracts. It also includes a cost summary and real-time status updates.

Note: Your transaction hashes, contract addresses, and accounts will be different from the above.

Note: To see how to interact with the contract, please skip to the next section.

1.1.6 Interacting with the contract

To interact with the contract, you can use the platon truffle console. The platon truffle console is similar to platon truffle Develop, except it connects to an existing blockchain.

```
platon-truffle console
```

You will see the following prompt:

```
truffle(development)>
```

Note: Console prompt: truffle (development)> The brackets refer to the currently connected network.

Interact with the contract using the console in the following ways:

As of platon truffle v5, the console supports async/await functions, enabling much simpler interactions with the contract.

- Begin by establishing both the deployed MetaCoin contract instance and the accounts created by either platon truffle's built-in blockchain:

```
truffle(development)> let instance = await MetaCoin.deployed()
truffle(development)> let accounts = await web3.platon.getAccounts()
```

- Check the metacoin balance of the account that deployed the contract:

```
truffle(development)> let balance = await instance.getBalance(accounts[0])
truffle(development)> balance.toNumber()
```

- Transfer some metacoin from one account to another:

```
truffle(development)> instance.sendCoin(accounts[1], 500)
```

- Check the balance of the account that received the metacoin:

```
truffle(development)> let received = await instance.getBalance(accounts[1])
truffle(development)> received.toNumber()
```

- Check the balance of the account that sent the metacoin:

```
truffle(development)> let newBalance = await instance.getBalance(accounts[0])
truffle(development)> newBalance.toNumber()
```

1.1.7 Continue learning

This quickstart showed you the basics of the platon truffle project lifecycle, but there is much more to learn. Please continue on with the rest of our documentation and especially our tutorials to learn more.

1.2 Installation

```
$ npm install -g platon-truffle
$ platon-truffle version
```

1.2.1 Requirements

- NodeJS v10.12.0 or later
- Ubuntu16.04 or later

platon-truffle also requires that you have a running PlatON client which supports the standard JSON RPC API (which is nearly all of them). There are many to choose from, and some better than others for development. We'll discuss them in detail in the Choosing an PlatON client section.

1.3 Creating a project

To use most platon truffle commands, you need to run them against an existing platon truffle project. So the first step is to create a platon truffle project.

You can create a bare project template, but for those just getting started, you can use `platon truffle Boxes`, which are example applications and project templates. We'll use the `MetaCoin box`, which creates a token that can be transferred between accounts:

1. Create a new directory for your platon truffle project:

   ```
   mkdir MetaCoin
   cd MetaCoin
   ```

2. Download ("unbox") the MetaCoin box:

   ```
   platon-truffle unbox metacoin
   ```

Note: You can use the platon-truffle unbox <box-name> command to download any of the other platon truffle Boxes.

Note: To create a bare platon truffle project with no smart contracts included, use platon-truffle init.

Note: You can use an optional –force to initialize the project in the current directory regardless of its state (e.g. even if it contains other files or directories). This applies to both the init and unbox commands. Be careful, this will potentially overwrite files that exist in the directory.

Once this operation is completed, you'll now 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`: platon truffle *configuration file*

1.4 Compiling Contracts

1.4.1 Location

All of your contracts are located in your project's `contracts/` directory. As contracts are written in `Solidity` or `cpp(wasm contract)`, all files containing contracts will have a file extension of `.sol` or `cpp`

With a bare platon truffle *project* (created through `platon-truffle init`), you're given a single `Migrations.sol` file that helps in the deployment process. If you're using a `platon-truffle Box`, you will have multiple files here.

1.4.2 Command

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

```
platon-truffle compile
```

Upon first run, all contracts will be compiled. Upon subsequent runs, platon 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.

1.4.3 Build artifacts

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

These artifacts are integral to the inner workings of platon truffle, and they play an important part in the successful deployment of your application. `You should not edit these files` as they'll be overwritten by contract compilation and deployment.

1.4.4 Dependencies

You can declare contract dependencies using Solidity's `import` command. platon truffle will compile contracts in the correct order and ensure all dependencies are sent to the compiler. Dependencies can be specified in two ways:

Importing dependencies via file name

To import contracts from a separate file, add the following code to your Solidity source file:

```
import "./AnotherContract.sol";
```

This will make all contracts within `AnotherContract.sol` available. Here, `AnotherContract.sol` is relative to the path of the current contract being written.

Note that Solidity allows other import syntaxes as well. See the Solidity `import documentation` for more information.

1.5 Running migrations

Migrations are JavaScript files that help you deploy contracts to the PlatON network. These files are responsible for staging your deployment tasks, and they're written under the assumption that your 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.

1.5.1 Command for solidity

To run your migrations, run the following:

```
$ platon-truffle migrate
```

This will run all migrations located within your project's `migrations` directory. At their simplest, migrations are simply a set of managed deployment scripts. If your migrations were previously run successfully, `platon-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. For local testing make sure to have a test blockchain and running before executing `migrate`.

1.5.2 Migration files

A simple migration file looks like this: Filename: `4_example_migration.js`

```
var MyContract = artifacts.require("XlbContract");

module.exports = function(deployer) {
  // deployment steps
  deployer.deploy(MyContract);
};
```

Note that the filename is prefixed with a number and is suffixed by a description. The numbered prefix is required in order to record whether the migration ran successfully. The suffix is purely for human readability and comprehension.

artifacts.require()

At the beginning of the migration, we tell Truffle which contracts we'd like to interact with via the `artifacts.require()` method. This method is similar to Node's `require`, but in our case it specifically returns a contract abstraction that we can use within the rest of our deployment script. The name specified should match `the name` of the contract definition within that source file. Do not pass the name of the source file, as files can contain more than one contract.

Consider this example where two contracts are specified within the same source file:

Filename: ./contracts/Contracts.sol

```
contract ContractOne {
  // ...
}

contract ContractTwo {
  // ...
}
```

To use only ContractTwo, your artifacts.require() statement would look like this:

```
var ContractTwo = artifacts.require("ContractTwo");
```

To use both contracts, you will need two artifacts.require() statements:

```
var ContractOne = artifacts.require("ContractOne");
var ContractTwo = artifacts.require("ContractTwo");
```

module.exports

All migrations must export a function via the module.exports syntax. The function exported by each migration should accept a deployer object as its first parameter. This object aides in deployment by both providing a clear syntax for deploying smart contracts as well as performing some of deployment's more mundane duties, such as saving deployed artifacts for later use. The deployer object is your main interface for staging deployment tasks, and its API is described at the bottom of this page.

Your migration function can accept other parameters as well. See the examples below.

1.5.3 Initial migration

platon-truffle requires you to have a Migrations contract in order to use the Migrations feature. This contract must contain a specific interface, but you're free to edit this contract at will. For most projects, this contract will be deployed initially as the first migration and won't be updated again. You will also receive this contract by default when creating a new project with truffle init.

FileName: contracts/Migrations.sol

```
pragma solidity >=0.4.8 <0.5.13;

contract Migrations {
  address public owner;

  // A function with the signature `last_completed_migration()`, returning a uint, is␣
↪required.
  uint public last_completed_migration;

  modifier restricted() {
    if (msg.sender == owner) _;
  }

```

(continues on next page)

(continued from previous page)

```
  function Migrations() {
    owner = msg.sender;
  }

  // A function with the signature `setCompleted(uint)` is required.
  function setCompleted(uint completed) restricted {
    last_completed_migration = completed;
  }

  function upgrade(address new_address) restricted {
    Migrations upgraded = Migrations(new_address);
    upgraded.setCompleted(last_completed_migration);
  }
}
```

You must deploy this contract inside your first migration in order to take advantage of the Migrations feature. To do so, create the following migration:

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);
};
```

From here, you can create new migrations with increasing numbered prefixes to deploy other contracts and perform further deployment steps.

1.5.4 Deployer

Your migration files will use the deployer to stage deployment tasks. As such, you can write deployment tasks synchronously and they'll be executed in the correct order:

```
// Stage deploying A before B
deployer.deploy(A);
deployer.deploy(B);
```

Alternatively, each function on the deployer can be used as a Promise, to queue up deployment tasks that depend on the execution of the previous task:

```
// Deploy A, then deploy B, passing in A's newly deployed address
deployer.deploy(A).then(function() {
  return deployer.deploy(B, A.address);
});
```

It is possible to write your deployment as a single promise chain if you find that syntax to be more clear. The deployer API is discussed at the bottom of this page.

1.5.5 Network considerations

It is possible to run deployment steps conditionally based on the network being deployed to. This is an advanced feature, so see the *Networks* section first before continuing.

To conditionally stage deployment steps, write your migrations so that they accept a second parameter, called `network`. Example:

```
module.exports = function(deployer, network) {
  if (network == "live") {
    // Do something specific to the network named "live".
  } else {
    // Perform a different step otherwise.
  }
}
```

1.5.6 Available accounts

Migrations are also passed the list of accounts provided to you by your PlatON client and web3 provider, for you to use during your deployments. This is the exact same list of accounts returned from `web3.platon.getAccounts()`.

```
module.exports = function(deployer, network, accounts) {
  // Use the accounts within your migrations.
}
```

1.5.7 Deployer API

The deployer contains many functions available to simplify your migrations.

deployer.deploy(contract, args. . . , options)

Deploy a specific contract, specified by the contract object, with optional constructor arguments. This is useful for singleton contracts, such that only one instance of this contract exists for your dapp. This will set the address of the contract after deployment (i.e., `Contract.address` will equal the newly deployed address), and it will override any previous address stored.

You can optionally pass an array of contracts, or an array of arrays, to speed up deployment of multiple contracts. Additionally, the last argument is an optional object that can include the key named `overwrite` as well as other transaction parameters such as `gas` and `from`. If `overwrite` is set to `false`, the deployer won't deploy this contract if one has already been deployed. This is useful for certain circumstances where a contract address is provided by an external dependency.

Note that you will need to deploy and link any libraries your contracts depend on first before calling `deploy`. See the `link` function below for more details.

Examples:

```
// Deploy a single contract without constructor arguments
deployer.deploy(A);

//  Deploy a single contract with constructor arguments
deployer.deploy(A, arg1, arg2, ...);

// Don't deploy this contract if it has already been deployed
deployer.deploy(A, {overwrite: false});

// Set a maximum amount of gas and `from` address for the deployment
deployer.deploy(A, {gas: 4612388, from: "0x...."});
```

(continues on next page)

(continued from previous page)

```
// Deploying multiple contracts as an array is now deprecated.
// This used to be quicker than writing three `deployer.deploy()` statements as the␣
↪deployer
// can perform the deployment as a single batched request.
// deployer.deploy([
//   [A, arg1, arg2, ...],
//   B,
//   [C, arg1]
// ]);

// External dependency example:
//
// For this example, our dependency provides an address when we're deploying to the
// live network, but not for any other networks like testing and development.
// When we're deploying to the live network we want it to use that address, but in
// testing and development we need to deploy a version of our own. Instead of writing
// a bunch of conditionals, we can simply use the `overwrite` key.
deployer.deploy(SomeDependency, {overwrite: false});
```

deployer.link(library, destinations)

Link an already-deployed library to a contract or multiple contracts. `destinations` can be a single contract or an array of multiple contracts. If any contract within the destination doesn't rely on the library being linked, the contract will be ignored.

Example:

```
// Deploy library LibA, then link LibA to contract B, then deploy B.
deployer.deploy(LibA);
deployer.link(LibA, B);
deployer.deploy(B);

// Link LibA to many contracts
deployer.link(LibA, [B, C, D]);
```

deployer.then(function() {. . . })

Just like a promise, run an arbitrary deployment step. Use this to call specific contract functions during your migration to add, edit and reorganize contract data.

Example:

```
var a, b;
deployer.then(function() {
  // Create a new version of A
  return A.new();
}).then(function(instance) {
  a = instance;
  // Get the deployed instance of B
  return B.deployed();
}).then(function(instance) {
  b = instance;
  // Set the new instance of A's address on B via B's setA() function.
```

(continues on next page)

(continued from previous page)

```
  return b.setA(a.address);
});
```

1.5.8 Command for wasm

To run your wasm migrations, for example: there is a test.cpp contract file in the contracts directory, run the following:

```
$ platon-truffle migrate --wasm --contract-name test
```

If you want to deploy all wasm contract, run the following:

```
$ platon-truffle migrate --wasm
```

1.6 Interacting with your contracts

1.6.1 Introduction

If you were writing raw requests to the PlatON network yourself in order to interact with your contracts, you'd soon realize that writing these requests is clunky and cumbersome. As well, you might find that managing the state for each request you've made is complicated. Fortunately, platon truffle takes care of this complexity for you, to make interacting with your contracts a breeze.

1.6.2 Reading and writing data

The PlatON network makes a distinction between writing data to the network and reading data from it, and this distinction plays a significant part in how you write your application. In general, writing data is called a `transaction` whereas reading data is called a `call`. Transactions and calls are treated very differently, and have the following characteristics.

Transactions

`Transactions` fundamentally change the state of the network. A `transaction` can be as simple as sending Lat to another account, or as complicated as executing a contract function or adding a new contract to the network. The defining characteristic of a `transaction` is that it writes (or changes) data. `Transactions` cost Lat to run, known as "gas", and `transactions` take time to process. When you execute a contract's function via a `transaction`, you cannot receive that function's return value because the `transaction` isn't processed immediately. In general, functions meant to be executed via a `transaction` will not return a value; they will return a `transaction` id instead. So in summary, `transactions`:

- Cost gas (Lat)
- Change the state of the network
- Aren't processed immediately
- Won't expose a return value (only a transaction id).

Calls

`Calls`, on the other hand, are very different. `Calls` can be used to execute code on the network, though no data will be permanently changed. `Calls` are free to run, and their defining characteristic is that they read data. When you execute a contract function via a call you will receive the return value immediately. In summary, `calls`:

- Are free (do not cost gas)
- Do not change the state of the network
- Are processed immediately
- Will expose a return value (hooray!)

Choosing between a `transaction` and a `call` is as simple as deciding whether you want to read data, or write it.

1.6.3 Introducing abstractions

Contract abstractions are the bread and butter of interacting with PlatON contracts from Javascript. In short, contract abstractions are wrapper code that makes interaction with your contracts easy, in a way that lets you forget about the many engines and gears executing under the hood. platon truffle uses its own contract abstraction via the `truffle-contract` module, and it is this contract abstraction that's described below.

In order to appreciate the usefulness of a contract abstraction, however, we first need a contract to talk about. We'll use the MetaCoin contract available to you through platon truffle Boxes via `platon-truffle unbox metacoin`.

```
pragma solidity >=0.4.25 <0.5.13;

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);

        constructor() public {
                balances[tx.origin] = 10000;
        }

        function sendCoin(address receiver, uint amount) public returns(bool␣
↪sufficient) {
                if (balances[msg.sender] < amount) return false;
                balances[msg.sender] -= amount;
                balances[receiver] += amount;
        emit Transfer(msg.sender, receiver, amount);
                return true;
        }

        function getBalance(address addr) public view returns(uint) {
                return balances[addr];
        }
}
```

This contract has three methods aside from the constructor (`sendCoin` and `getBalance`). All three methods can be executed as either a transaction or a call.

Now let's look at the Javascript object called `MetaCoin` provided for us by platon truffle, as made available in the `platon-truffle console`:

```
truffle(develop)> let instance = await MetaCoin.deployed()
truffle(develop)> instance

// outputs:
//
// Contract
// - address: "0xa9f441a487754e6b27ba044a5a8eb2eec77f6b92"
// - allEvents: ()
// - getBalance: ()
// - sendCoin: ()
// ...
```

Notice that the abstraction contains the exact same functions that exist within our contract. It also contains an address which points to the deployed version of the MetaCoin contract.

1.6.4 Executing contract functions

Using the abstraction you can easily execute contract functions on the PlatON network.

Making a transaction

There are three functions on the MetaCoin contract that we can execute. If you analyze each of them, you'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:

```
truffle(develop)> let accounts = await web3.platon.getAccounts()
truffle(develop)> instance.sendCoin(accounts[1], 10, {from: accounts[0]})
```

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.
- 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 ("transaction params"). Here, we set the `from` address ensuring this transaction came from `accounts[0]`. The transaction params that you can set correspond to the fields in an PlatON transaction:
  - `from`
  - `to`
  - `gas`
  - `gasPrice`
  - `value`
  - `data`

– nonce

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:

```
truffle(develop)> let balance = await instance.getBalance(accounts[0])
truffle(develop)> balance.toNumber()
```

What's interesting here:

- We received a return value. Note that since the PlatON network can handle very large numbers, we're given a BN object which we then convert to a number.

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

Processing transaction results

When you make a transaction, you're given a `result` object that gives you a wealth of information about the transaction.

```
truffle(develop)> let result = await contract.sendCoin(accounts[1], 10, {from:␣
↪accounts[0]})
truffle(develop)> result
```

Specifically, you get the following:

- `result.tx` *(string)* - Transaction hash
- `result.logs` *(array)* - Decoded events
- `result.receipt` *(object)* - Transaction receipt (includes the amount of gas used)

Catching events

Your contracts can fire events that you can catch to gain more insight into what your contracts are doing. The easiest way to handle events is by processing the `logs` array contained within `result` object of the transaction that triggered the event.

If we explicitly output the first log entry we can see the details of the event that was emitted as part of the `sendCoin` call (`Transfer(msg.sender, receiver, amount);`).

```
truffle(develop)> result.logs[0]
{ logIndex: 0,
  transactionIndex: 0,
  transactionHash: '0x3b33960e99416f687b983d4a6bb628d38bf7855c6249e71d0d16c7930a588cb2
↪',
  blockHash: '0xe36787063e114a763469e7dabc7aa57545e67eb2c395a1e6784988ac065fdd59',
  blockNumber: 8,
  address: '0x6891Ac4E2EF3dA9bc88C96fEDbC9eA4d6D88F768',
```

(continues on next page)

(continued from previous page)

```
  type: 'mined',
  id: 'log_3181e274',
  event: 'Transfer',
  args:
   Result {
     '0': '0x8128880DC48cde7e471EF6b99d3877357bb93f01',
     '1': '0x12B6971f6eb35dD138a03Bd6cBdf9Fc9b9a87d7e',
     '2': <BN: a>,
     __length__: 3,
     _from: '0x8128880DC48cde7e471EF6b99d3877357bb93f01',
     _to: '0x12B6971f6eb35dD138a03Bd6cBdf9Fc9b9a87d7e',
     _value: <BN: a> } }
```

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:

```
truffle(develop)> let newInstance = await MetaCoin.new()
truffle(develop)> newInstance.address
'0x64307b67314b584b1E3Be606255bd683C835A876'
```

Use a contract at a specific address

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

```
let specificInstance = await MetaCoin.at("0x1234...");
```

Sending lat to a contract

You may simply want to send Lat directly to a contract, or trigger a contract's 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.platon.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 Lat directly:

```
instance.send(web3.toVon(1, "lat")).then(function(result) {
  // Same result object as above.
});
```

1.6.5 Special methods on Truffle contract objects

There are a couple of special functions that you can find on the actual contract methods of your contract abstractions:

- `estimateGas`
- `sendTransaction`
- `call`

The first special method mentioned above is the `estimateGas` method. This, as you probably can guess, estimates the amount of gas that a transaction will require. If we wanted to estimate the gas for a transaction, we would call it on the contract method itself. It would look something like the following:

```
const instance = await MyContract.deployed();
const amountOfGas = await instance.sendTokens.estimateGas(4, myAccount);
```

This will give us an estimate of how much gas it will take to run the transaction specified.

Note that the arguments above (`4` and `myAccount`) correspond to whatever the signature of the contract method happens to be.

Another useful thing to note is that you can also call this on a contract's new method to see how much gas it will take to deploy. So you would do `Contract.new.estimateGas()` to get the gas estimate for the contract's deployment.

The next mentioned method is `sendTransaction`. In general, if you execute a contract method, platon truffle will intelligently figure out whether it needs to make a transaction or a call. If your function can be executed as a call, then platon truffle will do so and you will be able to avoid gas costs.

There may be some scenarios, however, where you want to force platon truffle to make a transaction. In these cases, you can use the `sendTransaction` method found on the method itself. This would look something like `instance.myMethod.sendTransaction()`.

For example, suppose I have a contract instance with the method `getTokenBalance`. I could do the following to force a transaction to take place while executing `getTokenBalance`:

```
const instance = await MyContract.deployed();
const result = await instance.getTokenBalance.sendTransaction(myAccount);
```

The `result` variable above will be the same kind of result you would get from executing any normal transaction in platon truffle. It will contain the transaction hash, the logs, etc.

The last method is `call` and the syntax is exactly the same as for `sendTransaction`. If you want to explicitly make a call, you can use the `call` method found on your contract abstraction's method. So you would write something that looks like const `result = await instance.myMethod.call()`.

1.6.6 Further reading

The contract abstractions provided by platon truffle contain a wealth of utilities for making interacting with your contracts easy. Check out the `truffle-contract` documentation for tips, tricks and insights.

1.7 Using platon-truffle console

Sometimes it's nice to work with your contracts interactively for testing and debugging purposes, or for executing transactions by hand. platon truffle provides you one easy way to do this via an interactive console, with your contracts available and ready to use.

- **platon-truffle console**: A basic interactive console connecting to any PlatON client

1.7.1 Why console?

Reasons to use **platon-truffle console**:

- You have a client you're already using, such as platon
- You want to migrate to a testnet (or the main PlatON network)
- You want to use a specific mnemonic or account list

1.7.2 Commands

All commands require that you be in your project folder. You do not need to be at the root.

Console

To launch the console:

```
truffle console
```

This will look for a network definition called `development` in the configuration, and connect to it, if available. You can override this using the `--network <name>` option or customize the `development` network settings. See more details in the Networks section as well as the *command reference*.

When you load the console, you'll immediately see the following prompt:

```
truffle(development)>
```

This tells you you're running within a platon truffle console using the `development` network.

1.7.3 Features

Both platon truffle Develop and the console provide most of the features available in the platon truffle command line tool. For instance, you can type `migrate --reset` within the console, and it will be interpreted the same as if you ran `platon-truffle migrate --reset` on the command line.

Additionally, both platon truffle Develop and the console have the following features:

- All of your compiled contracts are available and ready for use.
- After each command (such as `migrate --reset`) your contracts are reprovisioned so you can start using the newly assigned addresses and binaries immediately.
- The `web3` library is made available and is set to connect to your PlatON client.

Commands available

- `init`
- `compile`
- `deploy`
- `exec`
- `help`
- `migrate`

- `networks`
- `opcode`
- `test`
- `version`

If a platon truffle command is not available, it is because it is not relevant for an existing project (for example, `init`) or wouldn't make sense (for example, `console`).

See full *command reference* for more information.

1.8 Writing external scripts

Often you may want to run external scripts that interact with your contracts. platon truffle provides an easy way to do this, bootstrapping your contracts based on your desired network and connecting to your PlatON client automatically per your *project configuration*.

1.8.1 Command

To run an external script, perform the following:

```
$ platon-truffle exec <path/to/file.js>
```

1.8.2 File structure

In order for external scripts to be run correctly, platon truffle expects them to export a function that takes a single parameter as a callback:

```
module.exports = function(callback) {
  // perform actions
}
```

You can do anything you'd like within this script, so long as the callback is called when the script finishes. The callback accepts an error as its first and only parameter. If an error is provided, execution will halt and the process will return a non-zero exit code.

1.9 Testing your contracts

1.9.1 Framework

platon truffle comes standard with an automated testing framework to make testing your contracts a breeze. This framework lets you write simple and manageable tests in two different ways:

- In `Javascript` and `TypeScript`, for exercising your contracts from the outside world, just like your application.
- In `Solidity`, for exercising your contracts in advanced, bare-to-the-metal scenarios.

Both styles of tests have their advantages and drawbacks. See the next two sections for a discussion of each one.

1.9.2 Location

All solidity contract test files should be located in the `./test` directory. platon truffle will only run test files with the following file extensions: `.js`, `.ts`, `.es`, `.es6`, and `.jsx`, and `.sol`. All other files are ignored.

All wasm contract test files should be located in the *./test/wasm* directory

1.9.3 Command

To run all tests, simply run:

```
$ platon-truffle test
```

Alternatively, you can specify a path to a specific file you want to run, e.g.,

```
$ platon-truffle test ./path/to/test/file.js
```

You can also specify tests to run wasm contracts(contracts with null params to init)

```
$ platon-truffle test --wasm
```

You Can also specify specific wasm contracts to run test, If contract initialization parameters are not empty

```
$ platon-truffle test --wasm --contract-name ${ContractName} --params "[[], ]"
```

1.10 Writing solidity contract tests in javascript

platon truffle uses the Mocha testing framework and Chai for assertions to provide you with a solid framework from which to write your JavaScript tests. Let's dive in and see how platon truffle builds on top of Mocha to make testing your contracts a breeze.

Note: If you're unfamiliar with writing unit tests in Mocha, please see Mocha's documentation before continuing.

1.10.1 Use contract() instead of describe()

Structurally, your tests should remain largely unchanged from that of Mocha: Your tests should exist in the `./test` directory, they should end with a `.js` extension, and they should contain code that Mocha will recognize as an automated test. What makes platon truffle tests different from that of Mocha is the `contract()` function: This function works exactly like `describe()` except it enables platon truffle's clean-room features. It works like this:

- Before each `contract()` function is run, your contracts are redeployed to the running PlatON client so the tests within it run with a clean contract state.
- The `contract()` function provides a list of accounts made available by your PlatON client which you can use to write tests.

Since platon truffle uses Mocha under the hood, you can still use `describe()` to run normal Mocha tests whenever platon truffle clean-room features are unnecessary.

1.10.2 Use contract abstractions within your tests

Contract abstractions are the basis for making contract interaction possible from JavaScript (they're basically our flux capacitor). Because Truffle has no way of detecting which contracts you'll need to interact with within your tests, you'll need to ask for those contracts explicitly. You do this by using the `artifacts.require()` method, a method provided by Truffle that allows you to request a usable contract abstraction for a specific Solidity contract. As you'll see in the example below, you can then use this abstraction to make sure your contracts are working properly.

For more information on using contract abstractions, see the *Interacting With Your Contracts* section.

1.10.3 Using artifacts.require()

Using `artifacts.require()` within your tests works the same way as using it within your migrations; you just need to pass the name of the contract. See the artifacts.require() documentation in the Migrations section for detailed usage.

1.10.4 Using web3

A `web3` instance is available in each test file, configured to the correct provider. So calling `web3.platon.getBalance` just works!

1.10.5 Examples

Using `.then`

Here's an example test provided in the `MetaCoin Truffle Box`. Note the use of the `contract()` function, the `accounts` array for specifying available PlatON accounts, and our use of `artifacts.require()` for interacting directly with our contracts.

File: `./test/metacoin.js`

```
const MetaCoin = artifacts.require("MetaCoin");

contract("MetaCoin", accounts => {
  it("should put 10000 MetaCoin in the first account", () =>
    MetaCoin.deployed()
      .then(instance => instance.getBalance.call(accounts[0]))
      .then(balance => {
        assert.equal(
          balance.valueOf(),
          10000,
          "10000 wasn't in the first account"
        );
      }));

  it("should call a function that depends on a linked library", () => {
    let meta;
    let metaCoinBalance;
    let metaCoinLatBalance;

    return MetaCoin.deployed()
      .then(instance => {
        meta = instance;
```

(continues on next page)

(continued from previous page)

```
        return meta.getBalance.call(accounts[0]);
      })
      .then(outCoinBalance => {
        metaCoinBalance = outCoinBalance.toNumber();
        return meta.getBalanceInLat.call(accounts[0]);
      })
      .then(outCoinBalanceLat => {
        metaCoinLatBalance = outCoinBalanceLat.toNumber();
      })
      .then(() => {
        assert.equal(
          metaCoinLatBalance,
          2 * metaCoinBalance,
          "Library function returned unexpected function, linkage may be broken"
        );
      });
  });

  it("should send coin correctly", () => {
    let meta;

    // Get initial balances of first and second account.
    const account_one = accounts[0];
    const account_two = accounts[1];

    let account_one_starting_balance;
    let account_two_starting_balance;
    let account_one_ending_balance;
    let account_two_ending_balance;

    const amount = 10;

    return MetaCoin.deployed()
      .then(instance => {
        meta = instance;
        return meta.getBalance.call(account_one);
      })
      .then(balance => {
        account_one_starting_balance = balance.toNumber();
        return meta.getBalance.call(account_two);
      })
      .then(balance => {
        account_two_starting_balance = balance.toNumber();
        return meta.sendCoin(account_two, amount, { from: account_one });
      })
      .then(() => meta.getBalance.call(account_one))
      .then(balance => {
        account_one_ending_balance = balance.toNumber();
        return meta.getBalance.call(account_two);
      })
      .then(balance => {
        account_two_ending_balance = balance.toNumber();

        assert.equal(
          account_one_ending_balance,
          account_one_starting_balance - amount,
          "Amount wasn't correctly taken from the sender"
```

(continues on next page)

(continued from previous page)

```
      );
      assert.equal(
        account_two_ending_balance,
        account_two_starting_balance + amount,
        "Amount wasn't correctly sent to the receiver"
      );
    });
  });
});
```

This test will produce the following output:

```
  Contract: MetaCoin
    should put 10000 MetaCoin in the first account (83ms)
    should call a function that depends on a linked library (43ms)
    should send coin correctly (122ms)


  3 passing (293ms)
```

Using async/await

Here is a similar example, but using async/await notation:

```
const MetaCoin = artifacts.require("MetaCoin");

contract("2nd MetaCoin test", async accounts => {
  it("should put 10000 MetaCoin in the first account", async () => {
    let instance = await MetaCoin.deployed();
    let balance = await instance.getBalance.call(accounts[0]);
    assert.equal(balance.valueOf(), 10000);
  });

  it("should call a function that depends on a linked library", async () => {
    let meta = await MetaCoin.deployed();
    let outCoinBalance = await meta.getBalance.call(accounts[0]);
    let metaCoinBalance = outCoinBalance.toNumber();
    let outCoinBalanceLat = await meta.getBalanceInLat.call(accounts[0]);
    let metaCoinLatBalance = outCoinBalanceLat.toNumber();
    assert.equal(metaCoinLatBalance, 2 * metaCoinBalance);
  });

  it("should send coin correctly", async () => {
    // Get initial balances of first and second account.
    let account_one = accounts[0];
    let account_two = accounts[1];

    let amount = 10;

    let instance = await MetaCoin.deployed();
    let meta = instance;

    let balance = await meta.getBalance.call(account_one);
    let account_one_starting_balance = balance.toNumber();
```

(continues on next page)

(continued from previous page)

```
    balance = await meta.getBalance.call(account_two);
    let account_two_starting_balance = balance.toNumber();
    await meta.sendCoin(account_two, amount, { from: account_one });

    balance = await meta.getBalance.call(account_one);
    let account_one_ending_balance = balance.toNumber();

    balance = await meta.getBalance.call(account_two);
    let account_two_ending_balance = balance.toNumber();

    assert.equal(
      account_one_ending_balance,
      account_one_starting_balance - amount,
      "Amount wasn't correctly taken from the sender"
    );
    assert.equal(
      account_two_ending_balance,
      account_two_starting_balance + amount,
      "Amount wasn't correctly sent to the receiver"
    );
  });
});
```

This test will produce identical output to the previous example.

1.10.6 Specifying tests

You can limit the tests being executed to a specific file as follows:

```
platon-truffle test ./test/metacoin.js
```

See the full command reference for more information.

1.10.7 TypeScript File Support

platon truffle now supports tests saved as a `.ts` TypeScript file. Please see the *Writing Tests in JavaScript* guide for more information.

1.11 Writing wasm contract tests in javascript

1.11.1 Use contract() instead of describe()

Structurally, your tests should remain unchanged from that of Mocha: Your tests should exist in the `./test/wasm` directory, they should end with a `.js` extension

1.11.2 Using global contract object

The contract object uses the contract name as the key, and the contract object value contains the abi required for the deployment contract and the parameters for calling the sendTransaction deployment contract, so you can deploy the contract based on this. After the deployment contract is successful, you can obtain the transaction receipt, which contains the contract Address to initialize the contract object

The contract name is the file name with the .wasm suffix after the contract is compiled if you have a wasm contract file with name js_contracttest.cpp in *./contracts* directory, the compiled file js_contracttest.wasm and js_contracttest.abi.json file will be located in *./build/contracts* directory and the contract name will be js_contraccttest

1.11.3 Using web3

A `web3` instance is available in each test file, configured to the correct provider. So calling `web3.platon.getBalance` just works!

1.11.4 Examples

Here's an example wasm test:

```
const waitTime = 10000;
let contract = undefined;

describe("wasm unit test (you must update config before run this test)", function () {
    before("deploy contract", async function () {
            this.timeout(waitTime);
        const receipt = await web3.platon.sendTransaction(js_contracttest.
↪deployParams);
            contract = await new web3.platon.Contract(js_contracttest.abi,receipt.
↪contractAddress,{vmType:1});
    });

    it("call get method", async function () {
        this.timeout(waitTime);
        var result = await contract.methods.getUint64().call();
        assert.equal(result, 0, "getUint64 method should return 0");
    });
});
```

If you have already deployed a smart contract and do not want to redeploy, you can specify contract address in test file.

```
const waitTime = 10000;
let contract = undefined;
let contractAddress = "0x3F212ec13eAD7D409eba24a84c286dD1A527EeFD";

describe("wasm unit test (you must update config before run this test)", function () {
    before("deploy contract", async function () {
            contract = await new web3.platon.Contract(js_contracttest.abi,␣
↪contractAddress,{vmType:1});
    });

    it("call get method", async function () {
        this.timeout(waitTime);
        var result = await contract.methods.getUint64().call();
        assert.equal(result, 0, "getUint64 method should return 0");
    });
});
```

This test will produce the following output:

```
  Contract: js_contracttest
    wasm unit test (you must update config before run this test) call get method: 9ms

  1 passing (4s)
```

1.11.5 Specifying contract

You can specify a specific wasm contract to run the test

```
$ platon-truffle test --wasm --contract-name ${ContractName} --param $
↪{InitParamsString}
```

1.12 Writing test in solidity

Solidity test contracts live alongside Javascript tests as `.sol` files. When `platon-truffle test` is run, they will be included as a separate test suite per test contract. These contracts maintain all the benefits of the Javascript tests, direct access to your deployed contracts and the ability to import any contract dependency. In addition to these features, platon truffle's Solidity testing framework was built with the following issues in mind:

- Solidity tests shouldn't extend from any contract (like a `Test` contract). This makes your tests as minimal as possible and gives you complete control over the contracts you write.
- Solidity tests shouldn't be beholden to any assertion library. platon truffle provides a default assertion library for you, but you can change this library at any time to fit your needs.
- You should be able to run your Solidity tests against any PlatON client.

1.12.1 Example

Let's take a look at an example Solidity test before diving too deeply. Here's the example Solidity test provided for you by `platon-truffle unbox metacoin`:

```
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/MetaCoin.sol";

contract TestMetacoin {
  function testInitialBalanceUsingDeployedContract() {
    MetaCoin meta = MetaCoin(DeployedAddresses.MetaCoin());

    uint expected = 10000;

    Assert.equal(meta.getBalance(tx.origin), expected, "Owner should have 10000␣
↪MetaCoin initially");
  }

  function testInitialBalanceWithNewMetaCoin() {
    MetaCoin meta = new MetaCoin();

    uint expected = 10000;

    Assert.equal(meta.getBalance(tx.origin), expected, "Owner should have 10000␣
↪MetaCoin initially");
```

(continues on next page)

(continued from previous page)

```
  }
}
```

This produces the following output:

```
$ platon-truffle test
Compiling ConvertLib.sol...
Compiling MetaCoin.sol...
Compiling truffle/Assert.sol
Compiling truffle/DeployedAddresses.sol
Compiling ../test/TestMetacoin.sol...

  TestMetacoin
    ✓ testInitialBalanceUsingDeployedContract (61ms)
    ✓ testInitialBalanceWithNewMetaCoin (69ms)

  2 passing (3s)
```

1.12.2 Test structure

To better understand whats happening, let's discuss things in more detail.

Assertions

Your assertion functions like `Assert.equal()` are provided to you by the `truffle/Assert.sol` library. This is the default assertion library, however you can include your own assertion library so long as the library loosely integrates with platon truffle's test runner by triggering the correct assertion events. You can find all available assertion functions in `Assert.sol`.

Deployed addresses

The addresses of your deployed contracts (i.e., contracts that were deployed as part of your migrations) are available through the `truffle/DeployedAddresses.sol` library. This is provided by platon truffle and is recompiled and relinked before each suite is run to provide your tests with platon truffle's a clean room environment. This library provides functions for all of your deployed contracts, in the form of:

```
DeployedAddresses.<contract name>();
```

This will return an address that you can then use to access that contract. See the example test above for usage.

In order to use the deployed contract, you'll have to import the contract code into your test suite. Notice `import "../contracts/MetaCoin.sol";` in the example. This import is relative to the test contract, which exists in the `./test` directory, and it goes outside of the test directory in order to find the MetaCoin contract. It then uses that contract to cast the address to the `MetaCoin` type.

Test contract names

All test contracts must start with `Test`, using an uppercase `T`. This distinguishes this contract apart from test helpers and project contracts (i.e., the contracts under test), letting the test runner know which contracts represent test suites.

Test function names

Like test contract names, all test functions must start with `test`, lowercase. Each test function is executed as a single transaction, in order of appearance in the test file (like your Javascript tests). Assertion functions provided by `truffle/Assert.sol` trigger events that the test runner evaluates to determine the result of the test. Assertion functions return a boolean representing the outcome of the assertion which you can use to return from the test early to prevent execution errors.

before / after hooks

You are provided many test hooks, shown in the example below. These hooks are `beforeAll`, `beforeEach`, `afterAll` and `afterEach`, which are the same hooks provided by Mocha in your Javascript tests. You can use these hooks to perform setup and teardown actions before and after each test, or before and after each suite is run. Like test functions, each hook is executed as a single transaction. Note that some complex tests will need to perform a significant amount of setup that might overflow the gas limit of a single transaction; you can get around this limitation by creating many hooks with different suffixes, like in the example below:

```
import "truffle/Assert.sol";

contract TestHooks {
  uint someValue;

  function beforeEach() {
    someValue = 5;
  }

  function beforeEachAgain() {
    someValue += 1;
  }

  function testSomeValueIsSix() {
    uint expected = 6;

    Assert.equal(someValue, expected, "someValue should have been 6");
  }
}
```

This test contract also shows that your test functions and hook functions all share the same contract state. You can setup contract data before the test, use that data during the test, and reset it afterward in preparation for the next one. Note that just like your Javascript tests, your next test function will continue from the state of the previous test function that ran.

1.12.3 Advanced features

Solidity tests come with a few advanced features to let you test specific use cases within Solidity.

Testing for exceptions

You can easily test if your contract should or shouldn't raise an exception (i.e., for `require()`/`assert()`/`revert()` statements; `throw` on previous versions of Solidity).

This topic was first written about by guest writer Simon de la Rouviere in his tutorial Testing for Throws in platon truffle Solidity Tests. N.B. that the tutorial makes heavy use of exceptions via the deprecated keyword `throw`, replaced by `revert()`, `require()`, and `assert()` starting in Solidity v0.4.13.

Also, since Solidity v0.4.17, a function type member was added to enable you to access a function selector (e.g.: `this.f.selector`), and so, testing for throws with external calls has been made much easier:

```
pragma solidity ^0.5.0;

import "truffle/Assert.sol";

contract TestBytesLib2 {
    function testThrowFunctions() public {
        bool r;

        // We're basically calling our contract externally with a raw call,␣
↪forwarding all available gas, with
        // msg.data equal to the throwing function selector that we want to be sure␣
↪throws and using only the boolean
        // value associated with the message call's success
        (r, ) = address(this).call(abi.encodePacked(this.IThrow1.selector));
        Assert.isFalse(r, "If this is true, something is broken!");

        (r, ) = address(this).call(abi.encodePacked(this.IThrow2.selector));
        Assert.isFalse(r, "What?! 1 is equal to 10?");
    }

    function IThrow1() public pure {
        revert("I will throw");
    }

    function IThrow2() public pure {
        require(1 == 10, "I will throw, too!");
    }
}
```

Testing lat transactions

You can also test how your contracts react to receiving Lat, and script that interaction within Solidity. To do so, your Solidity test should have a public function that returns a `uint`, called `initialBalance`. This can be written directly as a function or a public variable, as shown below. When your test contract is deployed to the network, platon truffle will send that amount of Lat from your test account to your test contract. Your test contract can then use that Lat to script Lat interactions within your contract under test. Note that `initialBalance` is optional and not required.

```
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/MyContract.sol";

contract TestContract {
  // Truffle will send the TestContract one Lat after deploying the contract.
  uint public initialBalance = 1 lat;

  function testInitialBalanceUsingDeployedContract() {
    MyContract myContract = MyContract(DeployedAddresses.MyContract());

    // perform an action which sends value to myContract, then assert.
    myContract.send(...);
  }

  function () {
```

(continues on next page)

(continued from previous page)

```
    // This will NOT be executed when Lat is sent. \o/
  }
}
```

Note that platon truffle sends Lat to your test contract in a way that does **not** execute a fallback function, so you can still use the fallback function within your Solidity tests for advanced test cases.

1.13 Networks and APP deployment

Even the smallest project will interact with at the very least two blockchain nodes: One on the developer's machine, and the other representing the network where the developer will eventually deploy their application (such as the main public PlatON 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.

1.13.1 Configuration

See the Configuration section for more information.

1.13.2 Specifying a network

Most platon 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:

```
$ platon-truffle migrate --network live
```

In this example, platon truffle will run your migrations on the "live" network, which – if configured like the example – is associated with the public PlatON blockchain.

1.13.3 Specifying a wasm contract

If you want to deploy a specific wasm contract(contract file like contracts/test.cpp), you can use the following command

```
$ platon-truffle migrate --wasm --contract-name test
```

1.13.4 Build artifacts

As mentioned in the *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, platon 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, they'll automatically detect which network the PlatON client is connected to and use the correct contract artifacts accordingly.

1.13.5 Application deployment

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

1.14 Build Processes

> **Warning:** The *build* command and this approach is being deprecated. Please use third-party build tools like webpack or grunt, or see our *platon-truffle Boxes* for an example.

In order to provide tight integration with platon truffle for those that desire it, platon truffle allows you to specify a custom build pipeline meant to bootstrap and configure your application. Truffle provides three methods of integration, described below.

1.14.1 Running an external command

If you'd like platon truffle to run an external command whenever it triggers a build, simply include that option as a string within your project configuration, like so:

```
module.exports = {
  // This will run the `webpack` command on each build.
  //
  // The following environment variables will be set when running the command:
  // WORKING_DIRECTORY: root location of the project
  // BUILD_DESTINATION_DIRECTORY: expected destination of built assets (important for␣
↪`truffle serve`)
  // BUILD_CONTRACTS_DIRECTORY: root location of your build contract files (.sol.js)
  //
  build: "webpack"
}
```

Note that you're given ample environment variables with which to integrate with platon truffle, detailed above.

1.14.2 Providing a custom function

You can also provide a custom build function like the one below. Note you're given a plethora of information about your project which you can use to integrate tightly with Truffle.

```
module.exports = {
  build: function(options, callback) {
     // Do something when a build is required. `options` contains these values:
     //
     // working_directory: root location of the project
     // contracts_directory: root directory of .sol files
     // destination_directory: directory where truffle expects the built assets␣
↪(important for `truffle serve`)
  }
}
```

1.14.3 Creating a custom module

You could also create a module or object that implements the builder interface (i.e., is an object which contains a `build` function like the one above). This is great for those who want to maintain tighter integration with Truffle and publish a package to make everyone else's lives easier.

Here's an example using platon truffle's default builder:

```
var DefaultBuilder = require("truffle-default-builder");
module.exports = {
  build: new DefaultBuilder(...) // specify the default builder configuration here.
}
```

1.14.4 Bootstrapping your application

Whether you're building an application to run in the browser, or a command line tool, a Javascript library or a native mobile application, bootstrapping your contracts is the same, and using your deployed contract artifacts follows the same general process no matter the app you're building.

When configuring your build tool or application, you'll need to perform the following steps:

1. Get all your contract artifacts into your build pipeline / application. This includes all of the `.json` files within the `./build/contracts` directory.
2. Turn those `.json` contract artifacts into contract abstractions that are easy to use.
3. Provision those contract abstractions with a Web3 provider.
4. Use your contracts!

In Node, this is very easy to do. Let's take a look at an example that shows off the "purest" way of performing the above steps, since it exists outside of any build process or tool.

```
// Step 1: Get a contract into my application
var json = require("./build/contracts/MyContract.json");

// Step 2: Turn that contract into an abstraction I can use
var contract = require("truffle-contract");
var MyContract = contract(json);

// Step 3: Provision the contract with a web3 provider
MyContract.setProvider(new Web3.providers.HttpProvider("http://127.0.0.1:8545"));

// Step 4: Use the contract!
MyContract.deployed().then(function(deployed) {
  return deployed.someFunction();
});
```

All build processes and contract bootstrapping will follow this pattern. The key when setting up your own custom build process is to ensure you're consuming all of your contract artifacts and provisioning your abstractions correctly.

1.15 Configuration

1.15.1 Location

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

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*"
    }
  }
};
```

The default configuration ships with configuration for a single development network, running on `127.0.0.1:8545`. There are many other configuration options, detailed below.

1.15.2 General options

build

Build configuration of your application, if your application requires tight integration with platon truffle. Most users likely will not need to configure this option. See the *Build Processes* section for more details.

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 PlatON client is connected to a specific network, they'll use the contract artifacts associated that network to simplify app deployment. Networks are identified through PlatON'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, platon truffle will not be able to deploy your contracts. The default network configuration provided by `platon-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 platon 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:

```
$ platon-truffle migrate --network live
```

Example:

```
networks: {
  development: {
    host: "127.0.0.1",
```

(continues on next page)

(continued from previous page)

```
    port: 8545,
    network_id: "*", // match any network
    websockets: true
  },
  live: {
    host: "178.25.19.88", // Random IP for example purposes (do not use)
    port: 80,
    network_id: 1,        // PlatON 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 PlatON␣
↪network.
    //          - function that returns a web3 provider instance (see below.)
    //          - if specified, host and port are ignored.
    // skipDryRun: - true if you don't want to test run the migration locally before␣
↪the actual migration (default is false)
    // timeoutBlocks: - if a transaction is not mined, keep waiting for this number␣
↪of blocks (default is 50)
      }
  }
}
```

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

- `gas`: Gas limit used for deploys. Default is `6721975`.
- `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 PlatON client.
- `provider`: Default web3 provider using `host` and `port` options: `new Web3.providers.HttpProvider("http://<host>:<port>")`
- `websockets`: You will need this enabled to use the `confirmations` listener or to hear Events using `.on` or `.once`. Default is `false`.

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, 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: {
  test: {
    provider: function() {
      return new HDWalletProvider(mnemonic, "http://127.0.0.1:8545/");
    },
    network_id: '*',
```

(continues on next page)

(continued from previous page)

```
  },
}
```

If you specify host and port instead of provider, platon 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.

contracts_directory

The default directory for uncompiled contracts is ./contracts relative to the project root. If you wish to keep your contracts in a different directory you may specify a contracts_directory property.

Example:

To have platon truffle find contracts in ./allMyStuff/someStuff/theContractFolder (recursively) at compile time:

```
module.exports = {
  contracts_directory: "./allMyStuff/someStuff/theContractFolder",
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*",
    }
  }
};
```

Note: In addition to specifying a relative path, you can also use globs/regular expressions to selectively compile contracts.

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.

Example:

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. This is not recommended though, as an absolute path may not exist when compiled on another system. If you use absolute paths on Windows, make sure to use double backslashes for paths (example: /home/Users/Username/output).

migrations_directory

The default migrations directory is ./migrations relative to the project root. This can be changed with the migrations_directory key.

Example

```
module.exports = {
  migrations_directory: "./allMyStuff/someStuff/theMigrationsFolder",
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*",
    }
  }
};
```

mocha

Configuration options for the MochaJS testing framework. This configuration expects an object as detailed in Mocha's documentation.

Example:

```
mocha: {
  useColors: true
}
```

1.15.3 Compiler configuration

In the compilers object you can specify settings related to the compilers used by platon truffle.

solc

Solidity compiler settings. Supports optimizer settings for solc. You may specify...

- any solc-js version listed at solc-bin. Specify the one you want and platon truffle will get it for you.

- a natively compiled solc binary (you'll need to install this yourself, links to help below).
- a path to a locally available solc
- a solc-js parser for faster docker and native compilations

Truffle config example:

```
module.exports = {
  compilers: {
    solc: {
      version: <string>, // A version or constraint - Ex. "^0.5.0"
                         // Can also be set to "native" to use a native solc
      docker: <boolean>, // Use a version obtained through docker
      parser: "solcjs",  // Leverages solc-js purely for speedy parsing
      settings: {
        optimizer: {
          enabled: <boolean>,
          runs: <number>   // Optimize for how many times you intend to run the code
        },
        evmVersion: <string> // Default: "petersburg"
      }
    }
  }
}
```

external compilers

For more advanced use cases with artifact creation you can use the external compilers configuration. You can use this feature by adding a `compilers.external` object to your platon truffle config:

```
module.exports = {
  compilers: {
    external: {
      command: "./compile-contracts",
      targets: [{
         /* compilation output */
      }]
    }
  }
}
```

When you run platon truffle compile, platon truffle will run the configured command and look for contract artifacts specified by targets.

This new configuration supports a couple of main use cases:

- Your compilation command outputs platon truffle JSON artifacts directly. If your compilation command generates artifacts directly, or generates output that contains all the information for an artifact, configure a target as follows:

```
module.exports = {
  compilers: {
    external: {
      command: "./compile-contracts",
      targets: [{
        path: "./path/to/artifacts/*.json"
      }]
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

platon truffle will execute your script, then expand the glob (*) and find all .json files in the listed path and copy those over as artifacts in the build/contracts/ directory.

- Your compilation command outputs individual parts of an artifact, and you want platon truffle to generate the artifacts for you. The above use case might not be sufficient for all use cases. You can configure your target to run an arbitrary post-processing command:

```
module.exports = {
  compilers: {
    external: {
      command: "./compile-contracts",
      targets: [{
        path: "./path/to/preprocessed-artifacts/*.json",
        command: "./process-artifact"
      }]
    }
  }
}
```

This will run ./process-artifact for each matched .json file, piping the contents of that file as stdin. Your ./process-artifact command is then expected to output a complete platon truffle artifact as stdout.

Want to provide the path as a filename instead? Add `stdin: false` to your target configuration.

- You can also specify the individual properties of your contracts and have platon truffle generate the artifacts itself.

```
module.exports = {
  compilers: {
    external: {
      command: "./compile-contracts",
      targets: [{
        properties: {
          contractName: "MyContract",
          /* other literal properties */
        },
        fileProperties: {
          abi: "./output/contract.abi",
          bytecode: "./output/contract.bytecode",
          /* other properties encoded in output files */
        }
      }]
    }
  }
}
```

Specify `properties` and/or `fileProperties`, and platon truffle will look for those values when building the artifacts.

To override the working directory for all specified paths and running commands, use the `workingDirectory` option. For instance, the following will run `./proj/compile-contracts` and read `./proj/output/contract.abi`:

```
module.exports = {
  compilers: {
    external: {
      command: "./compile-contracts",
      workingDirectory: "./proj",
      targets: [{
        fileProperties: {
          abi: "./output/contract.abi",
          bytecode: "./output/contract.bytecode",
        }
      }]
    }
  }
}
```

wasm

The wasm compiler is configured as follows

```
module.exports = {
  compilers: {
    wasm: {
      version: "1.0.0"
    }
  }
}
```

The wasm compiler currently has no version management, so you can specify any version number for the time being

The wasm compiler and solc compiler cannot be specified at the same time, you need to choose according to the type of contract you are compiling if you want to compile solidity contract, you can only specific solc compilers config, if wasm, you need only specific wasm compilers

1.16 Contract abstractions

platon truffle provides contract abstractions for interacting with your contracts. Skip ahead to the api section for a list of contract methods.

1.16.1 Usage

To obtain a contract abstraction you can require it with the contract name from the `artifacts` object. Outside of the console this is an object available in migration files, tests, and exec scripts. You would require your contract as follows:

```
const MyContract = artifacts.require("MyContract");
```

You can also obtain one in the developer console. Your contract types are available here and all you need to do is use the `at`, `deployed`, or `new` method.

```
truffle(development)> const myContract = await MyContract.deployed();
```

You 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.

Each instance is tied to a specific address on the PlatON network, and each instance has a 1-to-1 mapping from Javascript functions to contract functions. For instance, if your Solidity contract had a function defined `someFunction(uint value) {}` (solidity), then you could execute that function on the network like so:

```
let deployed;
MyContract.deployed()
  .then((instance) => {
    deployed = instance;
    return deployed.someFunction(5);
  }).then((result) => {
    // Do something with the result or continue with more transactions.
  });
```

You can also use async/await syntax which is often much less verbose. We will use async/await for the rest of this document but you may also use a promises for interfacing with contract methods as well.

```
const deployed = await MyContract.deployed();
const result = await deployed.someFunction(5);
// Do something with the result or continue with more transactions.
```

See the processing transaction results section to learn more about the results object obtained from making transactions.

Contract methods and events have an EventEmitter interface. So you can set up handlers like the following:

```
const example = await artifacts.require("Example").deployed();

example
  .setValue(45)
  .on('transactionHash', hash => {})
  .on('receipt', receipt => {})
  .on('error', error => {})
  .on('confirmation', (num, receipt) => {})
  .then(receipt => {});
```

```
example
  .ExampleEvent()
  .on('data', event => ... etc ... )

example
  .ExampleEvent()
  .once('data', event => ... etc ... )
```

1.17 API

There are two API's you'll need to be aware of. One is the static Contract Abstraction API and the other is the Contract Instance API. The Abstraction API is a set of functions that exist for all contract abstractions, and those function exist on the abstraction itself (i.e., `MyContract.at()`). In contrast, the Instance API is the API available to contract instances – i.e., abstractions that represent a specific contract on the network – and that API is created dynamically based on functions available in your Solidity source file.

1.17.1 Contract Abstraction API

Each contract abstraction – `MyContract` in the examples above – have the following useful functions:

`MyContract.new([arg1, arg2, ...], [tx params])`

This function take whatever contructor parameters your contract requires and deploys a new instance of the contract to the network. There's an optional last argument which you can use to pass transaction parameters including the transaction from address, gas limit and gas price. This function returns a Promise that resolves into a new instance of the contract abstraction at the newly deployed address.

`MyContract.at(address)`

This function creates a new instance of the contract abstraction representing the contract at the passed in address. Returns a "thenable" object (not yet an actual Promise for backward compatibility). Resolves to a contract abstraction instance after ensuring code exists at the specified address.

`MyContract.deployed()`

Creates an instance of the contract abstraction representing the contract at its deployed address. The deployed address is a special value given to truffle-contract that, when set, saves the address internally so that the deployed address can be inferred from the given PlatON network being used. This allows you to write code referring to a specific deployed contract without having to manage those addresses yourself. Like `at()`, `deployed()` is thenable, and will resolve to a contract abstraction instance representing the deployed contract after ensuring that code exists at that location and that that address exists on the network being used.

`MyContract.link(instance)`

Link a library represented by a contract abstraction instance to MyContract. The library must first be deployed and have its deployed address set. The name and deployed address will be inferred from the contract abstraction instance. When this form of `MyContract.link()` is used, MyContract will consume all of the linked library's events and will be able to report that those events occurred during the result of a transaction.

Libraries can be linked multiple times and will overwrite their previous linkage.

Note: This method has two other forms, but this form is recommended.

`MyContract.link(name, address)`

Link a library with a specific name and address to MyContract. The library's events will not be consumed using this form.

`MyContract.link(object)`

Link multiple libraries denoted by an Object to MyContract. The keys must be strings representing the library names and the values must be strings representing the addresses. Like above, libraries' events will not be consumed using this form.

MyContract.networks()

View a list of network ids this contract abstraction has been set up to represent.

MyContract.setProvider(provider)

Sets the web3 provider this contract abstraction will use to make transactions.

MyContract.setNetwork(network_id)

Sets the network that MyContract is currently representing.

MyContract.hasNetwork(network_id)

Returns a boolean denoting whether or not this contract abstraction is set up to represent a specific network.

MyContract.defaults([new_defaults])

Get's and optionally sets transaction defaults for all instances created from this abstraction. If called without any parameters it will simply return an Object representing current defaults. If an Object is passed, this will set new defaults. Example default transaction values that can be set are:

```
MyContract.defaults({
  from: ...,
  gas: ...,
  gasPrice: ...,
  value: ...
})
```

Setting a default `from` address, for instance, is useful when you have a contract abstraction you intend to represent one user (i.e., one address).

MyContract.clone(network_id)

Clone a contract abstraction to get another object that manages the same contract artifacts, but using a different `network_id`. This is useful if you'd like to manage the same contract but on a different network. When using this function, don't forget to set the correct provider afterward.

```
const MyOtherContract = MyContract.clone(1337);
```

MyContract.numberFormat = number_type

You can set this property to choose the number format that abstraction methods return. The default behavior is to return BN.

```
// Choices are:  `["BigNumber", "BN", "String"].
const Example = artifacts.require('Example');
Example.numberFormat = 'BigNumber';
```

MyContract.timeout(block_timeout)

This method allows you to set the block timeout for transactions. Contract instances created from this abstraction will have the specified transaction block timeout. This means that if a transaction does not immediately get mined, it will retry for the specified number of blocks.

MyContract.autoGas = <boolean>

If this is set to true, instances created from this abstraction will use `web3.platon.estimateGas` and then apply a gas multiplier to determine the amount of gas to include with the transaction. The default value for this is `true`. See gasMultiplier.

MyContract.gasMultiplier(gas_multiplier)

This is the value used when `autoGas` is enabled to determine the amount of gas to include with transactions. The gas is computed by using `web3.platon.estimateGas` and multiplying it by the gas multiplier. The default value is `1.25`.

1.17.2 Contract Instance API

Each contract instance is different based on the source Solidity contract, and the API is created dynamically. For the purposes of this documentation, let's use the following Solidity source code below:

```
contract MyContract {
  uint public value;
  event ValueSet(uint val);
  function setValue(uint val) {
    value = val;
    emit ValueSet(value);
  }
  function getValue() constant returns (uint) {
    return value;
  }
}
```

From Javascript's point of view, this contract has three functions: `setValue`, `getValue` and `value`. This is because `value` is public and automatically creates a getter function for it.

Making a transaction via a contract function

When we call `setValue()`, this creates a transaction. From Javascript:

```
const result = await instance.setValue(5);
// result object contains import information about the transaction
console.log("Value was set to", result.logs[0].args.val);
```

The result object that gets returned looks like this:

```
{
  tx: "0x6cb0bbb6466b342ed7bc4a9816f1da8b92db1ccf197c3f91914fc2c721072ebd",
  receipt: {
    // The return value from web3.platon.getTransactionReceipt(hash)
```

(continues on next page)

(continued from previous page)

```
  },
  logs: [
    { logIndex: 0,
      transactionIndex: 0,
      transactionHash:
↪'0x728b4d1983cd00d93ae00b7adf76f78c1b32d922de636ead42e93f70cf58cdc9',
      blockHash: '0xdce5e6c580267c9bf1d82bf0a167fa60509ef9fc520b8619d8183a8373a42035',
      blockNumber: 19,
      address: '0x035b8A9e427d93D178E2D22d600B779717696831',
      type: 'mined',
      id: 'log_70be22b0',
      event: 'Transfer',
      args:
        Result {
          '0': '0x7FEb9FAA5aED0FD547Ccc70f00C19dDe95ea54d4',
          '1': '0x7FEb9FAA5aED0FD547Ccc70f00C19dDe95ea54d4',
          '2': <BN: 1>,
          __length__: 3,
          _from: '0x7FEb9FAA5aED0FD547Ccc70f00C19dDe95ea54d4',
          _to: '0x7FEb9FAA5aED0FD547Ccc70f00C19dDe95ea54d4',
          _value: <BN: 1>
        }
    }
  ],
}
```

Note that if the function being executed in the transaction has a return value, you will not get that return value inside this result. You must instead use an event (like `ValueSet`) and look up the result in the `logs` array.

Explicitly making a call instead of a transaction

We can call `setValue()` without creating a transaction by explicitly using `.call`:

```
const value = await instance.setValue.call(5);
```

This isn't very useful in this case, since `setValue()` sets things, and the value we pass won't be saved since we're not creating a transaction.

Calling getters

However, we can *get* the value using `getValue()`, using `.call()`. Calls are always free and don't cost any Lat, so they're good for calling functions that read data off the blockchain:

```
const value = await instance.getValue.call();
// value reprsents the `value` storage object in the solidity contract
// since the contract returns that value.
```

Even more helpful, however is we *don't even need* to use `.call` when a function is marked as `constant`, because `truffle-contract` will automatically know that that function can only be interacted with via a call:

```
const value = await instance.getValue();
// val reprsents the `value` storage object in the solidity contract
// since the contract returns that value.
```

Processing transaction results

When you make a transaction, you're given a `result` object that gives you a wealth of information about the transaction. You're given the transaction has (`result.tx`), the decoded events (also known as logs; `result.logs`), and a transaction receipt (`result.receipt`). In the below example, you'll recieve the `ValueSet()` event because you triggered the event using the `setValue()` function:

```
const result = await instance.setValue(5);
// result.tx => transaction hash, string
// result.logs => array of trigger events (1 item in this case)
// result.receipt => receipt object
```

Sending Lat / Triggering the fallback function

You can trigger the fallback function by sending a transaction to this function:

```
const result = instance.sendTransaction({...});
// Same result object as above.
```

This is promisified like all available contract instance functions, and has the same API as `web3.platon.sendTransaction` without the callback. The `to` value will be automatically filled in for you.

If you only want to send Lat to the contract a shorthand is available:

```
const result = await instance.send(web3.toVon(1, "lat"));
// Same result object as above.
```

Estimating gas usage

Run this function to estimate the gas usage:

```
const result = instance.setValue.estimateGas(5);
// result => estimated gas for this transaction
```

1.18 Truffle commands

This section will describe every command available in the Truffle application.

1.18.1 Usage

All commands are in the following form:

```
platon-truffle <command> [options]
```

Passing no arguments is equivalent to `platon-truffle help`, which will display a list of all commands and then exit.

1.18.2 Available commands

compile

Compile contract source files.

```
platon-truffle compile [--list <filter>] [--all] [--network <name>] [--quiet]
```

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

Options:

- `--list <filter>`: List all recent stable releases from solc-bin. If filter is specified then it will display only that type of release or docker tags. The filter parameter must be one of the following: prereleases, releases, latestRelease or docker.
- `--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.
- `--quiet`: Suppress all compilation output.

config

Display whether analytics are enabled or disabled and prompt whether to toggle the setting.

```
platon-truffle config [--enable-analytics|--disable-analytics]
```

Options:

- `--enable-analytics|--disable-analytics`: Enable or disable analytics.

console

Run a console with contract abstractions and commands available.

```
platon-truffle console [--network <name>] [--verbose-rpc]
```

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

See the `Using the console` section for more details.

Options:

- `--network <name>`: Specify the network to use. Network name must exist in the configuration.

create

Helper to create new contracts, migrations and tests.

```
platon-truffle create <artifact_type> <ArtifactName>
```

Options:

- `<artifact_type>`: Create a new artifact where artifact_type is one of the following: contract, migration or test. The new artifact is created along with one of the following files: `contracts/ArtifactName.sol`, `migrations/####_artifact_name.js` or `tests/artifact_name.js`. (required)
- `<ArtifactName>`: Name of new artifact. (required)

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.

deploy

`migrate` alias. reference *migrate*

exec

Execute a JS module within the platon truffle environment.

```
platon-truffle exec <script.js> [--network <name>] [--compile]
```

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

See the *Writing external scripts* section for more details.

Options

- `<script.js>`: JavaScript file to be executed. Can include path information if the script does not exist in the current directory. (required)
- `--network <name>`: Specify the network to use, using artifacts specific to that network. Network name must exist in the configuration.
- `--compile`: Compile contracts before executing the script.

help

Display a list of all commands or information about a specific command.

```
platon-truffle help [<command>]
```

Options

- `<command>`: Display usage information about the specified command.

init

Initialize new and empty PlatON project

```
platon-truffle init [--force]
```

Creates a new and empty platon truffle project within the current working directory.

Note: : Older versions of platon truffle used *platon-truffle init bare* to create an empty project. This usage has been deprecated. Those looking for the MetaCoin example that used to be available through *platon-truffle init* should use *platon-truffle unbox MetaCoin* instead.

Options

- `--force`: Initialize project regardless of the current working directory's state. Be careful, this could overwrite existing files that have name conflicts.

migrate

Run migrations to deploy contracts.

```
platon-truffle migrate [--reset] [--wasm] [--f <number>] [--to <number>] [--network
↪<name>] [--compile-all] [--contract-name] [--verbose-rpc] [--dry-run] [--
↪interactive]
```

Unless specified, this will run from the last completed migration. See the *Migrations* section for more details.

Options

- `--reset`: Run all migrations from the beginning, instead of running from the last completed migration.
- `--wasm`: migration for all wasm contract.
- `--f <number>`: Run contracts from a specific migration. The number refers to the prefix of the migration file.
- `--to <number>`: Run contracts to 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.
- `--contract-name`: migration for specific name wasm contract.
- `--verbose-rpc`: Log communication between platon truffle and the PlatON client.
- `--dry-run`: Fork the network specified and only perform a test migration.
- `--interactive`: Prompt to confirm that the user wants to proceed after the dry run.

networks

Show addresses for deployed contracts on each network.

```
platon-truffle networks [--clean]
```

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

Options

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

opcode

Print the compiled opcodes for a given contract.

```
platon-truffle opcode <contract_name>
```

Options

- `<contract_name>`: Name of the contract to print opcodes for. Must be a contract name, not a file name. (required)

test

Run JavaScript and Solidity tests.

```
platon-truffle test [<test_file>] [--compile-all] [--network <name>] [--verbose-rpc]
↪[--show-events]
```

Runs some or all tests within the `test/` directory as specified. See the section on Testing your contracts for more information.

Options

- `<test_file>`: Name of the test file to be run. Can include path information if the file does not exist in the current directory.
- `--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 platon truffle and the PlatON client.
- `--show-events`: Log all contract events.

version

Show version number and exit.

```
platon-truffle version
```