Hello Contract (Truffle + Remote Node)

How to deploy and interact your own smart contracts using a remote node.

tip As Celo is fully EVM compatible, we inherit the rich developer ecosystem and tooling of the Ethereum community. You will be deploying a typical hello world smart contract onto the Alfajores testnet with the common Ethereum tool, Truffle. note This page is similar to the Hello Contracts page, but this one will connect to a remote node (Forno) and do key management in the Truffle project.

Setup

This guide assumes that you have a basic Node/NPM setup.

info Learn more about the Truffle development framework here. As you may know, Truffle is built for Ethereum developers. Because Celo has a similar network architecture and runs the Ethereum Virtual Machine, Celo developers are able to leverage many Ethereum developer tools. But it is important to keep in mind the differences. If you haven't already, please review the Celo overview.

→ Celo Overview /general Clone this Truffle project from GitHub to get started.

git clone https://github.com/critesjosh/hello_contract-truffle.git This is a basic truffle project, with some additional files to help us with account management and deploying to a remote Celo test net node. Runnpm install to install of the project dependencies.

Hello World

Add a contract with the command

truffle create contract HelloWorld We will not go into the details of how to write Solidity in this exercise, but you can learn more at theSolidity documentation page .

The contract will just store a name for now:

contracts/HelloWorld.sol pragma

solidity

= 0.5.0

< 0.8.0 ;

contract

HelloWorld

{ string name =

'Celo' ;

function

getName ( )

public

view

returns

(string

memory)

{ return name ; }

function

setName (string

calldata newName)

external

{ name = newName ; } }

Prepare Deployment

Compile the contract

Before you deploy the contract, you need to compile the Solidity code into Ethereum bytecode. The following truffle command will look in the./contracts directory and compile any new or updated Solidity (.sol) contracts.

truffle compile After compiling the contract, you need to create a migration to deploy the contract. For that, create a file in the./migrations/ folder named2_deploy_helloworld.js :

info Learn more about Truffle migrations here. migrations/2_deploy_helloworld.js var

HelloWorld

= artifacts . require ("HelloWorld") ;

module . exports

=

function

(deployer)

{ deployer . deploy (HelloWorld) ; } ; info You can learn more about Truffle configuration options here.

Deploy to Alfajores (Remotely)

When you deploy contracts to the Celo network with a remote node, you have to sign the contract deployment transaction locally before sending it to the remote node to be broadcast to the network. This presents some unique challenges when using Ethereum development tools (like Truffle) because Celo transaction objects are slightly different than Ethereum transaction objects.

When you are ready to deploy your contract to Alfajores, you'll need a Celo client connected to the testnet. In this exercise you will connect to a remote node to read and write to the public testnet (Alfajores), but you could also run a testnet node locally to perform the same actions.

Here are the steps to go through to deploy the contract to the Alfajores testnet.

1. Connect to Forno (a remote Celo node service provider)
2. Get personal account information (generate a private key if required, stored in./.env
3.)
4. Get your personal account address and fund it via thefaucet
5. Get the compiled contract bytecode
6. Create and sign the contract deployment transaction
7. Send transaction to the network

Make sure the dependencies are installed with:

yarn install Run the createAccount.js script with:

node createAccount.js createAccount.js const

Web3

=

require ("web3") ; const web3 =

new

Web3 ("http://localhost:8545") ;

console . log (web3 . eth . accounts . create ( )) ; The provided code will print a private key / account pair in the terminal. Copy and paste the printedpriavteKey into aPRIVATE_KEY variable in a file called.env , similar to what is shown in the.envexample file. Theaddress that is printed with the private key is the account that we will fund with the faucet.

If you go to theAlfajores Faucet Page , you can faucet your account some CELO and see your balance increase.

Deploy the contract

Truffle Deployment

Before you can use truffle for the migration, you need to set up the proper configuration in./truffle-config.js . At the top of./truffle-config.js , set up thekit by connecting to the test network and adding the account you just funded.

truffle.config.js const

ContractKit

=

require ("@celo/contractkit") ; const

Web3

=

require ("web3") ; require ("dotenv") . config ( ) ;

const web3 =

new

Web3 ("https://alfajores-forno.celo-testnet.org") ; const kit =

ContractKit . newKitFromWeb3 (web3) ;

kit . connection . addAccount (process . env . PRIVATE_KEY) ; Then, in thenetworks object, you can add the initializedkit provider to analfajores property.

truffle.config.js networks :

{ test :

{ host :

"127.0.0.1" , port :

7545 , network_id :

"*" } , alfajores :

{ provider : kit . connection . web3 . currentProvider ,

// CeloProvider network_id :

44787 ,

// Alfajores network id gas :

4000000 ,

// You need to include the gas limit } } info Truffle doesn't estimate the gas properly, so you need to specify a gas limit intruffle.config.js . Now, deploy the contracts to Alfajores with this command:

truffle migrate

network alfajores

Custom Node.js Deployment

In this section, you will deploy a contract using a simple Node.js script to show how you can do it without using Truffle.

You need to compile theHelloWorld.sol contract using (if it isn't already):

truffle compile This command will generate aHelloWorld.json file in the./build/contracts/ directory.HelloWorld.json contains a lot of data about the contract, compiler and low level details. Import this file into the deployment scriptcelo_deploy.js with:

const

HelloWorld

=

require ("./build/contracts/HelloWorld.json") ; You are finally ready to deploy the contract. Use thekit to create a custom transaction that includes the contract bytecode.

celo_deploy.js let tx =

await kit . connection . sendTransaction ({ from : address , data :

HelloWorld . bytecode ,

// from ./build/contracts/HelloWorld.json }) ; info To deploy a contract on Celo, use thekit.connection.sendTransaction() function with noto: field and the contract bytecode in thedata field. The account that you are sending the transaction from must have enough CELO to pay the transaction fee, unless you specify another currency as thefeeCurrency , then you need enough of that currency to pay the transaction fee. The entire deployment script is about 20 lines of code.

celo_deploy.js const

Web3

=

require ("web3") ; const

ContractKit

=

require ("@celo/contractkit") ; const web3 =

new

Web3 ("https://alfajores-forno.celo-testnet.org") ; const privateKeyToAddress = require ("@celo/utils/lib/address") . privateKeyToAddress ; const kit =

ContractKit . newKitFromWeb3 (web3) ; require ("dotenv") . config ( ) ; const

HelloWorld

=

require ("./build/contracts/HelloWorld.json") ;

async

function

awaitWrapper ( )

{ kit . connection . addAccount (process . env . PRIVATE_KEY) ;

// this account must have a CELO balance to pay transaction fees

// This account must have a CELO balance to pay tx fees // get some testnet funds at https://faucet.celo.org const address =

privateKeyToAddress (process . env . PRIVATE_KEY) ; console . log (address) ;

let tx =

await kit . connection . sendTransaction ({ from : address , data :

HelloWorld . bytecode , }) ;

const receipt =

await tx . waitReceipt ( ) ; console . log (receipt) ; }

awaitWrapper ( ) ; Congratulations! You have deployed your first contract onto Celo! You can verify your contract deployment onBlockscout . You can get the transaction hash from the receipt and look it up on the block explorer.

Interacting with Custom Contracts

Now HelloWorld.sol is deployed onto the Alfajores testnet. How can you interact with the deployed contract using ContractKit?helloWorld.js includes some example code that shows how you can do this.

There are 3 functions defined inhelloWorld.js that accomplish this.

The first function,initContract() , reads the deployed contract information from the Truffle artifact atHelloWorld.json . With this information, you can create a new web3.js Contract instance:

helloWorld.js async

function

initContract ( )

{ // Check the Celo network ID const networkId =

await web3 . eth . net . getId ( ) ;

// Get the contract associated with the current network const deployedNetwork =

HelloWorld . networks [networkId] ;

// Create a new contract instance with the HelloWorld contract info let instance =

new

kit . web3 . eth . Contract (HelloWorld . abi , deployedNetwork && deployedNetwork . address) ;

getName (instance) ; setName (instance ,

"hello world!") ; } After creating the contract instance, theinitContract() function callsgetName() andsetName() .

ThegetName() function will call, return and print thegetName() function of the provided instance of the HelloWorld contract.

helloWorld.js async

function

getName (instance)

{ let name =

await instance . methods . getName ( ) . call ( ) ; console . log (name) ; } ThesetName() function is a bit more involved. First, it gets the account key from the provided./secret file, just like incelo_deploy.js . Then it creates atxObject that encodes a smart contract transaction call tosetName() with the providednewName to the provided instance of the HelloWorld contract. Then the function sends the encoded transaction object to the network, waits for a reciept and prints it to the console.

helloWorld.js async

function

setName (instance , newName)

{ // Add your account to ContractKit to sign transactions // This account must have a CELO balance to pay tx fees, get some https://faucet.celo.org kit . connection . addAccount (process . env . PRIVATE_KEY) ; const address =

privateKeyToAddress (process . env . PRIVATE_KEY) ;

// Encode the transaction to HelloWorld.sol according to the ABI let txObject =

await instance . methods . setName (newName) ;

// Send the transaction let tx =

await kit . sendTransactionObject (txObject ,

{

from : address }) ;

let receipt =

await tx . waitReceipt ( ) ; console . log (receipt) ; } The above method shows a more detail about how to create custom deployment transactions and scripts than the previous method.

As you can see, all the goodies from Ethereum apply to Celo, so virtually all tutorials and other content should be easily translatable to Celo.

Check outhttps://celo.org/developers for more resources! Edit this page Previous Celo Core Contracts (Wrapper/Registry) Next Querying on-chain identifiers with ODIS