

INSTRUCTIONAL MATERIAL

DEPARTMENT OF INFORMATION TECHNOLOGY

CREATING SMART CONTRACTS USING ETHEREUM

Ms. G. BHUVANESHWARI

Assistant Professor

Department of Information Technology



TAGORE ENGINEERING COLLEGE

Rathinamangalam, Chennai-600 127

CREATING SMART CONTRACTS USING ETHEREUM

Deploy Your First Smart Contract (web3.university)

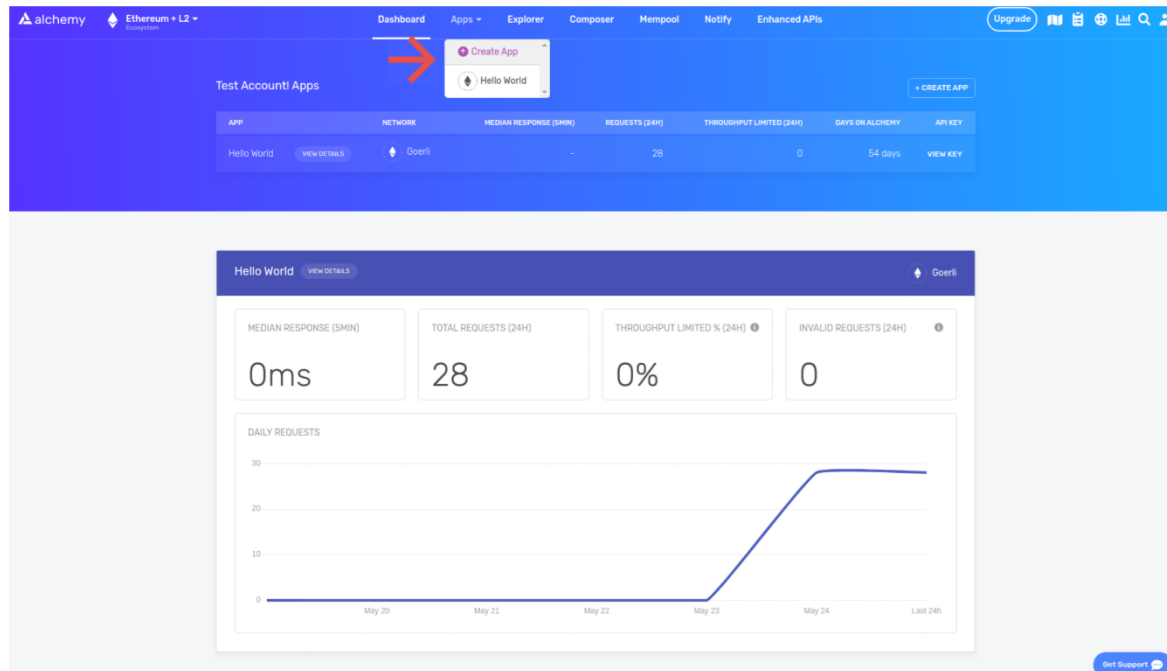
Step 1: Connect to the Ethereum network

There are many ways to make requests to the Ethereum chain. For simplicity, we'll use a free account on Alchemy, a blockchain developer platform and API that allows us to communicate with the Ethereum chain without having to run our own nodes. The platform also has developer tools for monitoring and analytics that we'll take advantage of in this tutorial to understand what's going on under the hood in our smart contract deployment.

Step 2: Create your app (and API key)

Once you've created an Alchemy account, you can generate an API key by creating an app. This will allow us to make requests to the Goerli test network. If you're not familiar with testnets, check out [this guide](#).

Navigate to the "Create App" page in your Alchemy Dashboard by hovering over "Apps" in the nav bar and clicking "Create App"



Name your app “Hello World”, offer a short description, select “Staging” for the Environment (used for your app bookkeeping), and choose “Goerli” for your network.

alchemy Ethereum + L2

Dashboard Apps Explorer Composer Mempool Notify Enhanced APIs Upgrade

Create App

NAME: Hello-World DESCRIPTION: E.g., Our user facing website

CHAIN: Ethereum NETWORK: Goerli

Create app App created!

Apps

APPS CREATED 1/5

NAME	NETWORK	DESCRIPTION	CUPS LIMIT	CONCURRENT REQUEST LIMIT	DAYS ON ALCHEMY	CREATED BY	ACTIONS
Hello World	Goerli		330	Unlimited	54 days	I	VIEW KEY - EDIT APP

The thing I hate most about Alchemy is:

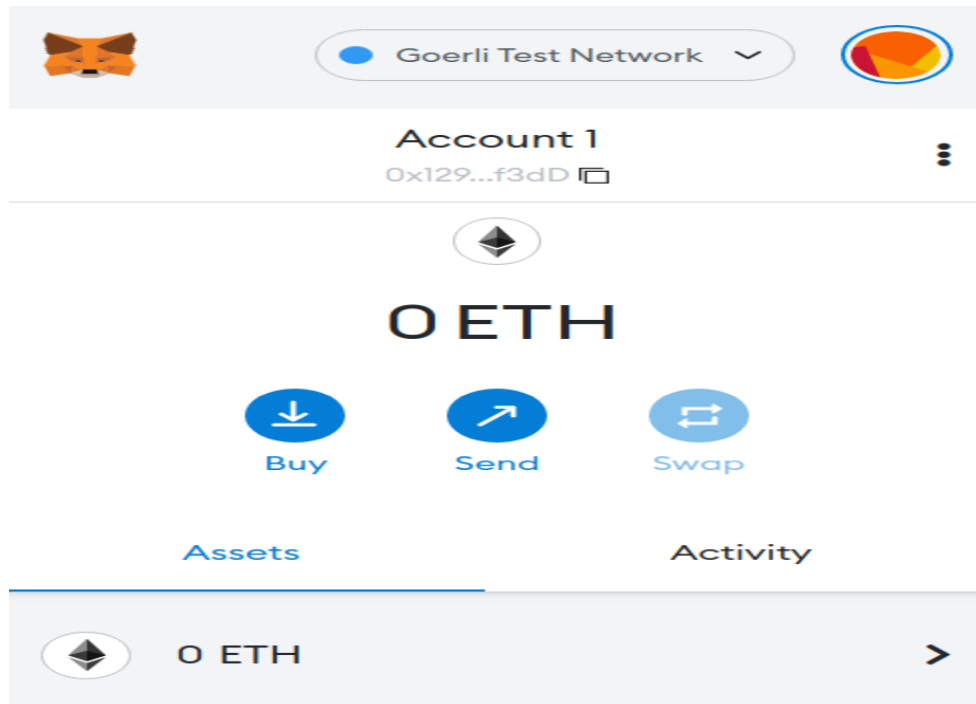
Type your mean answer

SUBMIT

© 2022 Alchemy Insights, Inc. Get Support

Step 3: Create an Ethereum account (address)

We need an Ethereum account to send and receive transactions. For this tutorial, we'll use Metamask, a virtual wallet in the browser used to manage your Ethereum account address. If you want to understand more about how transactions on Ethereum work, check out this page from the Ethereum foundation. You can download and create a Metamask account for free here. When you are creating an account, or if you already have an account, make sure to switch over to the “Goerli Test Network” in the upper right (so that we're not dealing with real money).



Step 4: Add ether from a Faucet

In order to deploy our smart contract to the test network, we'll need some fake Eth. To get Eth you can go to the Goerli faucet and enter your Goerli account address, then click "Send Me Eth." It may take some time to receive your fake Eth due to network traffic. (At the time of writing this, it took around 30 minutes.) You should see Eth in your MetaMask account soon after!

Step 5: Check your Balance

To double check our balance is there, let's make an `eth_getBalance` request using Alchemy's composer tool. This will return the amount of Eth in our wallet. Check out this video for instructions on how to use the composer tool!

After you input your Metamask account address and click "Send Request", you should see a response that looks like this:

```
{"jsonrpc": "2.0", "id": 0, "result": "0x2B5E3AF16B1880000"}
```

Step 6: Initialize our project

```
mkdir hello-world
```

```
cd hello-world
```

First, we'll need to create a folder for our project. Navigate to your command line and type:

Now that we're inside our project folder, we'll use npm init to initialize the project. If you don't already have npm installed, follow these instructions (we'll also need Node.js so download that too!).

npm init # (or npm init --yes)

It doesn't really matter how you answer the installation questions, here is how we did it for reference:

package name: (hello-world)

version: (1.0.0)

description: hello world smart contract

entry point: (index.js)

test command:

git repository:

keywords:

author:

license: (ISC)

About to write to /Users/.../.../.../hello-world/package.json:

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "hello world smart contract",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Step 7: Download Hardhat

Hardhat is a development environment to compile, deploy, test, and debug your Ethereum software. It helps developers when building smart contracts and dApps locally before deploying to the live chain.

Inside our hello-world project run:

```
npm install --save-dev hardhat
```

Step 8: Create Hardhat project

Inside our hello-world project folder, run:

```
npx hardhat
```

You should then see a welcome message and option to select what you want to do. Select “create an empty hardhat.config.js”:

```
888 888          888 888      888
888 888          888 888      888
888 888          888 888      888
888888888888 8888
b. 888d888 .d88888 88888b. 8888b. 888888
888 888  "88b 888P" d88" 888 888 "88b  "88b 888
888 888 .d888888 888 888 888 888 .d888888 888
888 888 888 888 888 Y88b 888 888 888 888 Y88b.
888 888 "Y888888 888 "Y88888 888 888 "Y888888 "Y888
```

 Welcome to Hardhat v2.0.11 

What do you want to do? ...

Create a sample project

› Create an empty hardhat.config.js

Quit

This will generate a hardhat.config.js file for us, which is where we’ll specify all of the set up for our project (on step 13).

Step 9: Add project folders

To keep our project organized we’ll create two new folders. Navigate to the root directory of your hello-world project in your command line and type

```
mkdir contracts
```

```
mkdir scripts
```

contracts/ is where we'll keep our hello world smart contract code file

scripts/ is where we'll keep scripts to deploy and interact with our contract

Step 10: Write our contract

You might be asking yourself, when the heck are we going to write code?? Well, here we are, on Step 10

Open up the hello-world project in your favorite editor (we like VSCode). Smart contracts are written in a language called Solidity which is what we will use to write our HelloWorld.sol smart contract.

1. Navigate to the "contracts" folder and create a new file called HelloWorld.sol
2. Below is a sample Hello World smart contract from the Ethereum Foundation that we will be using for this tutorial. Copy and paste in the contents below into your HelloWorld.sol file, and be sure to read the comments to under// Specifies the version of Solidity, using semantic versioning.

```
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma
```

```
pragma solidity >=0.7.3;
```

```
// Defines a contract named `HelloWorld`.
```

```
// A contract is a collection of functions and data (its state). Once deployed, a contract resides at a specific address on the Ethereum blockchain. Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html
```

```
contract HelloWorld {
```

```
    //Emitted when update function is called
```

```
    //Smart contract events are a way for your contract to communicate that something happened on the blockchain to your app front-end, which can be 'listening' for certain events and take action when they happen.
```

```
    event UpdatedMessages(string oldStr, string newStr);
```

```
    // Declares a state variable `message` of type `string`.
```

// State variables are variables whose values are permanently stored in contract storage. The keyword `public` makes variables accessible from outside a contract and creates a function that other contracts or clients can call to access the value.

```
string public message;
```

// Similar to many class-based object-oriented languages, a constructor is a special function that is only executed upon contract creation.

// Constructors are used to initialize the contract's data. Learn more: <https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constructors>

```
constructor (string memory initMessage) {
```

// Accepts a string argument `initMessage` and sets the value into the contract's `message` storage variable).

```
    message = initMessage;
```

```
}
```

// A public function that accepts a string argument and updates the `message` storage variable.

```
function update(string memory newMessage) public
```

```
{
```

```
    string memory oldMsg = message;
```

```
    message = newMessage;
```

```
    emit UpdatedMessages(oldMsg, newMessage);
```

```
}
```

```
}
```

stand what this contract does:

Step 11: Connect Metamask & Alchemy to your project

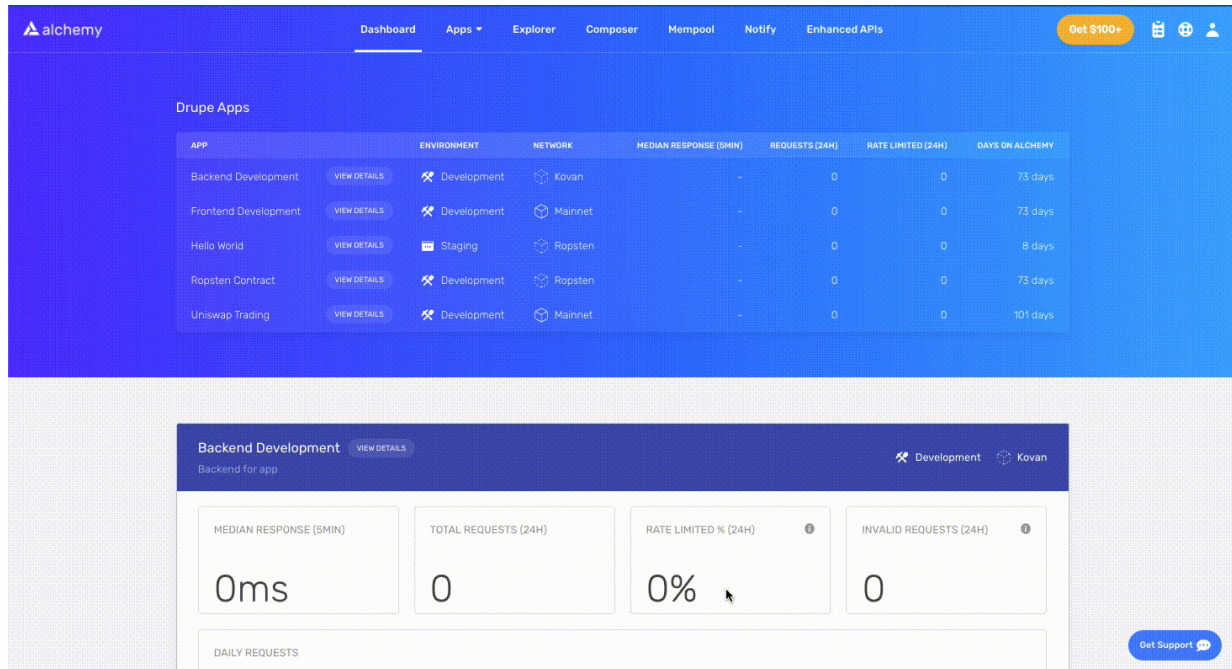
We've created a Metamask wallet, Alchemy account, and written our smart contract, now it's time to connect the three.

Every transaction sent from your virtual wallet requires a signature using your unique private key. To provide our program with this permission, we can safely store our private key (and Alchemy API key) in an environment file.

First, install the dotenv package in your project directory:

```
npm install dotenv --save
```

This is a super simple smart contract that stores a message upon creation and can be updated by calling the update function.



The screenshot shows the Alchemy dashboard. At the top, there's a navigation bar with 'Dashboard', 'Apps', 'Explorer', 'Composer', 'Mempool', 'Notify', and 'Enhanced APIs'. A 'Get \$100+' button is on the right. Below the navigation bar, there's a section titled 'Drupe Apps' which contains a table with columns: APP, ENVIRONMENT, NETWORK, MEDIAN RESPONSE (5MIN), REQUESTS (24H), RATE LIMITED (24H), and DAYS ON ALCHEMY. The table lists five apps: Backend Development, Frontend Development, Hello World, Ropsten Contract, and Uniswap Trading. Below the table, there's a detailed view for 'Backend Development' showing metrics: MEDIAN RESPONSE (5MIN) at 0ms, TOTAL REQUESTS (24H) at 0, RATE LIMITED % (24H) at 0%, and INVALID REQUESTS (24H) at 0. A 'DAILY REQUESTS' section is partially visible at the bottom.

APP	VIEW DETAILS	ENVIRONMENT	NETWORK	MEDIAN RESPONSE (5MIN)	REQUESTS (24H)	RATE LIMITED (24H)	DAYS ON ALCHEMY
Backend Development	VIEW DETAILS	Development	Kovan	-	0	0	73 days
Frontend Development	VIEW DETAILS	Development	Mainnet	-	0	0	73 days
Hello World	VIEW DETAILS	Staging	Ropsten	-	0	0	8 days
Ropsten Contract	VIEW DETAILS	Development	Ropsten	-	0	0	73 days
Uniswap Trading	VIEW DETAILS	Development	Mainnet	-	0	0	101 days

Backend Development		VIEW DETAILS	Development	Kovan
Backend for app				
MEDIAN RESPONSE (5MIN)	TOTAL REQUESTS (24H)	RATE LIMITED % (24H)	INVALID REQUESTS (24H)	
0ms	0	0%	0	
DAILY REQUESTS				

Your .env should look like this:

```
API_URL = "https://eth-goerli.alchemyapi.io/v2/your-api-key"
```

```
PRIVATE_KEY = "your-metamask-private-key"
```

To actually connect these to our code, we'll reference these variables in our hardhat.config.js file on step 13.

Ethers.js is a library that makes it easier to interact and make requests to Ethereum by wrapping standard JSON-RPC methods with more user-friendly methods.

Hardhat makes it super easy to integrate Plugins for additional tooling and extended functionality. We'll be taking advantage of the Ethers plugin for contract deployment (Ethers.js has some super clean contract deployment methods).

In your project directory type:

```
npm install --save-dev @nomiclabs/hardhat-ethers "ethers@^5.0.0"
```

We'll also require ethers in our hardhat.config.js in the next step.

Step 12: Install Ethers.js

Ethers.js is a library that makes it easier to interact and make requests to Ethereum by wrapping standard JSON-RPC methods with more user friendly methods.

Hardhat makes it super easy to integrate Plugins for additional tooling and extended functionality. We'll be taking advantage of the Ethers plugin for contract deployment (Ethers.js has some super clean contract deployment methods).

In your project directory type:

```
npm install --save-dev @nomiclabs/hardhat-ethers "ethers@^5.0.0"
```

We'll also require ethers in our hardhat.config.js in the next step.

Step 13: Update hardhat.config.js We've added several dependencies and plugins so far, now we need to update hardhat.config.js so that our project knows about all of them.

Update your hardhat.config.js to look like this:

```
/**
 * @type import('hardhat/config').HardhatUserConfig
 */
require('dotenv').config();
require("@nomiclabs/hardhat-ethers");
const { API_URL, PRIVATE_KEY } = process.env;
module.exports = {
  solidity: "0.7.3",
  defaultNetwork: "goerli",
  networks: {
    hardhat: {},
    goerli: {
      url: API_URL,
      accounts: [`0x${PRIVATE_KEY}`]
    }
  },
}
```

Step 14: Compile our contract

To make sure everything is working so far, let's compile our contract. The compile task is one of the built-in hardhat tasks.

From the command line run:

```
npx hardhat compile
```

You might get a warning about SPDX license identifier not provided in source file, but no need to worry about that — hopefully everything else looks good!

If not, you can always message in the Alchemy discord.

Step 15: Write our deploy script

Now that our contract is written and our configuration file is good to go, it's time to write our contract deploy script.

Navigate to the /scripts folder and create a new file called deploy.js, adding the following contents to it: sync function main() {

```
    const HelloWorld = await ethers.getContractFactory("HelloWorld");
    // Start deployment, returning a promise that resolves to a contract object
    const hello_world = await HelloWorld.deploy("Hello World!");
    console.log("Contract deployed to address:", hello_world.address);
}
main()
  .then(() => process.exit(0))
  .catch(error => {
    console.error(error);
    process.exit(1);
  });
```

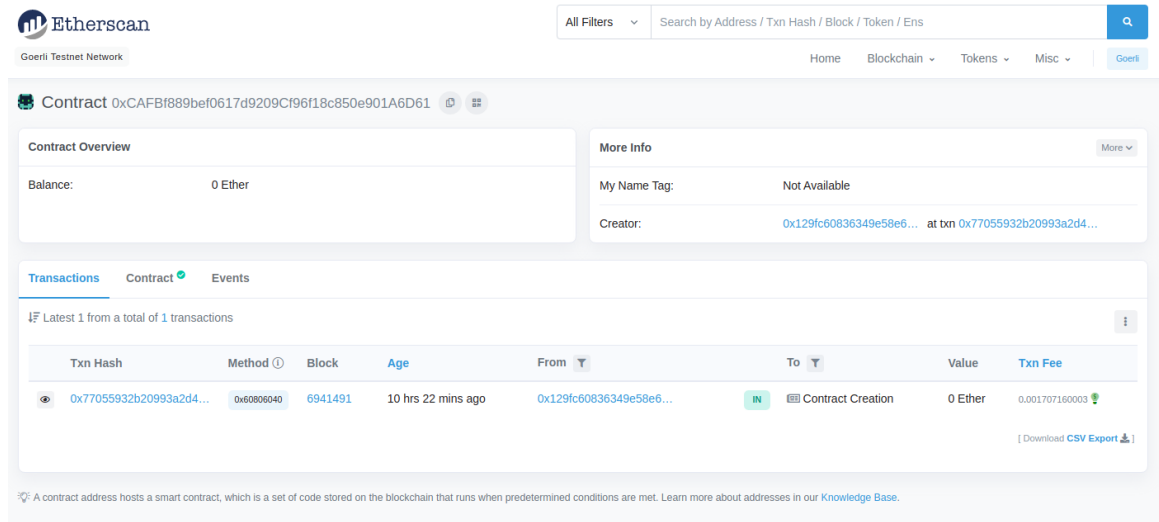
Hardhat does an amazing job of explaining what each of these lines of code does in their Contracts tutorial, we've adopted their explanations here.

```
const HelloWorld = await ethers.getContractFactory("HelloWorld");
```

A ContractFactory in ethers.js is an abstraction used to deploy new smart contracts, so HelloWorld here is a factory for instances of our hello world contract. When using the hardhat-ethers plugin ContractFactory and Contract, instances are connected to the first signer (owner) by default.

```
const hello_world = await HelloWorld.deploy();
```

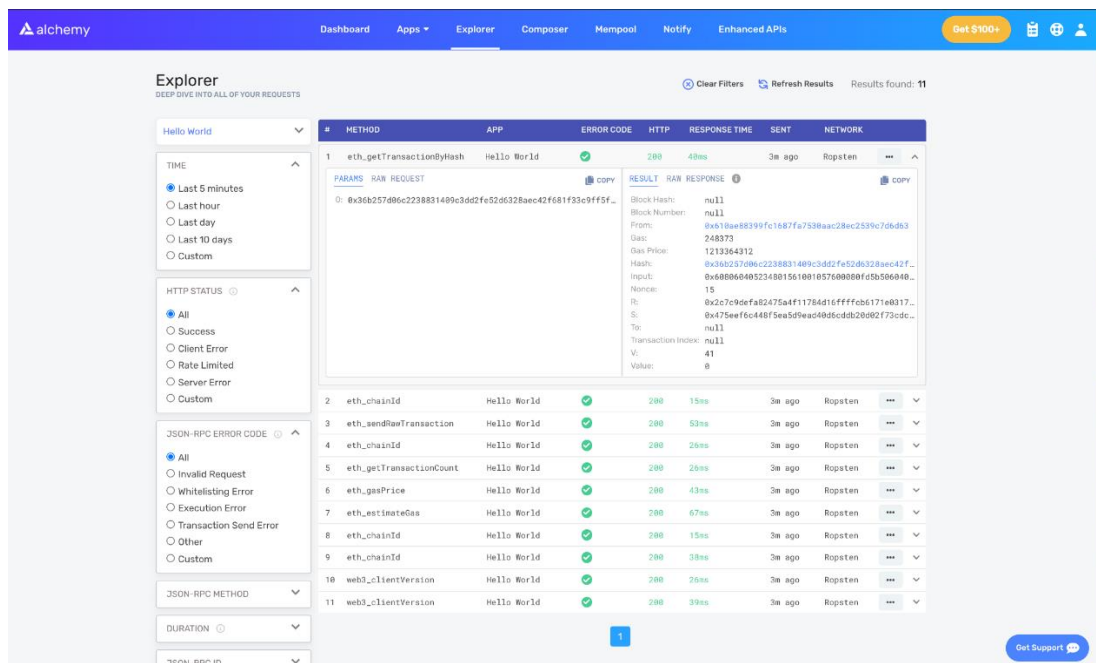
Calling `deploy()` on a `ContractFactory` will start the deployment, and return a `Promise` that resolves to a `Contract` object. This is the object that has a method for each of our smart contract functions.



The screenshot shows the Etherscan interface for a contract at address 0xCAFbF889bef0617d9209Cf96f18c850e901A6D61 on the Goerli Testnet. The 'Contract Overview' section shows a balance of 0 Ether. The 'More Info' section shows 'My Name Tag' as 'Not Available' and 'Creator' as 0x129fc60836349e58e... at tx 0x77055932b20993a2d4... The 'Transactions' tab is active, showing a single transaction with the following details:

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
0x77055932b20993a2d4...	0x60806040	6941491	10 hrs 22 mins ago	0x129fc60836349e58e...	Contract Creation	0 Ether	0.001707160003

The From address should match your Metamask account address and the To address will say “Contract Creation” but if we click into the transaction we’ll see our contract address in the To field:



The screenshot shows the Alchemy Explorer interface for the transaction 0x36b257d86c2238831409c3dd2fe52d6328aec42f681f33c9ff9f... The transaction details are as follows:

METHOD	APP	ERROR CODE	HTTP	RESPONSE TIME	SENT	NETWORK
eth_getTransactionByHash	Hello World	✓	200	48ms	3m ago	Ropsten

The 'PARAMS' section shows the raw request and response. The 'RESULT' section shows the raw response, which includes the contract address 0x129fc60836349e58e... in the 'To' field.

Congrats! You just deployed a smart contract to the Ethereum chain 🎉

To understand what's going on under the hood, let's navigate to the Explorer tab in our Alchemy dashboard . If you have multiple Alchemy apps make sure to filter by app and select "Hello World".

Transaction Details

Overview

State

[This is a Goerli Testnet transaction only]

Transaction Hash:

0x77055932b20993a2d4bc6fa27c2f4240b232f84303e157eca48ca64ad6642171

Status:

Success

Block:

6941491

2497 Block Confirmations

Timestamp:

10 hrs 25 mins ago (May-24-2022 04:05:12 PM +UTC)

From:

0x129fc60836349e58e63552c904eef800f587f3dd

To:

[Contract 0xcabfb889bef0617d9209cf96f18c850e901a6d61 Created]

Value:

0 Ether (\$0.00)

Transaction Fee:

0.001707160003072888 Ether (\$0.00)

Gas Price:

0.000000005000000009 Ether (5.000000009 Gwei)

Click to see More

A transaction is a cryptographically signed instruction from an account that changes the state of the blockchain. Block explorers track the details of all transactions in the