Quickstart: Build a decentralized app (Solidity)

Want to use Rust instead? Head over to the Stylus quickstart if you'd like to use Rust instead of Solidity. This quickstart is for web developers who want to start building decentralized applications (dApps) using Arbitrum . It makes no assumptions about your prior experience with Ethereum, Arbitrum, or Solidity. Familiarity with Javascript and yarn is expected. If you're new to Ethereum, consider studying the Ethereum documentation before proceeding.

What we're building

We're going to build a digital cupcake vending machine using Solidity smart contracts[1] . Our vending machine will follow two rules:

1. The vending machine will distribute a cupcake to anyone who hasn't recently received one.
2. The vending machine's rules can't be changed by anyone.

Here's our vending machine implemented with Javascript. To use it, enter a name in the form below and press the 'Cupcake please!' button, you should see your cupcake balance go up.

Free Cupcakes

web2 Name

Contract address

Cupcake please! Refresh balance Cupcake balance: 0(no name) Note that although this vending machine appears to follow the rules, it doesn't follow them as much as we'd like. The vending machine's business logic and data are hosted by a centralized service provider . We're trusting that this service provider isn't malicious, but:

1. Our centralized service provider can deny access to particular users.
2. A malicious actor can change the rules of the vending machine at any time, for example, to give their friends extra cupcakes.

Centralized third-party intermediaries represent a single point of failure that malicious actors may become incentivized to exploit. To mitigate this type of risk, we can decentralize our vending machine's business logic and data , rendering this type of exploitation infeasible.

This is Arbitrum's core value proposition to you, dear developer. Arbitrum makes it easy for you to deploy your vending machines to Ethereum's permissionless, trustless ,decentralized network of nodes[2] while keeping costs low for you and your users.

Let's implement the "web3" version of the above vending machine using Arbitrum.

Prerequisites

- VS Code
- : The IDE we'll use to build our vending machine. See code.visualstudio.com
- to install.
- Metamask
- : The wallet
- we'll use to interact with our vending machine. See metamask.io
- to install.
- Yarn
- : The package manager we'll use to install our dependencies. See yarnpkg.com
- to install.

We'll address remaining dependencies as we go.

Ethereum and Arbitrum in a nutshell

- Ethereum
-
  - Ethereum is a decentralized network of nodes
-
  - that use Ethereum's client software (like Offchain's Prysm
-
  -) to maintain a public blockchain
-
  - data structure.

- The data within Ethereum's blockchain data structure changes one transaction at a time.
- Smart contracts
- are small programs that execute transactions according to predefined rules. Ethereum's nodes host and execute smart contracts.
- You can use smart contracts to build decentralized apps (dApps) that use Ethereum's network to process transactions and store data.
- DApps let users carry their data and identity between applications without having to trust centralized service providers.
- People who run Ethereum validator nodes[3]
- can earn ETH for processing and validating transactions on behalf of users and dApps.
- These transactions can be expensive when the network is under heavy load.

- Arbitrum
  - Arbitrum is a suite of L2 scaling solutions for dApp developers.
  - Arbitrum One
  - is an L2 chain that implements theArbitrum Rollup protocol
  - .
  - You can use Arbitrum One to build user-friendly dApps with high throughput, low latency, and low transaction costs while inheriting Ethereum's high security standards[4]
  - .

Let's review our vending machine's Javascript implementation, then convert it into a Solidity smart contract, then deploy it to Arbitrum One.

We'll ask your smart contract for cupcakes using the vending machines on this page.

Review our Javascript vending machine

Here's our vending machine implemented as a Javascript class:

VendingMachine.js class

VendingMachine

{ // state variables = internal memory of the vending machine cupcakeBalances =

{ } ; cupcakeDistributionTimes =

{ } ;

// Vend a cupcake to the caller giveCupcakeTo (userId)

{ if

(this . cupcakeDistributionTimes [userId]

===

undefined)

{ this . cupcakeBalances [userId]

=

0 ; this . cupcakeDistributionTimes [userId]

=

0 ; }

// Rule 1: The vending machine will distribute a cupcake to anyone who hasn't recently received one. const fiveSeconds =

5000 ; const userCanReceiveCupcake =

this . cupcakeDistributionTimes [userId]

+ fiveSeconds <=

Date . now ( ) ; if

(userCanReceiveCupcake)

{ this . cupcakeBalances [userId] ++ ; this . cupcakeDistributionTimes [userId]

=

Date . now ( ) ; console . log (Enjoy your cupcake, { userId } !) ; return

true ; }

else

{ console . error ('HTTP 429: Too Many Cupcakes (you must wait at least 5 seconds between cupcakes)' ,) ; return

false ; } }

getCupcakeBalanceFor (userId)

{ return

this . cupcakeBalances [userId] ; } } TheVendingMachine class usesstate variables andfunctions to implementpredefined rules . This implementation is useful because it automates cupcake distribution, but there's a problem: it's hosted by a centralized server controlled by a third-party service provider.

Let's decentralize our vending machine's business logic and data by porting the above Javascript implementation into a Solidity smart contract.

Configure your project directory

Create adecentralized-cupcakes directory for your project and installhardhat using VS Code's integrated terminal:

mkdir decentralized-cupcakes cd decentralized-cupcakes yarn init -y yarn add hardhat @nomicfoundation/hardhat-toolbox -D This installs two packages:hardhat lets us write, test and deploy our smart contracts, andhardhat-toolbox is a bundle of popular Hardhat plugins that we'll use later.

Next, runyarn hardhat init to configure Hardhat. SelectCreate a JavaScript project when prompted. Make sure you specify yourdecentralized-cupcakes directory as the project root when asked.

At this point, you should see the following items (among others) in yourdecentralized-cupcakes project directory:

Item Description contracts/ Contains your smart contracts. You should see theLock.sol contract here. scripts/ Contains scripts that you can use to interact with your smart contracts. You should seedeploy.js here. hardhat.config.js Contains the configuration settings for Hardhat. Replace the contents ofhardhat.config.js with the following:

hardhat.config.js require ('@nomicfoundation/hardhat-toolbox') ;

// NEVER record important private keys in your code - this is for demo purposes const

SEPOLIA_TESTNET_PRIVATE_KEY

=

'' ; const

ARBITRUM_MAINNET_TEMPORARY_PRIVATE_KEY

=

'' ;

/* *@type import('hardhat/config').HardhatUserConfig* / module . exports

=

{ solidity :

'0.8.18' , networks :

{ hardhat :

{ chainId :

1337 , } , arbitrumSepolia :

{ url :

'https://sepolia-rollup.arbitrum.io/rpc' , chainId :

421614 , //accounts: [Sepolia_TESTNET_PRIVATE_KEY] } , arbitrumOne :

{ url :

'https://arb1.arbitrum.io/rpc' , //accounts: [ARBITRUM_MAINNET_TEMPORARY_PRIVATE_KEY] } , } , } ; Before compiling the defaultcontracts , you will need to install additional dependencies. Runyarn hardhat compile and expect it to fail for the first time — follow those instructions, then runyarn hardhat compile again until it runs successfully. You should seeCompiled 1 Solidity file successfully in the terminal output. You should also see a newdecentralized-cupcakes/artifacts/ directory. This directory contains the compiled smart contract.

Openscripts/deploy.js and replace its contents with the following:

scripts/deploy.js const hre =

require ('hardhat') ;

async

function

main ( )

{ const vendingMachine =

await hre . ethers . deployContract ('VendingMachine') ; await vendingMachine . waitForDeployment ( ) ; console . log (Cupcake vending machine deployed to { vendingMachine . target }) ; }

main ( ) . catch ((error)

=>

{ console . error (error) ; process . exit (1) ; }) ; We'll use this to deploy our smart contract in a moment. Next, deletecontracts/Lock.sol and replace it withcontracts/VendingMachine.sol , the smarter alternative to our Javascript implementation:

VendingMachine.sol pragma

solidity

^ 0.8.9 ;

// Rule 2: The vending machine's rules can't be changed by anyone. contract

VendingMachine

{ // state variables = internal memory of the vending machine mapping (address

=>

uint)

private _cupcakeBalances ; mapping (address

=>

uint)

private _cupcakeDistributionTimes ;

function

giveCupcakeTo (address userAddress)

public

returns

(bool)

{ // this code is unnecessary, but we're keeping it here so you can compare it to the JS implementation if

(_cupcakeDistributionTimes [userAddress]

==

0)

{ _cupcakeBalances [userAddress]

=

0 ; _cupcakeDistributionTimes [userAddress]

=

0 ; }

// Rule 1: The vending machine will distribute a cupcake to anyone who hasn't recently received one. uint fiveSecondsFromLastDistribution = _cupcakeDistributionTimes [userAddress]

+

5 seconds ; bool userCanReceiveCupcake = fiveSecondsFromLastDistribution <= block . timestamp ; if

(userCanReceiveCupcake)

{ _cupcakeBalances [userAddress] ++ ; _cupcakeDistributionTimes [userAddress]

= block . timestamp ; return

true ; }

else

{ revert ("HTTP 429: Too Many Cupcakes (you must wait at least 5 seconds between cupcakes)") ; } }

// Getter function for the cupcake balance of a user function

getCupcakeBalanceFor (address userAddress)

public

view

returns

(uint)

{ return _cupcakeBalances [userAddress] ; } } Note that this smart contract is written in Solidity, a language that compiles toEVM bytecode . This means that it can be deployed to any Ethereum-compatible blockchain, including Ethereum mainnet,Arbitrum One , andArbitrum Nova .

Runyarn hardhat compile again. You should seeCompiled 1 Solidity file successfully in the terminal output. You should also see a newdecentralized-cupcakes/artifacts/contracts/VendingMachine.sol directory.

Deploy the smart contract locally

To deploy ourVendingMachine smart contract locally, we'll use two terminal windows and a wallet:

1. We'll use the first terminal window to run Hardhat's built-in local Ethereum node
2. We'll then configure a wallet so we can interact with our smart contract after it's deployed to (1)
3. We'll then use the second terminal window to deploy our smart contract to (1)'s node

Run a local Ethereum network and node

Runyarn hardhat node from yourdecentralized-cupcakes directory to begin running a local Ethereum network powered by a single node. This will mimic Ethereum's behavior on your local machine by using Hardhat's built-inHardhat Network .

You should see something along the lines ofStarted HTTP and WebSocket JSON-RPC server at http://127.0.0.1:8545/ in your terminal. You should also see a number of test accounts automatically generated for you:

... Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 (10000 ETH) Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 ... Never share your private keys Your Ethereum Mainnet wallet's private key is the password to all of your money. Never share it with anyone; avoid copying it to your clipboard. Note that in the context of this quickstart, "account" refers to a public wallet address and its associated private key5 .

Configure Metamask

Next, open Metamask and create or import a wallet by following the displayed instructions. By default, Metamask will connect to Ethereum mainnet. To connect to our local "testnet", enable test networks for Metamask by clickingShow/hide test networks from the network selector dropdown. Then select theLocalhost 8545 network:

Your mainnet wallet won't have a balance on your local testnet's node, but we can import one of the test accounts into Metamask to gain access to 10,000 fake ETH. Copy the private key of one of the test accounts (it works with or without the0x prefix, so e.g.0xac0..f80 orac0..f80) and import it into Metamask:

You should see a balance of 10,000 ETH. Keep your private key handy; we'll use it again in a moment.

Next, click Metamask's network selector dropdown, and then click theAdd Network button. Click "Add a network manually" and then provide the following information:

- Network Name:Arbitrum Sepolia
- New RPC URL:https://sepolia-rollup.arbitrum.io/rpc
- Chain ID:421614
- Currency Symbol:ASPL

As we interact with our cupcake vending machine, we'll use Metamask's network selector dropdown to determine which network our cupcake transactions are sent to. For now, we'll leave the network set toLocalhost 8545 .

Deploy the smart contract to your local testnet

From another terminal instance, runyarn add --dev @nomicfoundation/hardhat-ethers ethers hardhat-deploy hardhat-deploy-ethers to install additional dependencies needed for contract deployment. Then runyarn hardhat run scripts/deploy.js --network localhost . This command will deploy your smart contract to the local testnet's node. You should see something likeCupcake vending machine deployed to 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512 in your terminal.0xe7...512 is the address of your smart contract in your local testnet.

Ensure that theLocalhost network is selected within Metamask. Then copy and paste yourcontract address below and clickGet cupcake! . You should be prompted to sign a transaction that gives you a cupcake.

Free Cupcakes

web3-localhost Metamask wallet address

Contract address

Cupcake please! Refresh balance Cupcake balance: 0(no name)

What's going on here?

Our firstVendingMachine is labeledWEB2 because it demonstrates traditional n-tier web application architecture:

TheWEB3-LOCALHOST architecture is similar to theWEB2 architecture, with one key difference: with theWEB3 version,the business logic and data live in an (emulated for now) decentralized network of nodes instead of a centralized network of servers.

Let's take a closer look at the differences between ourVendingMachine implementations:

WEB2

(the first one) WEB3-LOCALHOST

(the latest one) WEB3-ARB-SEPOLIA

(the next one) WEB3-ARB-MAINNET

(the final one) Data (cupcakes) Stored only in yourbrowser . (Usually, stored by centralized infrastructure.) Stored on yourdevice in anemulated Ethereum network (via smart contract). Stored on Ethereum'sdecentralized test network (via smart contract). Stored on Ethereum'sdecentralized mainnet network (via smart contract). Logic (vending) Served fromOffchain's servers . Executed by yourbrowser . Stored and executed by yourlocally emulated Ethereum network (via smart contract). Stored and executed by Arbitrum'sdecentralized test network (via smart contract). Stored and executed by Arbitrum'sdecentralized mainnet network (via smart contract). Presentation (UI) Served fromOffchain's servers . Rendered and executed by yourbrowser . Money Devs and users pay centralized service providers for server access using fiat currency. ← same, but only for the presentation-layer concerns (code that supports frontend UI/UX). ← same, but devs and users paytestnet ETH to testnet validators. ← same, but instead of testnet ETH, they usemainnet ETH . Next, we'll deploy our smart contract to a network of real nodes: Arbitrum's Sepolia testnet.

Deploy the smart contract to the Arbitrum Sepolia testnet

We were able to deploy to a local testnet for free because we were usingHardhat's built-in Ethereum network emulator . Arbitrum's Sepolia testnet is powered by a real network of real nodes, so we'll need to pay a small transaction fee to deploy our smart contract. This fee can be paid with the Arbitrum Sepolia testnet's token, ASPL.

ASPL IS SHORTHAND "ASPL" isn't a canonical term. It's just shorthand for "Arbitrum Sepolia testnet ETH" that we use for convenience. First, update thehardhat.config.js file to specify the private key of the test account that you'll use to deploy your smart contract (and pay the transaction fee):

hardhat.config.js // ... const

SEPOLIA_TESTNET_PRIVATE_KEY

=

" ;

// <- this should **not** begin with "0x" // ... accounts :

[SEPOLIA_TESTNET_PRIVATE_KEY] ;

// <- uncomment this line // ... caution Note that we're adding a private key to a config file. This isnot a best practice; we're doing it here for convenience. Consider using environment variables when working on a real project. Next, let's deposit some ASPL into the wallet corresponding to the private key we added tohardhat.config.js . At the time of this quickstart's writing, the easiest way to acquire ASPL is to bridge Sepolia ETH from Ethereum's L1 Sepolia network to Arbitrum's L2 Sepolia network:

1. Use an L1 Sepolia ETH faucet likesepoliafaucet.com
2. to acquire some testnet ETH on L1 Sepolia.
3. Bridge your L1 Sepolia ETH into Arbitrum L2 usingthe Arbitrum bridge
4. .

Once you've acquired some ASPL, you'll be able to deploy your smart contract to Arbitrum's Sepolia testnet by issuing the following command:

yarn hardhat run scripts/deploy.js --network arbitrumSepolia This tells hardhat to deploy the compiled smart contract through the RPC endpoint corresponding toarbitrumSepolia inhardhat.config.js . You should see the following output:

Cupcake vending machine deployed to 0xff825139321bd8fB8b720BfFC5b9EfDB7d6e9AB3 Congratulations! You've just deployedbusiness logic and data to Arbitrum Sepolia. This logic and data will be hashed and submitted within a transaction to Ethereum's L1 Sepolia network, and then it will be mirrored across all nodes in the Sepolia network6 .

To view your smart contract in a blockchain explorer, visithttps://sepolia.arbiscan.io/address/0x...B3 , but replace the0x...B3 part of the URL with the full address of your deployed smart contract.

SelectArbitrum Sepolia from Metamask's dropdown, paste your contract address into theVendingMachine below, and clickGet cupcake! . You should be prompted to sign a transaction that gives you a cupcake.

Free Cupcakes

web3-arb-sepolia Metamask wallet address

Contract address

Cupcake please! Refresh balance Cupcake balance: 0(no name)

Deploy the smart contract to Arbitrum One Mainnet

Now that we've verified that our smart contract works on Arbitrum's Sepolia testnet, we're ready to deploy it to Arbitrum One Mainnet. This is the same process as deploying to Arbitrum's Sepolia testnet, except that we'll need to pay a transaction fee in real ETH instead of ASPL.

Expect to see inconsistent ETH gas fees in this step - theGas and fees section contains more information about how gas fees are determined for Arbitrum transactions.

First, update thehardhat.config.js file to specify the private key of theone-time-use deployment account that you'll use to deploy your smart contract (and pay the transaction fee):

hardhat.config.js // ... const

ARBITRUM_MAINNET_TEMPORARY_PRIVATE_KEY

=

" ;

// <- this should **not** begin with "0x" // ... accounts :

[ARBITRUM_MAINNET_TEMPORARY_PRIVATE_KEY] ;

// <- uncomment this line // ... caution Note that we're adding a private key to a config file. This isnot a best practice. Consider using environment variables instead. Next, deposit some ETH into the wallet corresponding to the private key we added tohardhat.config.js . You'll then be able to deploy your smart contract to Arbitrum One Mainnet by issuing the following command:

yarn hardhat run scripts/deploy.js --network arbitrumOne You should see the following output:

Cupcake vending machine deployed to 0xff825139321bd8fB8b720BfFC5b9EfDB7d6e9AB3 Congratulations! You've just deployedbusiness logic and data to Ethereum's decentralized network of nodes by way of Arbitrum One2 .

To view your smart contract in a blockchain explorer, visithttps://arbiscan.io/address/0x...B3 , but replace the0x...B3 part of the URL with the full address of your deployed smart contract.

SelectArbitrum One from Metamask's dropdown, paste your contract address into theVendingMachine below, and clickGet cupcake! . You should be prompted to sign a transaction that gives you an immutable cupcake.

Free Cupcakes

web3-arb-one Metamask wallet address

Contract address

Cupcake please! Refresh balance Cupcake balance: 0(no name)

Summary

In this quickstart, we:

- Identifiedtwo business rules
- : 1) fair and permissionless cupcake distribution, 2) immutable business logic and data.
- Identified achallenge
- : These rules are difficult to follow in a centralized application.
- Identified asolution
- : Arbitrum makes it easy for developers to decentralize business logic and data (using Ethereum mainnet as a settlement layer).
- Converted a vending machine's Javascript business logic into aSolidity smart contract
- .
- Deployed our smart contract
- to Hardhat's local development network, and then Arbitrum's Sepolia testnet, and then Arbitrum One Mainnet.

If you have any questions or feedback, reach out to us onDiscord and/or click theRequest an update button at the top of this page - we're listening!

Next steps

- VisitHow to estimate gas
- to learn how to estimate the gas cost of your smart contract transactions.

- VisitRPC endpoints and providers
- for a list of public chains that you can deploy your smart contracts to.
- The vending machine example was inspired byEthereum.org's "Introduction to Smart Contracts"
- , which was inspired byNick Szabo's "From vending machines to smart contracts"
- .↩
- Although application front-ends are usually hosted by centralized services, smart contracts allow the underlying logic and data to be partially or fully decentralized. These smart contracts are hosted and executed by Ethereum's public, decentralized network of nodes. Arbitrum has its own network of nodes that use advanced cryptography techniques to "batch process" Ethereum transactions and then submit them to Ethereum L1, which significantly reduces the cost of using Ethereum. All without requiring developers to compromise on security or decentralization.↩
- There are multiple types of Ethereum nodes. The ones that earn ETH for processing and validating transactions are calledvalidators
- . SeeNodes and Networks
- for a beginner-friendly introduction to Ethereum's node types.↩
- When ourVendingMachine
- contract is deployed to Ethereum, it'll be hosted by Ethereum's decentralized network of nodes. Generally speaking, we won't be able to modify the contract's code after it's deployed.↩
- To learn more about how Ethereum wallets work, seeEthereum.org's introduction to Ethereum wallets
- .↩
- Visit theGentle Introduction to Arbitrum
- for a beginner-friendly introduction to Arbitrum's rollup protocol.↩ Edit this page Last updatedonMar 18, 2024 Previous Get started with Arbitrum Next How to estimate gas in Arbitrum