ies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/arrays/fixed-size-arrays-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
Arrays  
Writing Arrays  
Arrays  
Filtering an Array  
Fixed-Size Arrays  
Array Storage Layout  
Arrays Exercise  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Fixed-Size Arrays  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/arrays/writing-arrays-in-solidity-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
Arrays  
Writing Arrays  
Arrays  
Filtering an Array  
Fixed-Size Arrays  
Array Storage Layout  
Arrays Exercise  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Writing Arrays  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/contracts-and-basic-functions/basic-functions-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Introduction to Contracts  
Hello World  
Basic Types  
Basic Functions Exercise  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Basic Functions Exercise  
Each module in this course will contain exercises in which you are given a specification for a contract without being given specific instructions on how to build the contract. You must use what you've learned to figure out the best solution on your own!  
INFO  
Once you've learned how to deploy your contracts to a test network, you'll be given the opportunity to submit your contract address for review by an onchain unit test. If it passes, you'll receive an NFT pin recognizing your accomplishment.  
You'll deploy and submit this contract in the next module.  
The following exercise asks you to create a contract that adheres to the following stated specifications.  
Contract  
Create a contract called BasicMath. It should not inherit from any other contracts and does not need a constructor. It should have the following two functions:  
Adder  
A function called adder. It must:  
Accept two uint arguments, called \_a and \_b  
Return a uint sum and a bool error  
If \_a + \_b do not overflow, it should return the sum and an error of false  
If \_a + \_b overflow, it should return 0 as the sum, and an error of true  
Subtractor  
A function called subtractor. It must:  
Accept two uint arguments, called \_a and \_b  
Return a uint difference and a bool error  
If \_a - \_b does not underflow, it should return the difference and an error of false  
If \_a - \_b underflows, it should return 0 as the difference, and an error of true  
ON THIS PAGE  
Contract  
Adder  
Subtractor  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/contracts-and-basic-functions/basic-types  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Introduction to Contracts  
Hello World  
Basic Types  
Basic Functions Exercise  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Basic Types  
Solidity contains most of the basic types you are used to from other languages, but their properties and usage are often a little different than other languages and are likely much more restrictive. In particular, Solidity is a very explicit language and will not allow you to make inferences most of the time.  
Objectives  
By the end of this lesson you should be able to:  
Categorize basic data types  
List the major differences between data types in Solidity as compared to other languages  
Compare and contrast signed and unsigned integers  
Common Properties  
In Solidity, types must always have a value and are never undefined, null, or none. Because of this, each type has a default value. If you declare a variable without assigning a value, it will instead have the default value for that type. This property can lead to some tricky bugs until you get used to it.  
uint defaultValue;  
uint explicitValue = 0;  
  
// (defaultValue == explicitValue) <-- true  
Types can be cast from one type to another, but not as freely as you may expect. For example, to convert a uint256 into a int8, you need to cast twice:  
uint256 first = 1;  
int8 second = int8(int256(first));  
DANGER  
Overflow/underflow protection (described below), does not provide protection when casting.  
uint256 first = 256;  
int8 second = int8(int256(first)); // <- The value stored in second is 0  
Boolean  
Booleans can have a value of true or false. Solidity does not have the concept of truthy or falsey, and non-boolean values cannot be cast to bools by design. The short conversation in this issue explains why, and explains the philosophy why.  
Logical Operators  
Standard logical operators (!, &&, ||, ==, !=) apply to booleans. Short-circuiting rules do apply, which can sometimes be used for gas savings since if the first operator in an && is false or || is true, the second will not be evaluated. For example, the following code will execute without an error, despite the divide by zero in the second statement.  
// Bad code for example. Do not use.  
uint divisor = 0;  
if(1 < 2 || 1 / divisor > 0) {  
 // Do something...  
}  
You cannot use any variant of > or < with booleans, because they cannot be implicitly or explicitly cast to a type that uses those operators.  
Numbers  
Solidity has a number of types for signed and unsigned integers, which are not ignored as much as they are in other languages, due to potential gas-savings when storing smaller numbers. Support for fixed point numbers is under development, but is not fully implemented as of version 0.8.17.  
Floating point numbers are not supported and are not likely to be. Floating precision includes an inherent element of ambiguity that doesn't work for explicit environments like blockchains.  
Min, Max, and Overflow  
Minimum and maximum values for each type can be accessed with type(<type>).min and type(<type>).max. For example, type(uint).min is 0, and type(uint).max is equal to 2^256-1.  
An overflow or underflow will cause a transaction to revert, unless it occurs in a code block that is marked as unchecked.  
uint vs. int  
In Solidity, it is common practice to favor uint over int when it is known that a value will never (or should never) be below zero. This practice helps you write more secure code by requiring you to declare whether or not a given value should be allowed to be negative. Use uint for values that should not, such as array indexes, account balances, etc. and int for a value that does need to be negative.  
Integer Variants  
Smaller and larger variants of integers exist in many languages but have fallen out of favor in many instances, in part because memory and storage are relatively cheap. Solidity supports sizes in steps of eight from uint8 to uint256, and the same for int.  
Smaller sized integers are used to optimize gas usage in storage operations, but there is a cost. The EVM operates with 256 bit words, so operations involving smaller data types must be cast first, which costs gas.  
uint is an alias for uint256 and can be considered the default.  
Operators  
Comparisons (<=, <, ==, !=, >=, >) and arithmetic (+, -, \*, /, %, \*\*) operators are present and work as expected. You can also use bit and shift operators.  
uint and int variants can be compared directly, such as uint8 and uint256, but you must cast one value to compare a uint to an int.  
uint first = 1;  
int8 second = 1;  
  
if(first == uint8(second)) {  
 // Do something...  
}  
Addresses  
The address type is a relatively unique type representing a wallet or contract address. It holds a 20-byte value, similar to the one we explored when you deployed your Hello World contract in Remix. address payable is a variant of address that allows you to use the transfer and send methods. This distinction helps prevent sending Ether, or other tokens, to a contract that is not designed to receive it. If that were to happen, the Ether would be lost.  
Addresses are not strings and do not need quotes when represented literally, but conversions from bytes20 and uint160 are allowed.  
address existingWallet = 0xd9145CCE52D386f254917e481eB44e9943F39138;  
Members of Addresses  
Addresses contain a number of functions. balance returns the balance of an address, and transfer, mentioned above, can be used to send ether.  
function getBalance(address \_address) public view returns(uint) {  
 return \_address.balance;  
}  
Later on, you'll learn about call, delegatecall, and staticcall, which can be used to call functions deployed in other contracts.  
Contracts  
When you declare a contract, you are defining a type. This type can be used to instantiate one contract as a local variable inside a second contract, allowing the second to interact with the first.  
Byte Arrays and Strings  
Byte arrays come as both fixed-size and dynamically-sized. They hold a sequence of bytes. Arrays are a little more complicated than in other languages and will be covered in-depth later.  
Strings  
Strings are arrays in Solidity, not a type. You cannot concat them with +, but as of 0.8.12, you can use string.concat(first, second). They are limited to printable characters and escaped characters. Casting other data types to string is at best tricky, and sometimes impossible.  
Generally speaking, you should be deliberate when working with strings inside of a smart contract. Don't be afraid to use them when appropriate, but if possible, craft and display messages on the front end rather than spending gas to assemble them on the back end.  
Enums  
Enums allow you to apply human-readable labels to a list of unsigned integers.  
enum Flavors { Vanilla, Chocolate, Strawberry, Coffee }  
  
Flavors chosenFlavor = Flavors.Coffee;  
Enums can be explicitly cast to and from uint, but not implicitly. They are limited to 256 members.  
Constant and Immutable  
The constant and immutable keywords allow you to declare variables that cannot be changed. Both result in gas savings because the compiler does not need to reserve a storage slot for these values.  
As of 0.8.17, constant and immutable are not fully implemented. Both are supported on value types, and constant can also be used with strings.  
Constant  
Constants can be declared at the file level, or at the contract level. In Solidity, modifiers come after the type declaration. You must initialize a value when declaring a constant. Convention is to use SCREAMING\_SNAKE\_CASE for constants.  
uint constant NUMBER\_OF\_TEAMS = 10;  
  
contract Cars {  
 uint constant NUMBER\_OF\_CARS = 20;  
}  
At compilation, the compiler replaces every instance of the constant variable with its literal value.  
Immutable  
The immutable keyword is used to declare variables that are set once within the constructor, which are then never changed:  
contract Season {  
 immutable numberOfRaces;  
  
 constructor(uint \_numberOfRaces) {  
 numberOfRaces = \_numberOfRaces;  
 }  
}  
Conclusion  
You've learned the usage and some of the unique quirks of common variable types in Solidity. You've seen how overflow and underflow are handled and how that behavior can be overridden. You've learned why unsigned integers are used more commonly than in other languages, why floats are not present, and have been introduced to some of the quirks of working with strings. Finally, you've been introduced to the address and contract data types.  
ON THIS PAGE  
Common Properties  
Boolean  
Numbers  
Min, Max, and Overflow  
Addresses  
Members of Addresses  
uint vs. int  
Integer Variants  
Operators  
Contracts  
Members of Addresses  
Byte Arrays and Strings  
Enums  
Strings  
Constant and Immutable  
Conclusion  
Constant  
Immutable  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/contracts-and-basic-functions/hello-world-step-by-step  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Introduction to Contracts  
Hello World  
Basic Types  
Basic Functions Exercise  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Hello World  
As is tradition, we'll begin coding with a variant of "Hello World" written as a smart contract. There isn't really a console to write to\*, so instead, we'll write a contract that says hello to the sender, using the name they provide.  
\*You will be able to use console.log with Hardhat, with some restrictions.  
Objectives  
By the end of this lesson you should be able to:  
Construct a simple "Hello World" contract  
List the major differences between data types in Solidity as compared to other languages  
Select the appropriate visibility for a function  
Hello World  
Writing "Hello World" in a smart contract requires a little more consideration than in other languages. Your code is deployed remotely, but it isn't running on a server where you can access logs, or on your local machine where you have access to a console. One way to do it is to write a function that returns "Hello World".  
Creating the Contract  
To create a contract:  
Create a new workspace in Remix.  
Name it Hello World and delete the .deps folder.  
Leave .prettierrc.json and click the settings gear in the bottom left.  
Uncheck the top option to Generate contract metadata...  
Open the Solidity Compiler plugin and enable Auto compile.  
Create a new folder called contracts, and within that folder, create a file called hello-world.sol.  
Solidity files usually start with a comment containing an SPDX-License-Identifier. It's not a requirement, but there are a couple of advantages to doing this. First, everything you deploy on the blockchain is public. This doesn't mean you are giving away everything you deploy for free, nor does it mean you have the right to use the code from any deployed contract. The license determines allowed usage and is generally protected by international copyright laws, the same as any other code.  
If you don't want to give a license, you can put UNLICENSED. Common open source licenses, such as MIT and GPL-3.0 are popular as well. Add your license identifier:  
// SPDX-License-Identifier: MIT  
Below the license identifier, you need to specify which version of Solidity the compiler should use to compile your code. If by the time you read this, the version has advanced, you should try to use the most current version. Doing so may cause you to run into unexpected errors, but it's great practice for working in real-world conditions!  
pragma solidity 0.8.17;  
Finally, add a contract called HelloWorld. You should end up with:  
// SPDX-License-Identifier: MIT  
  
pragma solidity 0.8.17;  
  
contract HelloWorld {  
  
}  
SayHello Function  
Add a function to your contract called SayHello:  
function SayHello() {  
  
}  
You'll get a compiler syntax error for No visibility specified. Did you intend to add "public"?.  
Is public the most appropriate visibility specifier?  
It would work, but you won't be calling this function from within the contract, so external is more appropriate.  
You also need to specify a return type, and we've decided this function should return a string. You'll learn more about this later, but in Solidity, many of the more complex types require you to specify if they are storage or memory. You can then have your function return a string of "Hello World!.  
Don't forget your semicolon. They're mandatory in Solidity!  
You should have:  
function SayHello() external returns (string memory) {  
 return "Hello World!";  
}  
Before you deploy, check the Compiler plugin. You've got one last warning:  
Warning: Function state mutability can be restricted to pure  
Modifiers are used to modify the behavior of a function. The pure modifier prevents the function from modifying, or even accessing state. While not mandatory, using these modifiers can help you and other programmers know the intention and impact of the functions you write. Your final function should be similar to:  
function SayHello() external pure returns (string memory) {  
 return "Hello World!";  
}  
Deployment and Testing  
Confirm that there is a green checkmark on the icon for the compiler plugin, and then switch to the Deploy & Run Transactions plugin.  
Click the Deploy button and your contract should appear in Deployed Contracts. Open it up and then click the SayHello button. Did it work?  
You should see your message below the button. Another option to see the return for your HelloWorld function is to expand the entry in the console. You should see a decoded output of:  
{  
 "0": "string: Hello World!"  
}  
Greeter  
Now, let's modify your say hello function to greet a user by name, instead of just saying "Hello World!"  
First Pass Attempt  
You'd probably expect this to be pretty easy. Start by changing the name of the function (or adding a new one) to Greeter and giving it a parameter for a string memory \_name. The underscore is a common convention to mark functions and variables as internal to their scope. Doing so helps you tell the difference between a storage variable, and a memory variable that only exists within the call.  
Finally, try creating a return string similar to how you might in another language with "Hello " + \_name. You should have:  
// Bad code example: Does not work  
function Greeter(string memory \_name) external pure returns (string memory) {  
 return "Hello " + \_name + "!";  
}  
Unfortunately, this does not work in Solidity. The error message you receive is a little confusing:  
TypeError: Operator + not compatible with types literal\_string "Hello " and string memory.  
You might think that there is some sort of type casting or conversion error that could be solved by explicitly casting the string literal to string memory, or vice versa. This is a great instinct. Solidity is a very explicit language.  
However, you receive a similar error with "Hello " + "world".  
String concatenation is possible in Solidity, but it's a bit more complicated than most languages, for good reason. Working with string costs a large amount of gas, so it's usually better to handle this sort of processing on the front end.  
Plan B  
We still want to return something with the name provided by the user, so let's try something a little different. Solidity is a variadic language, which means it allows functions to return more than one value.  
Modify your return declaration: returns (string memory, string memory)  
Now, your function can return a tuple containing two strings!  
return ("Hello", \_name);  
Deploy and test your contract. You should get a decoded output with:  
{  
 "string \_name": "Your Name"  
}  
Full Example Code  
// SPDX-License-Identifier: MIT  
  
pragma solidity 0.8.17;  
  
contract HelloWorld {  
  
 function SayHello() external pure returns (string memory) {  
 return "Hello World!";  
 }  
  
 // Bad code example: Does not work  
 // function Greeter(string memory \_name) external pure returns (string memory) {  
 // return "Hello " + \_name;  
 // }  
  
 function Greeter(string memory \_name) external pure returns (string memory, string memory) {  
 return ("Hello", \_name);  
 }  
}  
Conclusion  
Congratulations! You've written and tested your first smart contract! You selected a license and a version of Solidity. You declared a contract and added a function that returns a value.  
You also learned more about some of the ways in which Solidity is more challenging to work with than other languages, and the additional elements you sometimes need to declare functions and types.  
ON THIS PAGE  
Hello World  
Greeter  
First Pass Attempt  
Plan B  
Full Example Code  
Conclusion  
First Pass Attempt  
Plan B  
Full Example Code  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/contracts-and-basic-functions/intro-to-contracts-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Introduction to Contracts  
Hello World  
Basic Types  
Basic Functions Exercise  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Introduction to Contracts  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/control-structures/  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
If, Else, and Else If  
Loops  
Require, Revert, and Error  
Control Structures  
Control Structures Exercise  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Control Structures  
Solidity supports many familiar control structures, but these come with additional restrictions and considerations due to the cost of gas and the necessity of setting a maximum amount of gas that can be spent in a given transaction.  
Objectives  
By the end of this lesson you should be able to:  
Control code flow with if, else, while, and for  
List the unique constraints for control flow in Solidity  
Utilize require to write a function that can only be used when a variable is set to true  
Write a revert statement to abort execution of a function in a specific state  
Utilize error to control flow more efficiently than with require  
Control Structures  
Solidity supports the basic conditional and iterative control structures found in other curly bracket languages, but it does not support more advanced statements such as switch, forEach, in, of, etc.  
Solidity does support try/catch, but only for calls to other contracts.  
CAUTION  
Yul is an intermediate-level language that can be embedded in Solidity contracts and is documented within the docs for Solidity. Yul does contain the switch statement, which can confuse search results.  
Conditional Control Structure Examples  
The if, else if, and else, statements work as expected. Curly brackets may be omitted for single-line bodies, but we recommend avoiding this as it is less explicit.  
function ConditionalExample(uint \_number) external pure returns (string memory) {  
 if(\_number == 0) {  
 return "The number is zero.";  
 } else if(\_number % 2 == 0) {  
 return "The number is even and greater than zero.";  
 } else {  
 return "The number is odd and is greater than zero.";  
 }  
}  
Iterative Control Structures  
The while, for, and do, keywords function the same as in other languages. You can use continue to skip the rest of a loop and start the next iteration. break will terminate execution of the loop, and you can use return to exit the function and return a value at any point.  
INFO  
You can use console.log by importing import "hardhat/console.sol";. Doing so will require you to mark otherwise pure contracts as view.  
uint times; // Default value is 0!  
for(uint i = 0; i <= times; i++) {  
 console.log(i);  
}  
  
uint timesWithContinue;  
for(uint i = 0; i <= timesWithContinue; i++) {  
 if(i % 2 == 1) {  
 continue;  
 }  
 console.log(i);  
}  
  
uint timesWithBreak;  
for(uint i = 0; i <= timesWithBreak; i++) {  
 // Always stop at 7  
 if(i == 7) {  
 break;  
 }  
 console.log(i);  
}  
  
uint stopAt = 10;  
while(stopAt <= 10) {  
 console.log(i);  
 stopAt++;  
}  
  
uint doFor = 10;  
do {  
 console.log(i);  
 doFor++;  
} while(doFor <= 10);  
Error Handling  
Solidity contains a set of relatively unique, built-in functions and keywords to handle errors. They ensure certain requirements are met, and completely abort all execution of the function and revert any state changes that occurred during function execution. You can use these functions to help protect the security of your contracts and limit their execution.  
The approach may seem different than in other environments. If an error occurs partly through a high-stakes transaction such as transferring millions of dollars of tokens, you do not want execution to carry on, partially complete, or swallow any errors.  
Revert and Error  
The revert keyword halts and reverses execution. It must be paired with a custom error. Revert should be used to prevent operations that are logically valid, but should not be allowed for business reasons. It is not a bug if a revert is triggered. Examples where revert and error would be used to control operations include:  
Allowing only certain senders to access functionality  
Preventing the withdrawal of a deposit before a certain date  
Allowing inputs under certain state conditions and denying them under others  
Custom errors can be declared without parameters, but they are much more useful if you include them:  
error OddNumberSubmitted(uint \_first, uint \_second);  
function onlyAddEvenNumbers(uint \_first, uint \_second) public pure returns (uint) {  
 if(\_first % 2 != 0 || \_second % 2 != 0) {  
 revert OddNumberSubmitted(\_first, \_second);  
 }  
 return \_first + \_second;  
}  
When triggered, the error provides the values in the parameters provided. This information is very useful when debugging, and/or to transmit information to the front end to share what has happened with the user:  
call to HelloWorld.onlyAddEvenNumbers errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
Error provided by the contract:  
OddNumberSubmitted  
Parameters:  
{  
 "\_first": {  
 "value": "1"  
 },  
 "\_second": {  
 "value": "2"  
 }  
}  
Debug the transaction to get more information.  
You'll also encounter revert used as a function, returning a string error. This legacy pattern has been retained to maintain compatibility with older contracts:  
function oldRevertAddEvenNumbers(uint \_first, uint \_second) public pure returns (uint) {  
 if(\_first % 2 != 0 || \_second % 2 != 0) {  
 // Legacy use of revert, do not use  
 revert("One of the numbers is odd");  
 }  
 return \_first + \_second;  
}  
The error provided is less helpful:  
call to HelloWorld.oldRevertAddEvenNumbers errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
The reason provided by the contract: "One of the numbers is odd".  
Debug the transaction to get more information.  
Require  
The require function is falling out of favor because it uses more gas than the pattern above. You should still become familiar with it because it is present in innumerable contracts, tutorials, and examples.  
require takes a logical condition and a string error as arguments. It is more gas efficient to separate logical statements if they are not interdependent. In other words, don't use && or || in a require if you can avoid it.  
For example:  
function requireAddEvenNumbers(uint \_first, uint \_second) public pure returns (uint) {  
 // Legacy pattern, do not use  
 require(\_first % 2 == 0, "First number is not even");  
 require(\_second % 2 != 0, "Second number is not even");  
  
 return \_first + \_second;  
}  
The output error message will be the first one that fails. If you were to submit 1, and 3 to this function, the error will only contain the first message:  
call to HelloWorld.requireAddEvenNumbers errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
The reason provided by the contract: "First number is not even".  
Debug the transaction to get more information.  
Assert and Panic  
The assert keyword throws a panic error if triggered. A panic is the same type of error that is thrown if you try to divide by zero or access an array out-of-bounds. It is used for testing internal errors and should never be triggered by normal operations, even with flawed input. You have a bug that should be resolved if an assert throws an exception:  
function ProcessEvenNumber(uint \_validatedInput) public pure {  
 // If assert triggers, input validation has failed. This should never  
 // happen!  
 assert(\_validatedInput % 2 == 0);  
 // Do something...  
}  
The output here isn't as helpful, so you may wish to use one of the patterns above instead.  
call to HelloWorld.ProcessEvenNumber errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
Note: The called function should be payable if you send value and the value you send should be less than your current balance.  
Debug the transaction to get more information.  
Conclusion  
In this lesson, you've learned how to control code flow with standard conditional and iterative operators. You've also learned about the unique keywords Solidity uses to generate errors and reset changes if one of them has been triggered. You've been exposed to both newer and legacy methods of writing errors, and learned the difference between assert and require.  
ON THIS PAGE  
Control Structures  
Error Handling  
Revert and Error  
Require  
Conclusion  
Revert and Error  
Require  
Assert and Panic  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/control-structures/control-structures-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
If, Else, and Else If  
Loops  
Require, Revert, and Error  
Control Structures  
Control Structures Exercise  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Control Structures Exercise  
Create a contract that adheres to the following specifications:  
Contract  
Create a single contract called ControlStructures. It should not inherit from any other contracts and does not need a constructor. It should have the following functions:  
Smart Contract FizzBuzz  
Create a function called fizzBuzz that accepts a uint called \_number and returns a string memory. The function should return:  
"Fizz" if the \_number is divisible by 3  
"Buzz" if the \_number is divisible by 5  
"FizzBuzz" if the \_number is divisible by 3 and 5  
"Splat" if none of the above conditions are true  
Do Not Disturb  
Create a function called doNotDisturb that accepts a uint called \_time, and returns a string memory. It should adhere to the following properties:  
If \_time is greater than or equal to 2400, trigger a panic  
If \_time is greater than 2200 or less than 800, revert with a custom error of AfterHours, and include the time provided  
If \_time is between 1200 and 1259, revert with a string message "At lunch!"  
If \_time is between 800 and 1199, return "Morning!"  
If \_time is between 1300 and 1799, return "Afternoon!"  
If \_time is between 1800 and 2200, return "Evening!"  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Contract  
Smart Contract FizzBuzz  
Do Not Disturb  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/control-structures/loops-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
If, Else, and Else If  
Loops  
Require, Revert, and Error  
Control Structures  
Control Structures Exercise  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Loops  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/control-structures/require-revert-error-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
If, Else, and Else If  
Loops  
Require, Revert, and Error  
Control Structures  
Control Structures Exercise  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Require, Revert, and Error  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/control-structures/standard-control-structures-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
If, Else, and Else If  
Loops  
Require, Revert, and Error  
Control Structures  
Control Structures Exercise  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
If, Else, and Else If  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/deployment-to-testnet/contract-verification-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Overview of Test Networks  
Test Networks  
Deployment to Base Sepolia  
Contract Verification  
Deployment Exercise  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Contract Verification  
Once your contract is deployed, you can verify it using a number of popular services. Doing so will let you users have confidence that your contract does what you claim, and will allow you to interact with it using a similar interface to what you used in Remix.  
Objectives  
By the end of this lesson you should be able to:  
Verify a contract on the Base Sepolia testnet and interact with it in BaseScan  
Verify the Contract  
Make sure you still have the address of the contract you deployed in the last article copied to the clipboard.  
You can interact with your deployed contract using Remix, the same as before, but it's also possible to interact with it through BaseScan. Paste your address in the search field to find it.  
On this page, you can review the balance, information about, and all the transactions that have ever occurred with your contract.  
Click the Contract tab in the main panel. At the top is a message asking you to Verify and Publish your contract source code.  
Verifying your contract maps the names of your functions and variables to the compiled byte code, which makes it possible to interact with the contract using a human-readable interface.  
Click the link. Your contract address is already entered.  
Under Please select Compiler Type choose \_Solidity (Single file)  
For Please Select Compiler Version select the version matching the pragma at the top of your source file. Our examples are currently using v0.8.17+commit.8df45f5f.  
For Please select Open Source License Type pick the license that matches what you selected for your contract as the SPDX-License-Identifier. Pick None if you followed the Solidity-recommended practice of using UNLICENSED.  
On the next page, copy and paste your source code in the window. Verify that you are not a robot, and click Verify and Publish. You should see a success message.  
Click the linked address to your contract to return to the contract page. You'll now see your code!  
TIP  
If you have imports, you'll need to right click on the name of the file and choose Flatten. Submit the newly generated filename\_flattened.sol for verification.  
Interact with the Contract  
You can now interact with your contract using BaseScan. Click the Read Contract button. Both of your functions will be listed here and can be tested using the web interface.  
You won't have anything under Write Contract because this contract doesn't have any functions that save data to state.  
Conclusion  
With your contracts verified, you can interact with them using online tools and your users can be secure that your code does what you claim.  
ON THIS PAGE  
Conclusion  
Verify the Contract  
Interact with the Contract  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/deployment-to-testnet/deployment-to-base-sepolia-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Overview of Test Networks  
Test Networks  
Deployment to Base Sepolia  
Contract Verification  
Deployment Exercise  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Deployment to Base Sepolia  
Remix contains a simulation of a blockchain that you can use to rapidly deploy and test your contracts. This simulation only exists within your browser so you can't share it with others, use external tools, or a front end to interact with it. However, you can also deploy to a variety of testnets from within Remix. Doing so will allow you to share your contract with others, at the cost of making it public.  
Objectives  
By the end of this lesson you should be able to:  
Deploy a contract to the Base Sepolia testnet and interact with it in BaseScan  
Prepare for Deployment  
Testnets operate in a similar, but not exactly the same manner as the main networks they shadow. You need a wallet with the appropriate token to interact with them by deploying a new contract or calling functions in a deployed contract.  
Set Up a Wallet  
If you already have a wallet set up exclusively for development, you can skip to the next section. Otherwise, now is the time to jump in!  
DANGER  
It is very dangerous to use a wallet with valuable assets for development. You could easily write code with a bug that transfers the wrong amount of the wrong token to the wrong address. Transactions cannot be reversed once sent!  
Be safe and use separate wallets for separate purposes.  
First, add the Coinbase or Metamask wallet to your browser, and then set up a new wallet. As a developer, you need to be doubly careful about the security of your wallet! Many apps grant special powers to the wallet address that is the owner of the contract, such as allowing the withdrawal of all the Ether that customers have paid to the contract or changing critical settings.  
Once you've completed the wallet setup, enable developer settings and turn on testnets (Coinbase Settings, Metamask Settings).  
Add Base Sepolia to your Wallet  
Use the faucet to add Base Sepolia ETH to your wallet. You can also ask Base personnel on Discord or other social media for some!  
Get Testnet Ether  
Testnet tokens have no real value, but the supply is not unlimited. You can use a faucet to get a small amount of Sepolia Ether to pay gas fees for testing. Most faucets allow you to ask for a small amount each day, and some won't send you more if your balance is too high.  
You can find many faucets by searching, and it's good to keep a few bookmarked because they have a tendency to go down from time to time. Faucet providers are constantly combating bad actors and sometimes need to disable their faucets while doing so.  
You can also access the faucets on the web.  
Once you have testnet Base Sepolia Ether, you can view your balance under the Testnets tab in the Coinbase wallet or by selecting the testnet from the network dropdown in Metamask. Sadly, it's not actually worth the amount listed!  
Deploying to Testnet  
Once you have testnet Ether, you can deploy your BasicMath contract!  
Selecting the Environment  
Open the Deploy & Run Transactions tab. Under Environment, select Injected Provider. It will list Coinbase, Metamask, or any other wallet you have activated here.  
If that option is not available, you can add it by choosing Customize this list...  
The first time you do this, your wallet will ask you to confirm that you want to connect this app (Remix) to your wallet.  
Once you are connected, you'll see the name of the network below the Environment dropdown.  
For Base Sepolia, you should see Custom (84532) network. The old network, Goerli, was 84531. If you don't see the correct network, change the active network in your wallet.  
Deploy the Contract  
Click the orange Deploy button. Because it costs gas to deploy a contract, you'll be asked to review and confirm a transaction.  
DANGER  
Always carefully review all transactions, confirming the transaction cost, assets transferred, and network. As a developer, you'll get used to approving transactions regularly. Do the best you can to avoid getting into the habit of clicking Confirm without reviewing the transaction carefully. If you feel pressured to Confirm before you run out of time, it is almost certainly a scam.  
After you click the Confirm button, return to Remix and wait for the transaction to deploy. Copy its address and navigate to sepolia.basescan.org.  
Conclusion  
You now have the power to put smart contracts on the blockchain! You've only deployed to a test network, but the process for real networks is exactly the same - just more expensive!  
ON THIS PAGE  
Prepare for Deployment  
Deploying to Testnet  
Selecting the Environment  
Deploy the Contract  
Get Testnet Ether  
Conclusion  
Selecting the Environment  
Deploy the Contract  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/deployment-to-testnet/deployment-to-testnet-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Overview of Test Networks  
Test Networks  
Deployment to Base Sepolia  
Contract Verification  
Deployment Exercise  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Deployment Exercise  
You've already built and deployed your Basic Math contract for this exercise. Now it's time to submit the address and earn an NFT pin to commemorate your accomplishment!  
CAUTION  
We're currently in beta, so you'll only need to pay testnet funds to submit your contract, but this means you'll be getting a testnet NFT.  
Stay tuned for updates!  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/deployment-to-testnet/overview-of-test-networks-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Overview of Test Networks  
Test Networks  
Deployment to Base Sepolia  
Contract Verification  
Deployment Exercise  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Overview of Test Networks  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/deployment-to-testnet/test-networks  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Overview of Test Networks  
Test Networks  
Deployment to Base Sepolia  
Contract Verification  
Deployment Exercise  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Test Networks  
This article provides a concise overview of Base test networks, highlighting their advantages, potential challenges, and comparing some of the most popular testnets.  
Objectives:  
By the end of this lesson you should be able to:  
Describe the uses and properties of the Base testnet  
Compare and contrast Ropsten, Rinkeby, Goerli, and Sepolia  
Why Testnets?  
As you dive into the development of smart contracts and onchain apps on Base, mainnet, or other networks, you'll need a safe, controlled, and efficient environment for testing and experimentation. Test networks, or testnets, serve as essential tools for you to test your smart contracts before deploying them to the mainnet, minimizing the risk of failures or vulnerabilities in live applications.  
By simulating the mainnet environment, testnets offer a realistic representation of real-world conditions, complete with network latency, gas fees, and other factors that impact the performance of smart contracts. This accurate representation enables you to identify potential issues, optimize your applications, and ensure the best possible user experience for your end-users. Moreover, testnets allow you to familiarize yourself with the Base ecosystem and gain valuable hands-on experience, making them indispensable tools for both seasoned developers and newcomers to the world of blockchain development.  
The Advantages of Using Testnets  
Testnets offer several key advantages to developers:  
Real-time feedback: Developers can quickly identify and fix errors or vulnerabilities in their smart contracts, ensuring robust and secure applications.  
No cost or risk: Testnets use "fake" ether, enabling developers to deploy and interact with smart contracts without incurring any financial cost or risk.  
Easy accessibility: Testnets are readily available for developers to join, allowing them to focus on development rather than infrastructure setup.  
Stress testing: Testnets provide a suitable environment to stress test the Ethereum network infrastructure under various conditions. By simulating high transaction volumes, developers can evaluate how their applications perform under load and optimize them accordingly.  
Protocol upgrades and improvements: Testnets allow developers to test new protocol updates, improvements, and potential forks before implementing them on the mainnet. This process helps identify any issues or incompatibilities and ensures a smoother transition to new features and optimizations.  
Challenges Associated with Testnets  
While Ethereum testnets provide a valuable testing environment for developers, there are some challenges and limitations you should be aware of when using them:  
Network congestion: Just like the mainnet, testnets can experience periods of network congestion due to high transaction volumes or other factors. During these periods, developers might face slow transaction processing times, which could impact their testing process and potentially delay development.  
Testnet instability: Testnets may occasionally face downtime or network issues, which can disrupt the testing process. While these events are relatively rare, it's essential to be prepared for such occurrences and have a backup plan, such as switching to another testnet or using a local development environment.  
Differences in network behavior: Although testnets simulate the mainnet environment, there might be subtle differences in network behavior, gas fees, or transaction processing times between the two. These differences could potentially impact the performance of your smart contracts on the mainnet. It's crucial to be aware of these discrepancies and make any necessary adjustments before deploying your smart contracts to the mainnet.  
Limited resources: Testnet Ether is generally available for free through faucets, but these sources might have daily limits or other restrictions on the amount of testnet Ether that can be obtained. This limitation could affect your ability to perform extensive testing or carry out large-scale experiments on the testnet.  
Popular Testnets  
Several well-known testnets have emerged over the years, each with its own set of features and benefits.  
L1 Testnets  
Ropsten: Ropsten played a significant role in Ethereum's history but was effectively deprecated by late 2022 when the Merge took place. The Merge marked the transition from proof-of-work to proof-of-stake consensus for the Ethereum mainnet. Ropsten's vulnerability to spam attacks and network instability made it unreliable for testing purposes.  
Rinkeby: Rinkeby offered better security than Ropsten and used a proof-of-authority consensus mechanism. However, it lacked decentralization and client diversity, which ultimately led to its decline in popularity. After the Merge, Rinkeby is no longer a recommended test network.  
Goerli: Launched in early 2019, Goerli initially utilized a multi-client proof-of-authority consensus model to improve stability and security. Following the Merge, it transitioned to a proof-of-stake consensus mechanism, maintaining its cross-client compatibility and making it an ideal choice for developers. As of January 2024, Goerli is being sunset in favor of Sepolia.  
Sepolia: As one of the two original primary testnets alongside Goerli, Sepolia is designed for developers seeking a lighter weight chain for faster synchronization and interaction. As of January 2024, it is now the preferred testnet and developers should migrate to using it.  
L2 Testnets  
Base Sepolia: As new Layer-2 networks emerged that settled on Ethereum's Layer-1, the need for testnets dedicated to these L2 networks also arose. For instance, the L2 network Base has its own testnet, known as Base Sepolia. This testnet settles on the Ethereum Sepolia L1 testnet, providing an environment for testing L2-specific features and smart contracts.  
Optimism Sepolia: Optimism, an Ethereum Layer-2 scaling solution utilizing Optimistic Rollups, has its own testnet called Optimism Sepolia. This testnet is also built on the Ethereum Sepolia L1 testnet and offers a testing environment for developers to experiment with Optimism's Layer-2 features, smart contracts, and apps.  
Conclusion  
Ethereum and L2 testnets are essential for the safe and efficient development of smart contracts and apps, offering numerous advantages such as real-time feedback, cost-free testing, easy accessibility, stress testing, and protocol upgrade testing. Despite certain challenges associated with testnets, developers continue to rely on them for a robust testing environment. Among the various options, Sepolia has emerged as preferred choices for Ethereum developers due to enhanced security, stability, and strong community support. As the Ethereum ecosystem evolves, incorporating Layer-2 networks and other innovations, testnets will continue to play a crucial role in fostering blockchain development and contributing to the overall success and growth of the space.  
See Also  
Networks  
The History of Ethereum Testnets  
ON THIS PAGE  
Why Testnets?  
The Advantages of Using Testnets  
Challenges Associated with Testnets  
Popular Testnets  
Conclusion  
L1 Testnets  
L2 Testnets  
See Also  
See Also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/development-tools/overview  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Overview  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Overview  
As the popularity and possibilities of building onchain have increased, so has the number, quality, and ease of setup for a variety of smart contract development environments. Most of the smart contract development content in Base Learn is done in Remix, an online IDE that allows you to dive right into learning without worrying about setup.  
However, the setup of professional tools for local development is far less burdensome than it used to be. Foundry and Hardhat are two popular choices, both with excellent communities.  
You may wish to select and install one of these now, but feel free to skip those sections and go to right Smart Contract Development if you're happy learning with an online editor!  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-20-token/analyzing-erc-20-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
Analyzing the ERC-20 Standard  
The ERC-20 Token Standard  
OpenZeppelin ERC-20 Implementation  
ERC-20 Testing  
ERC-20 Implementation  
ERC-20 Tokens Exercise  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Analyzing the ERC-20 Standard  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-20-token/erc-20-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
Analyzing the ERC-20 Standard  
The ERC-20 Token Standard  
OpenZeppelin ERC-20 Implementation  
ERC-20 Testing  
ERC-20 Implementation  
ERC-20 Tokens Exercise  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
ERC-20 Tokens Exercise  
Create a contract that adheres to the following specifications.  
Contract  
Create a contract called WeightedVoting. Add the following:  
A maxSupply of 1,000,000  
Errors for:  
TokensClaimed  
AllTokensClaimed  
NoTokensHeld  
QuorumTooHigh, returning the quorum amount proposed  
AlreadyVoted  
VotingClosed  
A struct called Issue containing:  
An OpenZeppelin Enumerable Set storing addresses called voters  
A string issueDesc  
Storage for the number of votesFor, votesAgainst, votesAbstain, totalVotes, and quorum  
Bools storing if the issue is passed and closed  
CAUTION  
The unit tests require this struct to be constructed with the variables in the order above.  
An array of Issues called issues  
An enum for Vote containing:  
AGAINST  
FOR  
ABSTAIN  
Anything else needed to complete the tasks  
Add the following functions.  
Constructor  
Initialize the ERC-20 token and burn the zeroeth element of issues.  
Claim  
Add a public function called claim. When called, so long as a number of tokens equalling the maximumSupply have not yet been distributed, any wallet that has not made a claim previously should be able to claim 100 tokens. If a wallet tries to claim a second time, it should revert with TokensClaimed.  
Once all tokens have been claimed, this function should revert with an error AllTokensClaimed.  
CAUTION  
In our simple token, we used totalSupply to mint our tokens up front. The ERC20 implementation we're using also tracks totalSupply, but does it differently.  
Review the docs and code comments to learn how.  
Create Issue  
Implement an external function called createIssue. It should add a new Issue to issues, allowing the user to set the description of the issue, and quorum - which is how many votes are needed to close the issue.  
Only token holders are allowed to create issues, and issues cannot be created that require a quorum greater than the current total number of tokens.  
This function must return the index of the newly-created issue.  
CAUTION  
One of the unit tests will break if you place your check for quorum before the check that the user holds a token. The test compares encoded error names, which are not human-readable. If you are getting -> AssertionError: �s is not equal to �9� or similar, this is likely the issue.  
Get Issue  
Add an external function called getIssue that can return all of the data for the issue of the provided \_id.  
EnumerableSet has a mapping underneath, so it can't be returned outside of the contract. You'll have to figure something else out.  
HINT  
The return type for this function should be a struct very similar to the one that stores the issues.  
Vote  
Add a public function called vote that accepts an \_issueId and the token holder's vote. The function should revert if the issue is closed, or the wallet has already voted on this issue.  
Holders must vote all of their tokens for, against, or abstaining from the issue. This amount should be added to the appropriate member of the issue and the total number of votes collected.  
If this vote takes the total number of votes to or above the quorum for that vote, then:  
The issue should be set so that closed is true  
If there are more votes for than against, set passed to true  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
CAUTION  
The contract specification contains actions that can only be performed once by a given address. As a result, the unit tests for a passing contract will only be successful the first time you test.  
You may need to submit a fresh deployment to pass  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Contract  
Constructor  
Claim  
Create Issue  
Get Issue  
Vote  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-20-token/erc-20-standard  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
Analyzing the ERC-20 Standard  
The ERC-20 Token Standard  
OpenZeppelin ERC-20 Implementation  
ERC-20 Testing  
ERC-20 Implementation  
ERC-20 Tokens Exercise  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
The ERC-20 Token Standard  
In this article, we'll delve into the structure and specifications of ERC-20 tokens, uncovering the key elements that contribute to their widespread adoption and diverse use cases.  
Objectives:  
By the end of this lesson you should be able to:  
Analyze the anatomy of an ERC-20 token  
Review the formal specification for ERC-20  
Introduction  
The emergence of the ERC-20 token standard marked a significant milestone in the evolution of the Ethereum ecosystem, providing a unified and adaptable framework for creating and managing fungible tokens. As the backbone for a vast array of applications in decentralized finance and beyond, ERC-20 tokens facilitate seamless collaboration and interoperability within the Ethereum ecosystem. Their adaptable nature and standardized structure have made them the go-to choice for developers and users alike, laying the groundwork for continued innovation and growth in the Ethereum space.  
The ERC-20 token standard has not only streamlined the creation of new tokens but also bolstered the overall user experience by establishing a consistent set of rules and functions. As a result, it has garnered widespread adoption and solidified its position as the de facto standard for fungible tokens on Ethereum, driving the expansion of the decentralized economy and fostering the development of novel applications and services.  
ERC-20 Specification  
EIP-20 (Ethereum Improvement Proposal 20) is the formal specification for ERC-20, defining the requirements to create compliant tokens on the Ethereum blockchain. EIP-20 prescribes the mandatory functions, optional functions, and events a token must implement to achieve ERC-20 compliance. Adherence to EIP-20 allows developers to create tokens compatible with existing Ethereum applications and services, streamlining integration.  
Anatomy of an ERC-20 Token  
An ERC-20 token consists of a smart contract that implements the standardized interface, which comprises a set of six mandatory functions:  
totalSupply(): Returns the total supply of the token.  
balanceOf(address): Provides the balance of tokens held by a specific address.  
transfer(address, uint256): Transfers a specified amount of tokens from the sender's address to the specified recipient's address.  
transferFrom(address, address, uint256): Enables a third party to transfer tokens on behalf of the token owner, given that the owner has approved the transaction.  
approve(address, uint256): Allows the token owner to grant permission to a third party to spend a specified amount of tokens on their behalf.  
allowance(address, address): Returns the amount of tokens the token owner has allowed a third party to spend on their behalf.  
Additionally, ERC-20 tokens can include optional functions that provide descriptive information about the token:  
name(): Returns the name of the token, for example, "Uniswap."  
symbol(): Provides the token's symbol, like "UNI."  
decimals(): Indicates the number of decimal places the token can be divided into, typically 18 for most tokens.  
Benefits of ERC-20 Standardization  
The standardization of ERC-20 tokens provides several benefits for both developers and users. For developers, it simplifies the process of creating new tokens by providing a consistent set of functions and conventions. This reduces the likelihood of errors and ensures a smooth integration with existing applications and services in the Ethereum ecosystem.  
For users, the standardized interface makes it easier to interact with a wide variety of tokens, regardless of their specific purpose or implementation. This means that users can effortlessly check their token balance, transfer tokens, or approve transactions using the same set of functions, whether they are interacting with a governance token like UNI or a stablecoin like USDC.  
Applications  
ERC-20 tokens find wide-ranging applications in various categories, each with its unique purpose and functionality:  
Utility tokens: These tokens can be used to access specific services or features within a platform. Examples include Filecoin (FIL) for decentralized storage, Basic Attention Token (BAT) for digital advertising, and Decentraland's MANA for purchasing virtual land and assets.  
Governance tokens: These tokens grant voting rights and influence over the development of a project, allowing holders to participate in decision-making processes. Examples include Uniswap (UNI), Aave (AAVE), and Compound (COMP).  
Stablecoins: These tokens maintain a relatively stable value pegged to a reserve of assets or fiat currency, providing a less volatile option for transactions and trading. Examples include USD Coin (USDC), Tether (USDT), and MakerDAO's DAI.  
Liquidity tokens: Liquidity providers on DeFi platforms often receive ERC-20 tokens as a representation of their share in a liquidity pool. These tokens can be staked or traded, and they enable users to earn rewards for providing liquidity. Examples include Uniswap LP tokens and Curve LP tokens.  
Rewards tokens: Some platforms issue ERC-20 tokens as incentives for users to participate in their ecosystem, such as staking, lending, or providing liquidity. These tokens can be earned as passive income or used to access additional platform features. Examples include Synthetix (SNX) and SushiSwap (SUSHI).  
Each of these use cases demonstrates the adaptability of ERC-20 tokens to serve different needs within the blockchain ecosystem.  
Conclusion  
By providing a consistent framework for fungible tokens and adhering to the formal EIP-20 specification, ERC-20 has enabled the development of countless projects and applications that have revolutionized how value is exchanged and managed on Ethereum. Analyzing the anatomy of an ERC-20 token and reviewing its formal specification reveal the versatility and importance of this token standard.  
See Also  
Introduction to Ethereum Improvement Proposals (EIPs)  
EIP-20: ERC-20 Token Standard  
ERC-20 Token Standard  
ON THIS PAGE  
Introduction  
ERC-20 Specification  
Anatomy of an ERC-20 Token  
Benefits of ERC-20 Standardization  
Applications  
Conclusion  
See Also  
See Also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-20-token/erc-20-testing-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
Analyzing the ERC-20 Standard  
The ERC-20 Token Standard  
OpenZeppelin ERC-20 Implementation  
ERC-20 Testing  
ERC-20 Implementation  
ERC-20 Tokens Exercise  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
ERC-20 Testing  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-20-token/erc-20-token-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
Analyzing the ERC-20 Standard  
The ERC-20 Token Standard  
OpenZeppelin ERC-20 Implementation  
ERC-20 Testing  
ERC-20 Implementation  
ERC-20 Tokens Exercise  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
ERC-20 Implementation  
The ERC-20 is a standard that allows for the development of fungible tokens and helps sites and apps, such as exchanges, know how to find and display information about these tokens. You can leverage existing implementations, such as the one by OpenZeppelin to develop your own tokens.  
Objectives  
By the end of this lesson you should be able to:  
Describe OpenZepplin  
Import the OpenZepplin ERC-20 implementation  
Describe the difference between the ERC-20 standard and OpenZeppelin's ERC20.sol  
Build and deploy an ERC-20 compliant token  
Setting Up the Contract  
Create a new Solidity file, add the license and pragma, and import the ERC-20 implementation linked above.  
Add a contract called MyERC20Token that inherits from ERC20.  
// SPDX-License-Identifier: MIT  
  
pragma solidity ^0.8.17;  
  
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol";  
  
contract MyERC20Token is ERC20 {  
  
}  
Adding a Constructor  
Review the constructor on line 53 of the OpenZeppelin implementation. It requires strings for the name and symbol you wish to use for your token. They're using a slightly different naming convention by putting the \_ after the name of the parameters. Like any other function, you can pass variables of any name as long as they're the right type, so feel free to continue adding the \_ in front in your contract's constructor:  
constructor(string memory \_name, string memory \_symbol) ERC20(\_name, \_symbol) {  
  
}  
CAUTION  
There is neither a governing body nor built-in programmatic rules preventing you, or anyone else, from using the same name and symbol as an already in-use token. Scammers often take advantage of this fact, and even well-meaning developers can cause confusion by not being careful here.  
That's it. You're done! Deploy and test, and you should see all of the functionality called for by the standard and provided by the OpenZeppelin implementation.  
Do some testing. You'll see that the totalSupply and all balances are zero.  
By default, the decimal for the token will be 18, which is the most common choice. Remember, there aren't decimal types yet, so 1.0 ETH is really a uint holding 1 \* 10\*\*18, or 1000000000000000000.  
ERC-20 Further Testing  
Line 251 of the OpenZeppelin implementation contains a \_mint function, but it's internal. As a result, you'll need to figure out a minting mechanism and add it via your own contract.  
Minting in the Constructor  
One method of using the \_mint function is to create an initial supply of tokens in the constructor. Add a call to \_mint that awards 1 full token to the contract creator. Remember, the decimal is 18. Minting literally 1 is creating a tiny speck of dust.  
constructor(string memory \_name, string memory \_symbol) ERC20(\_name, \_symbol) {  
 \_mint(msg.sender, 1 \* 10\*\*18);  
}  
Redeploy. Without you needing to do anything, you should find that the totalSupply is now 1000000000000000000, as is the balanceOf the deploying address.  
You can also use this to mint to other users. Go ahead and add the second and third accounts:  
Reveal code  
  
  
  
  
  
  
Switch back to the first account and redeploy. Test to confirm that each account has the appropriate amount of tokens.  
Testing the Transfer Function  
Try using the transfer function to move tokens around.  
What happens if you try to burn a token by sending it to the zero address? Give it a try!  
You'll get an error, because protecting from burning is built into the \_transfer function.  
transact to MyERC20Token.transfer pending ...  
transact to MyERC20Token.transfer errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
Reason provided by the contract: "ERC20: transfer to the zero address".  
Debug the transaction to get more information.  
Testing the Transfer From Function  
You might have noticed that there's another function called transferFrom. What's that for? Check the documentation in the contract to find out!  
This function works with the allowance function to give the owner of one wallet permission to spend up to a specified amount of tokens owned by another. Exchanges can make use of this to allow a user to post tokens for sale at a given price without needing to take possession of them.  
ERC-20 Final Thoughts  
The world is still figuring out how to handle all of the new possibilities tokens provide. Old laws are being applied in new ways, and new laws are being written. Different jurisdictions are doing this in unique and sometimes conflicting ways.  
You should consult with a lawyer in your jurisdiction before releasing your own tokens.  
Conclusion  
In this lesson, you've learned how easy it is to create an ERC-20 compliant token by using the OpenZeppelin implementation. You've reviewed at least one method to mint an initial supply of tokens, and that it's up to you to figure out the best way to create your tokens and follow all relevant laws and regulations.  
ON THIS PAGE  
Setting Up the Contract  
ERC-20 Further Testing  
Minting in the Constructor  
ERC-20 Final Thoughts  
Minting in the Constructor  
Testing the Transfer Function  
Testing the Transfer From Function  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-20-token/openzeppelin-erc-20-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
Analyzing the ERC-20 Standard  
The ERC-20 Token Standard  
OpenZeppelin ERC-20 Implementation  
ERC-20 Testing  
ERC-20 Implementation  
ERC-20 Tokens Exercise  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
OpenZeppelin ERC-20 Implementation  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-721-token/erc-721-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
ERC-721 Token Standard  
The ERC-721 Token Standard  
ERC-721 Token On Opensea  
OpenZeppelin ERC-721 Implementation  
Implementing an ERC-721  
ERC-721 Token  
ERC-721 Tokens Exercise  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
ERC-721 Tokens Exercise  
Create a contract that adheres to the following specifications.  
Contract  
Create a contract called HaikuNFT. Add the following to the contract:  
A struct called Haiku to store the address of the author and line1, line2, and line3  
A public array to store these haikus  
A public mapping to relate sharedHaikus from the address of the wallet shared with, to the id of the Haiku NFT shared  
A public counter to use as the id and to track and share the total number of Haikus minted  
If 10 Haikus have been minted, the counter should be at 11, to serve as the next id  
Do NOT assign an id of 0 to a haiku  
Other variables as necessary to complete the task  
Add the following functions.  
Constructor  
As appropriate.  
Mint Haiku  
Add an external function called mintHaiku that takes in the three lines of the poem. This function should mint an NFT for the minter and save their Haiku.  
Haikus must be unique! If any line in the Haiku has been used as any line of a previous Haiku, revert with HaikuNotUnique().  
You don't have to count syllables, but it would be neat if you did! (No promises on whether or not we counted the same as you did)  
Share Haiku  
Add a public function called shareHaiku that allows the owner of a Haiku NFT to share that Haiku with the designated address they are sending it \_to. Doing so should add it to that address's entry in sharedHaikus.  
If the sender isn't the owner of the Haiku, instead revert with an error of NotYourHaiku. Include the id of the Haiku in the error.  
DANGER  
Remember, everything on the blockchain is public. This sharing functionality can be expanded for features similar to allowing an app user to display the selected shared haiku on their profile.  
It does nothing to prevent anyone and everyone from seeing or copy/pasting the haiku!  
Get Your Shared Haikus  
Add a public function called getMySharedHaikus. When called, it should return an array containing all of the haikus shared with the caller.  
If there are no haikus shared with the caller's wallet, it should revert with a custom error of NoHaikusShared, with no arguments.  
CAUTION  
The contract specification contains actions that can only be performed once by a given address. As a result, the unit tests for a passing contract will only be successful the first time you test.  
You may need to submit a fresh deployment to pass  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Contract  
Constructor  
Mint Haiku  
Share Haiku  
Get Your Shared Haikus  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-721-token/erc-721-on-opensea-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
ERC-721 Token Standard  
The ERC-721 Token Standard  
ERC-721 Token On Opensea  
OpenZeppelin ERC-721 Implementation  
Implementing an ERC-721  
ERC-721 Token  
ERC-721 Tokens Exercise  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
ERC-721 Token On Opensea  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-721-token/erc-721-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
ERC-721 Token Standard  
The ERC-721 Token Standard  
ERC-721 Token On Opensea  
OpenZeppelin ERC-721 Implementation  
Implementing an ERC-721  
ERC-721 Token  
ERC-721 Tokens Exercise  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
ERC-721 Token  
Punks, Apes, and birds of all kinds. You've heard about them, seen them, and may even be lucky enough to own a famous NFT. Or maybe you've just bought into a random collection and aren't sure what to do with your NFT. NFTs aren't really pictures, or anything else specific. They're a method of proving ownership of a digital asset. Anyone can right-click on a picture of a monkey and set it as their profile picture, but only the owner can use it with apps that utilize web3 ownership.  
The ERC-721 token standard is the underlying technical specification that not only makes digital ownership possible, it provides a standardized way for marketplaces, galleries, and other sites to know how to interact with these digital items.  
Objectives  
By the end of this lesson you should be able to:  
Analyze the anatomy of an ERC-721 token  
Compare and contrast the technical specifications of ERC-20 and ERC-721  
Review the formal specification for ERC-721  
Build and deploy an ERC-721 compliant token  
Use an ERC-721 token to control ownership of another data structure  
Implementing the OpenZeppelin ERC-721 Token  
JPGs may be all the rage right now but in the future, the selfie you post on social media, a text message you send to your mother, and the +4 battleaxe you wield in your favorite MMO might all be NFTs.  
Import and Setup  
Start by opening the OpenZeppelin ERC-721 in Github. Copy the link and use it to import the ERC-721 contract. Create your own contract, called MyERC721, that inherits from ERC721Token. Add a constructor that initializes the \_name and \_symbol.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
Minting NFTs  
The minting function that is provided by OpenZeppelin, \_safeMint, is internal. To use it to let your customers mint NFTs, you'll need to implement a function in your contract that calls the one in the imported contract.  
Before you can do that, you need a way to supply the two parameters needed for \_safeMint:  
address to - the owner of the new NFT  
uint256 tokenId - the ID number for the new NFT  
The owner is easy, you can simply use msg.sender to grant ownership to the wallet doing the minting.  
ID is slightly more challenging. A common practice is to simply assign the total number of NFTs, including the one being minted, as the tokenId. Doing so is straightforward, makes it easier to find all of the NFTs within a collection, and helps lean in to the common community perception that lower-number NFTs are better, just like other limited-edition collectibles.  
CAUTION  
Obfuscating certain information, such as customer IDs, is often considered a best practice. Doing so might make it harder for an attacker who has circumvented other security functions from getting access to more data. If 134 is a valid customer\_id, it is likely that 135 is too. The same can't be said for bfcb51bd-c04f-42d5-8116-3def754e8c32.  
This practice is not as useful on the blockchain, because all information is public.  
To implement ID generation, simply add a uint called counter to storage and initialize it as 1, either at declaration or in the constructor.  
Now, you can add a function called redeemNFT that calls safeMint using the msg.sender and counter, and then increments the counter:  
Reveal code  
  
  
  
  
  
DANGER  
As a programmer, you've probably gone through great pains to internalize the idea of zero-indexing. Arrays start at 0. The pixel in the top-left corner of your screen is located at 0, 0.  
As a result, you need to be very careful when working with Solidity because there isn't the concept of undefined, and "deleted" values return to their default value, which is 0 for numbers.  
To prevent security risks, you'll need to make sure that you never give an ID or array index of 0 to anything. Otherwise, attempting to delete a value, such as a struct member called authorizedSellerID might give the wallet address stored at index 0 access to that resource.  
Deploy and test. Be sure to:  
Mint several NFTs  
Transfer an NFT from one Remix account to another  
Try to transfer an NFT to 0x0000000000000000000000000000000000000000  
ERC-721 URIs  
The ERC-721 standard includes the option to define a URI associated with each NFT. These are intended to point to a json file following the ERC721 Metadata JSON Schema  
{  
 "title": "Asset Metadata",  
 "type": "object",  
 "properties": {  
 "name": {  
 "type": "string",  
 "description": "Identifies the asset to which this NFT represents"  
 },  
 "description": {  
 "type": "string",  
 "description": "Describes the asset to which this NFT represents"  
 },  
 "image": {  
 "type": "string",  
 "description": "A URI pointing to a resource with mime type image/\* representing the asset to which this NFT represents. Consider making any images at a width between 320 and 1080 pixels and aspect ratio between 1.91:1 and 4:5 inclusive."  
 }  
 }  
}  
Note that they don't have to. In the OpenZeppelin implementation, the function that returns the \_baseURI is virtual and must be overridden by an inheriting contract.  
// OpenZeppelin ERC-721  
/\*\*  
 \* @dev Base URI for computing {tokenURI}. If set, the resulting URI for each  
 \* token will be the concatenation of the `baseURI` and the `tokenId`. Empty  
 \* by default, can be overridden in child contracts.  
 \*/  
function \_baseURI() internal view virtual returns (string memory) {  
 return "";  
}  
The owner of the contract can therefore choose what the value is and when, how, or if it is changeable. For example, the [Bored Ape Yacht Club] contract has a function allowing the owner to set or change the \_baseURI, changing where the metadata is stored, and potentially what is in it.  
// From boredapeyachtclub.sol  
function setBaseURI(string memory baseURI) public onlyOwner {  
 \_setBaseURI(baseURI);  
}  
The metadata for BAYC is stored on IPFS, but some projects even use centralized, web2 storage options!  
NFT Switcheroo  
Doodles is another NFT collection that uses IPFS to store metadata. Let's modify our contract to swap metadata back and forth from one collection to the other.  
Start by saving the IPFS metadata bases as constants, at the contract level. Add an enum to enable selection between these two choices, and an instance of that enum.  
Reveal code  
  
  
  
  
  
  
Finally, add an override of \_baseURI that returns the appropriate selection based on which collection is active, and a function to swap the URI.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Deploy, mint some NFTs, and call tokenURI to find the information for token number 1. You should get:  
https://ipfs.io/ipfs/QmeSjSinHpPnmXmspMjwiXyN6zS4E9zccariGR3jxcaWtq/1  
This links to the metadata json file for the first Bored Ape:  
{  
 "image": "ipfs://QmPbxeGcXhYQQNgsC6a36dDyYUcHgMLnGKnF8pVFmGsvqi",  
 "attributes": [  
 {  
 "trait\_type": "Mouth",  
 "value": "Grin"  
 },  
 {  
 "trait\_type": "Clothes",  
 "value": "Vietnam Jacket"  
 },  
 {  
 "trait\_type": "Background",  
 "value": "Orange"  
 },  
 {  
 "trait\_type": "Eyes",  
 "value": "Blue Beams"  
 },  
 {  
 "trait\_type": "Fur",  
 "value": "Robot"  
 }  
 ]  
}  
IPFS links don't work natively directly in the browser, but you can see the image here:  
https://ipfs.io/ipfs/QmPbxeGcXhYQQNgsC6a36dDyYUcHgMLnGKnF8pVFmGsvqi/  
Now, call your switchURI function and then call tokenURI again for token 1.  
Now, you'll get a new link for metadata:  
https://ipfs.io/ipfs/QmPMc4tcBsMqLRuCQtPmPe84bpSjrC3Ky7t3JWuHXYB4aS/1  
Which contains the metadata for Doodle 1 instead of BAYC 1:  
{  
 "image": "ipfs://QmTDxnzcvj2p3xBrKcGv1wxoyhAn2yzCQnZZ9LmFjReuH9",  
 "name": "Doodle #1",  
 "description": "A community-driven collectibles project featuring art by Burnt Toast. Doodles come in a joyful range of colors, traits and sizes with a collection size of 10,000. Each Doodle allows its owner to vote for experiences and activations paid for by the Doodles Community Treasury. Burnt Toast is the working alias for Scott Martin, a Canadian\u2013based illustrator, designer, animator and muralist.",  
 "attributes": [  
 {  
 "trait\_type": "face",  
 "value": "holographic beard"  
 },  
 {  
 "trait\_type": "hair",  
 "value": "white bucket cap"  
 },  
 {  
 "trait\_type": "body",  
 "value": "purple sweater with satchel"  
 },  
 {  
 "trait\_type": "background",  
 "value": "grey"  
 },  
 {  
 "trait\_type": "head",  
 "value": "gradient 2"  
 }  
 ]  
}  
Your robot ape is now a person with a rainbow beard!  
https://ipfs.io/ipfs/QmTDxnzcvj2p3xBrKcGv1wxoyhAn2yzCQnZZ9LmFjReuH9  
Conclusion  
In this lesson, you've learned how to use OpenZeppelin's ERC-721 implementation to create your own NFT contract. You've also learned how NFT metadata is stored, and that it is not necessarily immutable.  
ON THIS PAGE  
Implementing the OpenZeppelin ERC-721 Token  
ERC-721 URIs  
NFT Switcheroo  
Minting NFTs  
Conclusion  
NFT Switcheroo  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-721-token/erc-721-standard  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
ERC-721 Token Standard  
The ERC-721 Token Standard  
ERC-721 Token On Opensea  
OpenZeppelin ERC-721 Implementation  
Implementing an ERC-721  
ERC-721 Token  
ERC-721 Tokens Exercise  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
The ERC-721 Token Standard  
In this article, we'll delve into the ERC-721 token standard, exploring its technical specs, applications, and how it differs from the ERC-20 standard.  
Objectives:  
By the end of this lesson you should be able to:  
Analyze the anatomy of an ERC-721 token  
Compare and contrast the technical specifications of ERC-20 and ERC-721  
Review the formal specification for ERC-721  
Introduction  
The development of the Ethereum ecosystem has been marked by key milestones, two of which are the inception of the ERC-20 and ERC-721 token standards. While ERC-20 provided a foundational framework for fungible tokens, ERC-721 established a flexible and adaptable infrastructure for non-fungible tokens (NFTs).  
The ERC-721 token standard is pivotal in the Ethereum ecosystem for creating and managing unique digital assets. With its consistent rules and functions, it has greatly enhanced the user experience, solidifying its position as the go-to standard for non-fungible tokens. ERC-721 has been instrumental in expanding the digital collectibles market and spurring the development of new applications and services.  
ERC-721 Specification  
EIP-721 (Ethereum Improvement Proposal 721) is the formal specification for ERC-721, defining the requirements for creating compliant non-fungible tokens on Ethereum. EIP-721 prescribes mandatory functions and events that a token must implement to achieve ERC-721 compliance. Adherence to EIP-721 ensures compatibility of unique tokens with existing Ethereum applications and services, simplifying integration.  
Anatomy of an ERC-721 Token  
An ERC-721 token comprises a smart contract implementing the standardized interface, which includes six primary functions:  
balanceOf(address) Returns the number of tokens held by a specific address.  
ownerOf(uint256): Provides the owner of a specified token.  
safeTransferFrom(address, address, uint256): Transfers a specific token's ownership from one address to another.  
transferFrom(address, address, uint256): Allows a third party to transfer tokens on the token owner's behalf, given the owner's approval.  
approve(address, uint256): Enables the token owner to permit a third party to transfer a specific token on their behalf.  
getApproved(uint256): Shows the approved address for a specific token.  
These functions ensure each ERC-721 token has a unique identifier and can be owned and transferred individually.  
ERC-721 Vs ERC-20  
The ERC-721 and ERC-20 token standards share a common goal of providing a set of standards for tokens on the Ethereum network but diverge in terms of functionality and use cases.  
ERC-20 tokens are fungible, meaning each token is identical to every other token; they are interchangeable like currency. On the other hand, ERC-721 tokens are non-fungible, meaning each token is unique and not interchangeable with any other token. This uniqueness is made possible through the ownerOf() and getApproved() functions, which provide information about the ownership of each unique token.  
The ERC-20 standard has primarily found use in creating cryptocurrencies for apps, governance tokens, utility tokens, stablecoins, and more. The ERC-721 standard, conversely, has been adopted largely for creating unique digital assets like collectibles, digital art, and tokenized virtual real estate, among other applications.  
Benefits of ERC-721 Standardization  
Standardizing non-fungible tokens via the ERC-721 token standard presents substantial benefits to developers and users in the Ethereum ecosystem. Developers have access to a standardized set of functions, leading to less code ambiguity, fewer errors, and a streamlined development process. This uniformity also ensures smooth integration with existing apps and platforms on Ethereum.  
For users, the ERC-721 standard offers an intuitive, consistent interface for interacting with a wide array of unique tokens. Regardless of the token's specific use or design, users can reliably check their ownership of tokens, transfer tokens to other addresses, and approve transactions. This consistency enhances usability across the Ethereum platform, from digital art marketplaces to tokenized real estate and gaming applications.  
Applications  
ERC-721 tokens find wide-ranging applications in various categories:  
Digital Art: Artists can create unique digital artworks as ERC-721 tokens. These tokens can be sold or traded on platforms like OpenSea, Rarible, and Coinbase NFT. Examples include work by the digital artist Beeple.  
Gaming: Game assets such as characters, items, and land can be tokenized as ERC-721 tokens, providing players with true ownership of their in-game assets. Examples include Axie Infinity and Decentraland.  
Collectibles: ERC-721 tokens can represent unique collectible items in a digital space. Examples include NBA Top Shot moments and CryptoPunks.  
Virtual Real Estate: Virtual real estate can be tokenized as ERC-721 tokens, providing proof of ownership and facilitating trade on virtual platforms. Examples include parcels of land in Cryptovoxels and Decentraland.  
Conclusion  
ERC-721, with its consistent framework for non-fungible tokens, has revolutionized the unique digital asset space on Ethereum. This standard, when contrasted with ERC-20, highlights Ethereum's capacity for both fungible and unique asset types. Adhering to the EIP-721 specification, ERC-721 tokens have significantly influenced the Ethereum-based digital economy. From digital art to gaming, these tokens underscore their importance and role as catalysts in the burgeoning NFT revolution.  
See Also  
EIP-721: ERC-721 Token Standard  
ERC-721 Token Standard  
ON THIS PAGE  
Introduction  
ERC-721 Specification  
Anatomy of an ERC-721 Token  
ERC-721 Vs ERC-20  
Benefits of ERC-721 Standardization  
Applications  
Conclusion  
See Also  
See Also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-721-token/erc-721-standard-video  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
ERC-721 Token Standard  
The ERC-721 Token Standard  
ERC-721 Token On Opensea  
OpenZeppelin ERC-721 Implementation  
Implementing an ERC-721  
ERC-721 Token  
ERC-721 Tokens Exercise  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
ERC-721 Token Standard  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-721-token/implementing-an-erc-721-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
ERC-721 Token Standard  
The ERC-721 Token Standard  
ERC-721 Token On Opensea  
OpenZeppelin ERC-721 Implementation  
Implementing an ERC-721  
ERC-721 Token  
ERC-721 Tokens Exercise  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Implementing an ERC-721  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/erc-721-token/openzeppelin-erc-721-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
ERC-721 Token Standard  
The ERC-721 Token Standard  
ERC-721 Token On Opensea  
OpenZeppelin ERC-721 Implementation  
Implementing an ERC-721  
ERC-721 Token  
ERC-721 Tokens Exercise  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
OpenZeppelin ERC-721 Implementation  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/error-triage/  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
Error Triage  
Error Triage  
Error Triage Exercise  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Error Triage  
Debugging is a part of every language, platform, and framework. The EVM is a unique and relatively constrained computer, so you'll encounter some types of errors that may be unfamiliar. In this article, we'll explore some of the more common ones and share methods for resolving them.  
Objectives  
By the end of this lesson you should be able to:  
Debug common solidity errors including transaction reverted, out of gas, stack overflow, value overflow/underflow, index out of range, etc.  
Compiler Errors  
Compiler errors are manifold but almost always very easy to debug, since the error message usually tells you what is wrong and how to fix it.  
Type Errors  
You will get a compiler error if you try to assign a literal to the wrong type.  
// Bad code example, do not use  
function compilerTypeError() public pure returns (uint) {  
 uint myNumber = "One";  
 return myNumber;  
}  
from solidity:  
TypeError: Type literal\_string "One" is not implicitly convertible to expected type uint256.  
 --> contracts/ErrorTriage.sol:8:9:  
 |  
8 | uint myNumber = "One";  
 | ^^^^^^^^^^^^^^^^^^^^^  
Fix by correcting the type or value, as appropriate for your needs:  
Reveal code  
  
  
  
  
  
Conversion Errors  
Conversion errors occur when you attempt to implicitly convert one type to another. Solidity only allows this under very narrow circumstances where there is no possibility of ambiguous interpretation of the data.  
// Bad code example, do not use  
function compilerConversionError() public pure returns (uint) {  
 int8 first = 1;  
  
 return first;  
}  
from solidity:  
TypeError: Return argument type int8 is not implicitly convertible to expected type (type of first return variable) uint256.  
 --> contracts/ErrorTriage.sol:15:16:  
 |  
15 | return first;  
 | ^^^^^  
Fix by explicitly casting as necessary:  
Reveal code  
  
  
  
  
  
  
TIP  
You'll commonly need to use multiple conversions to bridge from one type to another.  
Operator Errors  
You cannot use operators between types as flexibly as you may be used to.  
// Bad code example, do not use  
function compilerOperatorError() public pure returns (uint) {  
 int8 first = 1;  
 uint256 second = 2;  
  
 uint sum = first + second;  
  
 return sum;  
}  
Operator errors are often paired with a type error.  
from solidity:  
TypeError: Operator + not compatible with types int8 and uint256.  
 --> contracts/ErrorTriage.sol:22:20:  
 |  
22 | uint sum = first + second;  
 | ^^^^^^^^^^^^^^  
  
from solidity:  
TypeError: Type int8 is not implicitly convertible to expected type uint256.  
 --> contracts/ErrorTriage.sol:22:9:  
 |  
22 | uint sum = first + second;  
 | ^^^^^^^^^^^^^^^^^^^^^^^^^  
Resolve by explicitly converting to the final type:  
Reveal code  
  
  
  
  
  
  
  
  
  
Stack Depth Limit  
The EVM stack has 1024 slots, but only the top 16 slots are accessible. As a result, you can only have fewer than 16 variables in scope at one time.  
CAUTION  
Other items can also use up these slots. You are not guaranteed 15 slots, it can be lower.  
// Bad code example, do not use  
function stackDepthLimit() public pure returns (uint) {  
 uint first = 1;  
 uint second = 2;  
 uint third = 3;  
 uint fourth = 4;  
 uint fifth = 5;  
 uint sixth = 6;  
 uint seventh = 7;  
 uint eighth = 8;  
 uint ninth = 9;  
 uint tenth = 10;  
 uint eleventh = 11;  
 uint twelfth = 12;  
 uint thirteenth = 13;  
 uint fourteenth = 14;  
 uint fifteenth = 15;  
 uint sixteenth = 16;  
  
 return first +  
 second +  
 third +  
 fourth +  
 fifth +  
 sixth +  
 seventh +  
 eighth +  
 ninth +  
 tenth +  
 eleventh +  
 twelfth +  
 thirteenth +  
 fourteenth +  
 fifteenth +  
 sixteenth;  
 }  
from solidity:  
CompilerError: Stack too deep. Try compiling with --via-ir (cli) or the equivalent viaIR: true (standard JSON) while enabling the optimizer. Otherwise, try removing local variables.  
 --> contracts/ErrorTriage.sol:92:17:  
 |  
92 | eighth +  
 | ^^^^^^  
Resolve this error by breaking up large functions and separating operations into different levels of scope.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Logical Errors  
Logical errors occur when your code is syntactically correct, but still results in a data state that is a violation of the rules of the language.  
A panic occurs when your code tries to do an illegal operation. These return with a very basic error code, which Remix unfortunately hides. However, it makes up for that annoyance by providing a very powerful debugger.  
CAUTION  
The Remix VM doesn't behave exactly the same as true onchain operations, so note that these errors will not behave exactly the same if triggered while testing with Hardhat, or called from a front end.  
For each of these examples, copy them into Remix to explore with the debugger on your own.  
Array Index Out-of-Bounds  
A panic will be triggered if you try to access an array at an invalid index.  
// Bad code example, do not use  
function badGetLastValue() public pure returns (uint) {  
 uint[4] memory arr = [uint(1), 2, 3, 4];  
  
 return arr[arr.length];  
}  
Running this function will result in the following error in the console:  
call to ErrorTriage.badGetLastValue errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
Note: The called function should be payable if you send value and the value you send should be less than your current balance.  
Debug the transaction to get more information.  
Click the Debug button to open the debugger.  
The debugger contains panels with information about variables in storage, memory, what's on the stack, and so on. You can also add breakpoints to lines of code to further help with debugging.  
One of the most useful features is the link near the top instructing you to "Click here to jump where the call reverted."  
Click that link and the debugger will jump to the point of failure, and highlight the code that caused the panic. Neat!  
You can find the specific error here, but it's difficult.  
Look in the Memory panel. The first item at 0x0 has a hash starting with 0x4e487b71. This code indicates a panic.  
The second item, at 0x20 has the error code of 32 hidden in it, which is for array out-of-bounds.  
It's sometimes better to just review the code first to see if the error is obvious.  
function badGetLastValueFixed() public pure returns (uint) {  
 uint[4] memory arr = [uint(1), 2, 3, 4];  
  
 return arr[arr.length-1];  
}  
Out of Gas  
The default settings for Remix make it difficult to trigger an out of gas error because the VM will often crash first. For this example, go to the Deploy & Run Transactions tab and reduce the gas limit to 300000.  
If you write code that can have an ambiguous execution time, it becomes very difficult to accurately estimate gas limits.  
In this example, each loop has a 1 in 1000 chance of ending.  
DANGER  
block.timestamp can be manipulated. DO NOT use this as a source of randomness if any value can be derived from one outcome over another!  
// Bad code example, do not use  
function badRandomLoop() public view returns (uint) {  
 uint seed = 0;  
 // DO NOT USE THIS METHOD FOR RANDOM NUMBERS!!! IT IS EASILY EXPLOITABLE!!!  
 while(uint(keccak256(abi.encodePacked(block.timestamp, seed))) % 1000 != 0) {  
 seed++;  
 // ...do something  
 }  
  
 return seed;  
}  
Run this function a few times. Often, it will work just fine. Other times, an error appears:  
call to ErrorTriage.badLoop errored: VM error: out of gas.  
  
out of gas  
 The transaction ran out of gas. Please increase the Gas Limit.  
  
Debug the transaction to get more information.  
The error message here is a bit misleading. You do not usually want to fix this by increasing the gas limit. If you're getting a gas error because the transaction didn't estimate for enough gas, it's better to refactor for better predictability.  
function badRandomLoopFixed() public view returns (uint) {  
 // DO NOT USE THIS METHOD FOR RANDOM NUMBERS!!! IT IS EASILY EXPLOITABLE!!!  
 uint times = uint(keccak256(abi.encodePacked(block.timestamp))) % 1000;  
  
 for(uint i = 0; i <= times; i++) {  
 // ...do something  
 }  
  
 return times;  
}  
Overflow or Underflow  
The uint type will panic in the event of an overflow or underflow.  
function badSubstraction() public pure returns (uint) {  
 uint first = 1;  
 uint second = 2;  
 return first - second;  
}  
As before, you can see the panic code and panic type in memory.  
In this case, the error type is 11, for overflow/underflow outside of an unchecked block.  
Fix by changing your code to handle the expected range of values.  
Reveal code  
  
  
  
  
  
  
Divide by Zero  
Divide by zero errors also trigger a panic, with a code of 12.  
function badDivision() public pure returns (uint) {  
 uint first = 1;  
 uint second = 0;  
 return first / second;  
}  
Don't divide by zero.  
Conclusion  
In this lesson, you reviewed the causes of and solutions for a number of compiler errors and logical errors that you may encounter.  
ON THIS PAGE  
Compiler Errors  
Logical Errors  
Array Index Out-of-Bounds  
Out of Gas  
Overflow or Underflow  
Divide by Zero  
Conclusion  
Array Index Out-of-Bounds  
Out of Gas  
Overflow or Underflow  
Divide by Zero  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/error-triage/error-triage-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
Error Triage  
Error Triage  
Error Triage Exercise  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Error Triage Exercise  
Copy the starter code into a new file in Remix.  
Debug the existing functions in the provided contract.  
// SPDX-License-Identifier: MIT  
  
pragma solidity ^0.8.17;  
  
contract ErrorTriageExercise {  
 /\*\*  
 \* Finds the difference between each uint with it's neighbor (a to b, b to c, etc.)  
 \* and returns a uint array with the absolute integer difference of each pairing.  
 \*/  
 function diffWithNeighbor(  
 uint \_a,  
 uint \_b,  
 uint \_c,  
 uint \_d  
 ) public pure returns (uint[] memory) {  
 uint[] memory results = new uint[](3);  
  
 results[0] = \_a - \_b;  
 results[1] = \_b - \_c;  
 results[2] = \_c - \_d;  
  
 return results;  
 }  
  
 /\*\*  
 \* Changes the \_base by the value of \_modifier. Base is always >= 1000. Modifiers can be  
 \* between positive and negative 100;  
 \*/  
 function applyModifier(  
 uint \_base,  
 int \_modifier  
 ) public pure returns (uint) {  
 return \_base + \_modifier;  
 }  
  
 /\*\*  
 \* Pop the last element from the supplied array, and return the popped  
 \* value (unlike the built-in function)  
 \*/  
 uint[] arr;  
  
 function popWithReturn() public returns (uint) {  
 uint index = arr.length - 1;  
 delete arr[index];  
 return arr[index];  
 }  
  
 // The utility functions below are working as expected  
 function addToArr(uint \_num) public {  
 arr.push(\_num);  
 }  
  
 function getArr() public view returns (uint[] memory) {  
 return arr;  
 }  
  
 function resetArr() public {  
 delete arr;  
 }  
}  
  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/error-triage/error-triage-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
Error Triage  
Error Triage  
Error Triage Exercise  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Error Triage  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/ethereum-virtual-machine/evm-diagram  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
EVM Diagram  
In this article, we'll examine the inner workings of the EVM, its components, and its role within the Ethereum network.  
Objectives  
By the end of this lesson you should be able to:  
Diagram the EVM  
What is the EVM?  
The Ethereum Virtual Machine (EVM) is the core engine of Ethereum. It is a Turing-complete, sandboxed virtual machine designed to execute smart contracts on the network. The term "sandboxed" means that the EVM operates in an isolated environment, ensuring that each smart contract's execution does not interfere with others or the underlying blockchain. As we've learned, the EVM's Turing-complete nature allows developers to write complex programs that can perform any computationally feasible task.  
The EVM employs a sophisticated resource management system using gas to regulate computation costs and prevent network abuse. It also supports a rich ecosystem of apps by providing a versatile set of opcodes for smart contract logic, and fostering interoperability with various programming languages, tools, and technologies. This adaptability has made the EVM a fundamental component in the advancement and growth of the Ethereum network.  
EVM Components  
The EVM has several key components that enable it to process and manage smart contracts. Let's define them:  
World State: Represents the entire Ethereum network, including all accounts and their associated storage.  
Accounts: Entities that interact with the Ethereum network, including Externally Owned Accounts (EOAs) and Contract Accounts.  
Storage: A key-value store associated with each contract account, containing the contract's state and data.  
Gas: A mechanism for measuring the cost of executing operations in the EVM, which protects the network from spam and abuse.  
Opcodes: Low-level instructions that the EVM executes during smart contract processing.  
Execution Stack: A last-in, first-out (LIFO) data structure for temporarily storing values during opcode execution.  
Memory: A runtime memory used by smart contracts during execution.  
Program Counter: A register that keeps track of the position of the next opcode to be executed.  
Logs: Events emitted by smart contracts during execution, which can be used by external systems for monitoring or reacting to specific events.  
EVM Execution Model  
In simple terms, when a transaction is submitted to the network, the EVM first verifies its validity. If the transaction is deemed valid, the EVM establishes an execution context that incorporates the current state of the network and processes the smart contract's bytecode using opcodes. As the EVM runs the smart contract, it modifies the blockchain's world state and consumes gas accordingly. However, if the transaction is found to be invalid, it will be dismissed by the network without further processing. Throughout the smart contract's execution, logs are generated that provide insights into the contract's performance and any emitted events. These logs can be utilized by external systems for monitoring purposes or to respond to specific events.  
Gas and Opcode Execution  
While we have already delved into the concept of gas in a previous lesson, it is worth reiterating its critical role within the EVM and as a fundamental component of Ethereum. Gas functions as a metric for quantifying the computational effort needed to carry out operations in the EVM. Every opcode in a smart contract carries a specific gas cost, which reflects the computational resources necessary for its execution.  
Opcodes are the low-level instructions executed by the EVM. They represent elementary operations that allow the EVM to process and manage smart contracts.  
During execution, the EVM reads opcodes from the smart contract, and depending on the opcode, it may update the world state, consume gas, or revert the state if an error occurs. Some common opcodes include:  
ADD: Adds two values from the stack.  
SUB: Subtracts two values from the stack.  
MSTORE: Stores a value in memory.  
SSTORE: Stores a value in contract storage.  
CALL: Calls another contract or sends ether.  
Stack and Memory  
The EVM stack and memory are critical components of the EVM architecture, as they enable smart contracts to manage temporary data during opcode execution. The stack is a last-in, first-out (LIFO) data structure that is used for temporarily storing values during opcode execution. It is managed by the EVM and is separate from the contract's storage. The stack supports two primary operations: push and pop.  
The push operation adds a value to the top of the stack, while the pop operation removes the top value from the stack. These operations are used to manage temporary data during opcode execution. For example, an opcode that performs an addition operation might push the two operands onto the stack, perform the addition, and then pop the result off the top of the stack.  
During contract execution, memory serves as a collection of bytes, organized in an array, for the purpose of temporarily storing data. It can be read from and written to by opcodes. Memory is often used to store temporary data during opcode execution, such as when working with dynamically sized data like strings or arrays that are being manipulated or computed within the smart contract before being stored in the contract's storage. When a smart contract needs to store temporary data during opcode execution, it can use the memory to store that data.  
EVM Architecture and Execution Context  
To fully grasp the EVM architecture and its components, it's important to see how they all come together in a cohesive manner. The following diagram provides an in-depth visualization of the EVM architecture, showcasing the interactions between key elements such as transactions, gas, opcodes, and the world state. With this diagram, you can see how each component plays a vital role in the seamless execution of smart contracts on the Ethereum network.  
Image Source: Mastering Ethereum by Andreas M. Antonopoulos and Gavin Wood, licensed under CC BY-SA 4.0  
Conclusion  
The EVM plays a vital role within the Ethereum network. By examining the EVM's key components as well as its architecture and execution model, we've gained insight into the engine of Ethereum and how it enables the smooth execution of smart contracts on the platform.  
See Also  
The Ethereum Virtual Machine (Mastering Ethereum)  
Ethereum Virtual Machine (Ethereum docs)  
ON THIS PAGE  
What is the EVM?  
EVM Components  
EVM Execution Model  
Gas and Opcode Execution  
Stack and Memory  
EVM Architecture and Execution Context  
Conclusion  
See Also  
See Also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/etherscan/etherscan-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Etherscan  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Etherscan  
In this article, you'll learn about Etherscan, a blockchain explorer to inspect the Blockchain state and activity.  
Objectives  
By the end of this lesson, you should be able to:  
List some of the features of Etherscan  
Read data from the Bored Apes Yacht Club contract on Etherscan  
Write data to a contract using Etherscan.  
Overview  
Etherscan is a popular Blockchain explorer that works for several different networks. In it, you can explore the state and activity of a particular network.  
You can explore:  
Blocks  
Transactions  
Smart contracts  
And more!  
For instance, the following shows the details of a Block:  
Where you see information such as:  
Timestamp  
Transactions  
Block height  
And other details  
There are many variations of Etherscan for different networks. For example:  
Base  
Base Sepolia  
Sepolia Etherscan  
Reading data from smart contracts using Etherscan  
One of the things you can do with Etherscan is interact with already-deployed contracts.  
For example, if you want to read information from a famous contract such as BAYC, you can simply go to Etherscan and explore the contract:  
You are able to see information such as:  
The ETH balance it holds  
The contract creator  
The transaction when it was created  
Latest transactions  
And the verified contract  
In the Contract tab, you can see the full source code of BAYC:  
For a developer, verifying contracts is important since it gives transparency to your users. However, there are some risks because this means that bad actors can see the full source code and can try to exploit it.  
In order to read the state of the BAYC, you can go to the main menu and select the option Read Contract:  
After you select that option, you are able to see all of the read functions of the contract.  
You can also query who is the owner of the BAYC with id 150:  
Writing data to smart contracts using Etherscan  
In a similar fashion, you can read data from smart contracts using Etherscan. It is also possible to write data.  
To write data, go to the Write Contract tab:  
From there, connect your wallet by clicking the Connect with web3 button.  
After you connect, the following UI appears:  
You can then call the functions you wish to write to.  
INFO  
Be aware that you may need to have real Ethereum in case you want to write to a contract in Ethereum mainnet. Also, any logic that the smart contract defines will be respected. This means that if you try to write to a contract that verifies certain conditions during the transaction (e.g., a function where only the owner of the contract can write information), then you won't be able to execute the transaction if you are not the owner.  
Conclusion  
In this lesson, you've learned some of the main features of Etherscan, including interacting with already-deployed contracts such as BAYC in order to read and write data.  
See also  
ON THIS PAGE  
Overview  
Reading data from smart contracts using Etherscan  
Writing data to smart contracts using Etherscan  
Conclusion  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/etherscan/etherscan-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Etherscan  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Etherscan  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/events/hardhat-events-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Events  
In this article, you'll learn how events work in Solidity by reviewing some practical examples and common use cases of events.  
CAUTION  
This tutorial has been moved as part of a reorganization! It assumes you are using Hardhat. Everything in this lesson will work with minor adjustments if you are working in Foundry or Remix.  
Objectives  
By the end of this lesson, you should be able to:  
Write and trigger an event  
List common uses of events  
Understand events vs. smart contract storage  
Overview  
Understanding how Solidity events work is important in the world of smart contract development. Events provide a powerful way to create event-driven applications on the blockchain. They allow you to notify external parties, such as off-chain applications, user interfaces, and any entity that wants to listen for events of a particular contract.  
In this tutorial, you'll learn how to declare, trigger, and utilize events, gaining the knowledge necessary to enhance the functionality and user experience of your decentralized applications.  
What are events?  
From the official solidity documentation, events are:  
...an abstraction on top of the EVM’s logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.  
...when you call them, they cause the arguments to be stored in the transaction’s log – a special data structure in the blockchain. These logs are associated with the address of the contract that emitted them, are incorporated into the blockchain, and stay there as long as a block is accessible (forever as of now, but this might change in the future).  
In other words, events are an abstraction that allow you to store a transaction's log information in the blockchain.  
Your first solidity event  
Start by creating a first event in the Lock.sol contract that's included by default in Hardhat.  
The event is called Created and includes the address of the creator and the amount that was sent during the creation of the smart contract. Then, emit the event in the constructor:  
emit Created(msg.sender, msg.value);  
The contract is:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Lock {  
 uint public unlockTime;  
 address payable public owner;  
  
 event Created(address owner, uint amount);  
  
 constructor(uint \_unlockTime) payable {  
 require(  
 block.timestamp < \_unlockTime,  
 "Unlock time should be in the future"  
 );  
  
 unlockTime = \_unlockTime;  
 owner = payable(msg.sender);  
  
 emit Created(msg.sender, msg.value);  
 }  
}  
Events can be defined at the file level or as inheritable members of contracts (including interfaces). You can also define the event in an interface as:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
interface ILock {  
 event Created(address owner, uint amount);  
}  
  
contract Lock is ILock {  
 uint public unlockTime;  
 address payable public owner;  
  
 constructor(uint \_unlockTime) payable {  
 require(  
 block.timestamp < \_unlockTime,  
 "Unlock time should be in the future"  
 );  
  
 unlockTime = \_unlockTime;  
 owner = payable(msg.sender);  
  
 emit Created(msg.sender, msg.value);  
 }  
}  
You can test the event by simplifying the original test file with the following code:  
import {  
 time,  
} from "@nomicfoundation/hardhat-toolbox/network-helpers";  
import { ethers } from "hardhat";  
  
describe("Lock tests", function () {  
 describe("Deployment", function () {  
 it("Should set the right unlockTime", async function () {  
 const ONE\_YEAR\_IN\_SECS = 365 \* 24 \* 60 \* 60;  
 const ONE\_GWEI = 1\_000\_000\_000;  
  
 const lockedAmount = ONE\_GWEI;  
 const unlockTime = (await time.latest()) + ONE\_YEAR\_IN\_SECS;  
  
 // Contracts are deployed using the first signer/account by default  
 const [owner] = await ethers.getSigners();  
  
 // But we do it explicit by using the owner signer  
 const LockFactory = await ethers.getContractFactory("Lock", owner);  
 const lock = await LockFactory.deploy(unlockTime, { value: lockedAmount });  
  
 const hash = await lock.deploymentTransaction()?.hash  
 const receipt = await ethers.provider.getTransactionReceipt(hash as string)  
  
 console.log("Sender Address", owner.address)  
 console.log("Receipt.logs", receipt?.logs)  
  
 const defaultDecoder = ethers.AbiCoder.defaultAbiCoder()  
 const decodedData = defaultDecoder.decode(['address', 'uint256'], receipt?.logs[0].data as string)  
 console.log("decodedData", decodedData)  
 });  
 });  
});  
Notice that the previous code is logging the sender address and the logs coming from the transaction receipt. You are also decoding the receipts.logs[0].data field that contains the information emitted by the event but not in a human-readable way, since it is encoded. For that reason, you can use AbiCoder to decode the raw data.  
By running npx harhdat test, you should be able to see the following:  
 Lock tests  
 Deployment  
Sender Address 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266  
Receipt.logs [  
 Log {  
 provider: HardhatEthersProvider {  
 \_hardhatProvider: [LazyInitializationProviderAdapter],  
 \_networkName: 'hardhat',  
 \_blockListeners: [],  
 \_transactionHashListeners: Map(0) {},  
 \_eventListeners: []  
 },  
 transactionHash: '0xad4ff104036f23096ea5ed165bff1c3e1bc0f53e375080f84bce4cc108c28cee',  
 blockHash: '0xb2117cfd2aa8493a451670acb0ce14228b06d17bf545cd7efad6791aeac83c05',  
 blockNumber: 1,  
 removed: undefined,  
 address: '0x5FbDB2315678afecb367f032d93F642f64180aa3',  
 data: '0x000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb92266000000000000000000000000000000000000000000000000000000003b9aca00',  
 topics: [  
 '0x0ce3610e89a4bb9ec9359763f99110ed52a4abaea0b62028a1637e242ca2768b'  
 ],  
 index: 0,  
 transactionIndex: 0  
 }  
]  
decodedData Result(2) [ '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266', 1000000000n ]  
 ✔ Should set the right unlockTime (1008ms)  
Notice the value f39fd6e51aad88f6f4ce6ab8827279cfffb92266 is encoded in the property data, which is the address of the sender.  
Event topics  
Another important feature is that events can be indexed by adding the indexed attribute to the event declaration.  
For example, if you modify the interface with:  
interface ILock {  
 event Created(address indexed owner, uint amount);  
}  
Then, if you run npx hardhat test again, an error may occur because the decoding assumes that the data field contains an address and a uint256. But by adding the indexed attribute, you are instructing that the events will be added to a special data structure known as "topics". Topics have some limitations, since the maximum indexed attributes can be up to three parameters and a topic can only hold a single word (32 bytes).  
You then need to modify the decoding line in the test file with the following:  
const decodedData = defaultDecoder.decode(['uint256'], receipt?.logs[0].data as string)  
Then, you should be able to see the receipt as:  
 Lock tests  
 Deployment  
Sender Address 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266  
Receipt.logs [  
 Log {  
 provider: HardhatEthersProvider {  
 \_hardhatProvider: [LazyInitializationProviderAdapter],  
 \_networkName: 'hardhat',  
 \_blockListeners: [],  
 \_transactionHashListeners: Map(0) {},  
 \_eventListeners: []  
 },  
 transactionHash: '0x0fd52fd72bca26879474d3e512fb812489111a6654473fd288c6e8ec0432e09d',  
 blockHash: '0x138f74df5637315099d31aedf5bf643cf95c2bb7ae923c21fcd7f0075cb55324',  
 blockNumber: 1,  
 removed: undefined,  
 address: '0x5FbDB2315678afecb367f032d93F642f64180aa3',  
 data: '0x000000000000000000000000000000000000000000000000000000003b9aca00',  
 topics: [  
 '0x0ce3610e89a4bb9ec9359763f99110ed52a4abaea0b62028a1637e242ca2768b',  
 '0x000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb92266'  
 ],  
 index: 0,  
 transactionIndex: 0  
 }  
]  
decodedData Result(1) [ 1000000000n ]  
 ✔ Should set the right unlockTime (994ms)  
Notice the topics property, which now contains the address of the sender: f39fd6e51aad88f6f4ce6ab8827279cfffb92266.  
Common uses of events  
Solidity events have several common use cases, which are described in the following sections.  
User notifications  
Events can be used to notify users or external systems about certain contract actions.  
Logging  
Events are primarily used to log significant changes within the contract, providing a transparent and verifiable history of what has occurred.  
Historical state reconstruction  
Events can be valuable for recreating the historical state of a contract. By capturing and analyzing emitted event logs, you can reconstruct past states, offering a transparent and auditable history of the contract's actions and changes.  
Debugging and monitoring  
Events are essential for debugging and monitoring contract behavior, as they provide a way to observe what's happening on the blockchain.  
The ability to use events to recreate historical states provides an important auditing and transparency feature, allowing users and external parties to verify the contract's history and actions. While not a common use, it's a powerful capability that can be particularly useful in certain contexts.  
Events vs. smart contract storage  
Although it is possible to rely on events to fully recreate the state of a particular contract, there are a few other options to consider.  
Existing services such as The Graph allow you to index and create GraphQL endpoints for your smart contracts and generate entities based on custom logic. However, you must pay for that service since you are adding an intermediate layer to your application. This has the following benefits, such as:  
the ability to simply query one particular endpoint to get all the information you need  
your users will pay less gas costs due to the minimization of storage usage in your contract  
But storing all of the information within the smart contract and relying fully on it to access data can create more complexity, since not all of the data is directly query-able. The benefits of this approach include:  
your application requires only the smart contract address to access all of the required data  
there are fewer dependencies involved, which makes this approach more crypto native in the sense that everything is in the blockchain (but, storing all the data in the blockchain will cause higher gas costs)  
As a smart contract developer, you must evaluate which options work best for you.  
Conclusion  
In this lesson, you've learned the basics of Solidity events and their importance in Ethereum smart contract development. You now understand how to declare and trigger events, a few of their common use cases, and the difference between events and smart contract storage.  
Now that you have a solid grasp of events and their versatile applications, you can leverage them to build more sophisticated and interactive smart contracts that meet your specific needs, all while being mindful of the cost considerations.  
See also  
ON THIS PAGE  
Overview  
What are events?  
Your first solidity event  
Event topics  
Common uses of events  
Events vs. smart contract storage  
User notifications  
Logging  
Historical state reconstruction  
Debugging and monitoring  
Conclusion  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/exercise-contracts  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Exercise Contracts  
Many of the sections in Base Learn contain an exercise to test your knowledge on the material you have just completed. We tell you what to do, but not how to do it. You have to apply your knowledge and demonstrate the new abilities you have earned.  
Upon success, you'll be granted a non-transferable, or soulbound, NFT as a memento of your learning. You can track your progress on the progress page.  
Below is a list of the exercises, with links to view their code. The unit tests are written in a bespoke framework in Solidity, but the patterns should be recognizable to most engineers.  
Exercise Code  
Deploying to a Testnet 0x075eB9Dc52177Aa3492E1D26f0fDE3d729625d2F  
Control Structures 0xF4D953A3976F392aA5509612DEfF395983f22a84  
Storage 0x567452C6638c0D2D9778C20a3D59749FDCaa7aB3  
Arrays 0x5B0F80cA6f5bD60Cc3b64F0377f336B2B2A56CdF  
Mappings 0xD32E3ACe3272e2037003Ca54CA7E5676f9b8D06C  
Structs 0x9eB1Fa4cD9bd29ca2C8e72217a642811c1F6176d  
Inheritance 0xF90dA05e77a33Fe6D64bc2Df84e7dd0069A2111C  
Imports 0x8dD188Ec36084D59948F90213AFCd04429E33c0c  
Errors 0xC1BD0d9A8863f2318001BC5024c7f5F58a2236F7  
The "new" Keyword 0x4f21e69d0CDE8C21cF82a6b37Dda5444716AFA46  
Minimal Tokens 0x10Ce928030E136EcC74d4a4416Db9b533e3c694D  
ERC-20 Tokens 0x4F333c49B820013e5E6Fe86634DC4Da88039CE50  
ERC-721 Tokens 0x15534ED3d1dBA55148695B2Ba4164F147E47a10c  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/frontend-setup/building-an-onchain-app  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Wallet Connectors  
Building an Onchain App  
Connecting to the Blockchain  
Reading and Displaying Data  
Writing to Contracts  
Exercise Contracts  
Building an Onchain App  
While it's convenient and fast to start from a template, the template may not fit your needs. Whether you prefer a different stack, or have already started building the traditional web components of your app, it's common to need to manually add onchain libraries to get your app working.  
In this guide, you'll build the beginnings of an app similar to the one created by the RainbowKit quick start, but you'll do it piece by piece. You can follow along, and swap out any of our library choices with the ones you prefer.  
Objectives  
By the end of this guide you should be able to:  
Identify the role of a wallet aggregator in an onchain app  
Debate the pros and cons of using a template  
Add a wallet connection to a standard template app  
Creating the Traditional App  
Start by running the Next.js script to create a Next.js app:  
npx create-next-app@latest --use-yarn  
This script will accept ., if you want to add the project to the root of a folder you've already created. Otherwise, name your project. Select each option in the generation script as you see fit. We recommend the following selections:  
Use Typescript?: Yes  
Use ESLint?: Yes  
Use Tailwind?: Your preference  
Use src/ directory?: Yes  
Use App Router?: Yes  
Customize the default import alias?: No  
INFO  
The default Next.js script installs Tailwind. RainbowKit's does not.  
Run your app with yarn dev to make sure it generated correctly.  
Manually Installing RainbowKit, Wagmi, and Viem  
The quick start guide for RainbowKit also contains step-by-step instructions for manual install. You'll be following an adjusted version here. Most of the setup is actually for configuring wagmi, which sits on top of viem and makes it much easier to write React that interacts with the blockchain.  
Start by installing the dependencies:  
npm install @rainbow-me/rainbowkit wagmi viem@2.x @tanstack/react-query  
INFO  
Onchain libraries and packages tend to require very current versions of Node. If you're not already using it, you may want to install nvm.  
Adding Imports, Connectors, Config  
In Next.js with the app router, the root of your app is found in app/layout.tsx, if you followed the recommended setup options. As you want the blockchain provider context to be available for the entire app, you'll add it here.  
You'll need to set up your providers in a second file, so that you can add 'use client'; to the top. Doing so forces this code to be run client side, which is necessary since your server won't have access to your users' wallet information.  
CAUTION  
You must configure these wrappers in a separate file. It will not work if you try to add them and 'use client'; directly in layout.tsx!  
Add a new file in the app folder called providers.tsx.  
Imports  
As discussed above, add 'use client'; to the top of the file.  
Continue with the imports:  
import '@rainbow-me/rainbowkit/styles.css';  
import { useState, type ReactNode } from 'react';  
import { getDefaultConfig, RainbowKitProvider } from '@rainbow-me/rainbowkit';  
import { WagmiProvider } from 'wagmi';  
import { base, baseSepolia } from 'wagmi/chains';  
import { QueryClientProvider, QueryClient } from '@tanstack/react-query';  
CAUTION  
If you're adapting this guide to a different set of libraries or platforms, you may need to import styles.css differently. You'll know this is the case if you get ugly text at the bottom of the page instead of a nice modal when you click the connect button.  
Config  
Now, you need to configure the chains, wallet connectors, and providers for your app. You'll use getDefaultConfig for now, to get started. See our guide on Connecting to the Blockchain for more information on blockchain providers.  
INFO  
To take advantage of a more advanced set of options with OnchainKit, see our tutorial on how to Use the Coinbase Smart Wallet and EOAs with OnchainKit. If you just want to customize the list of wallets in RainbowKit, see our tutorial for Coinbase Smart Wallet with RainbowKit.  
You'll need a projectId from Wallet Connect Cloud, which you can get for free on their site. Make sure to insert it in the appropriate place.  
DANGER  
Remember, everything on the frontend is public! Be sure to configure the allowlist for your WalletConnect id!  
const config = getDefaultConfig({  
 appName: 'Cool Onchain App',  
 projectId: 'YOUR\_PROJECT\_ID',  
 chains: [base, baseSepolia],  
 ssr: true, // If your dApp uses server side rendering (SSR)  
});  
Returning the Context Providers  
TanStack Query is now a required dependency for wagmi, and you need to add it as a React context provider. The short version is that it helps with state management. Read the docs for the long version!  
Add an exported function for the providers. This sets up the QueryClient and returns props.children wrapped in all of your providers.  
export function Providers(props: { children: ReactNode }) {  
 const [queryClient] = useState(() => new QueryClient());  
  
 return (  
 <WagmiProvider config={config}>  
 <QueryClientProvider client={queryClient}>  
 <RainbowKitProvider>{props.children}</RainbowKitProvider>  
 </QueryClientProvider>  
 </WagmiProvider>  
 );  
}  
Using Your new Providers  
Open layout.tsx. Import your Providers, being careful if you use auto-import as there are many other things with similar names in the list. Wrap the children in your return with the new Providers.  
return (  
 <html lang="en">  
 <body className={inter.className}>  
 <Providers>{children}</Providers>  
 </body>  
 </html>  
);  
Adding the Connect Button  
You're now ready to add your connect button. You can do this anywhere in your app, thanks to the RainbowKitProvider. Common practice would be to place the button in your app's header. Since the Next.js template doesn't have one, you can just add it to the top of the automatically generated page, rather than spending time implementing React components.  
Open up page.tsx, and import the ConnectButton:  
import { ConnectButton } from '@rainbow-me/rainbowkit';  
Then, simply add the ConnectButton component at the top of the first <div>:  
// This function has been simplified to save space.  
export default function Home() {  
 return (  
 <main className="flex min-h-screen flex-col items-center justify-between p-24">  
 <div className="z-10 w-full max-w-5xl items-center justify-between font-mono text-sm lg:flex">  
 <ConnectButton />  
  
 {/\* Other Code...\*/}  
 </p>  
 </div>  
 </main>  
 );  
}  
Run your app with yarn dev, and you should be able to use the RainbowKit connect button to connect with your wallet and switch between networks.  
You use the Connect Button props to modify its properties, or you can customize the connect button extensively. Some users dislike having the connect button display their token balance. Try disabling it with:  
<ConnectButton showBalance={false} />  
Conclusion  
In this guide, you've learned how to assemble your onchain app from several pieces. You can use this knowledge to integrate a wallet connection with an existing site, or adjust the stack to meet your preferences. Finally, you've learned how to insert and customize the connect button.  
If you're looking to quickly bootstrap a simple app, you can always use a script, such as the RainbowKit quick start. If you're looking for a robust start for a consumer application, check out OnchainKit!  
ON THIS PAGE  
Creating the Traditional App  
Adding Imports, Connectors, Config  
Imports  
Using Your new Providers  
Imports  
Config  
Returning the Context Providers  
Adding the Connect Button  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/frontend-setup/overview  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Connecting to the Blockchain  
Reading and Displaying Data  
Writing to Contracts  
Exercise Contracts  
Overview  
Welcome! The course you are about to begin will rapidly introduce you to frontend web development for onchain apps and enable you to write websites that can call your smart contract functions in a similar way to how traditional sites interact with APIs.  
Prerequisites  
Before these lessons, you should:  
Be comfortable with traditional frontend development using React, ideally with NextJS  
Possess a general understanding of the EVM and smart contracts  
Objectives  
By the end of this course, you should be able to:  
Frontend Setup  
Identify the role of a wallet aggregator in an onchain app  
Debate the pros and cons of using a template  
Scaffold a new onchain app with RainbowKit  
Add a wallet connection to a standard template app  
Connecting to the Blockchain  
Compare and contrast public providers vs. vendor providers vs. wallet providers  
Select the appropriate provider for several use cases  
Set up a provider in wagmi and use it to connect a wallet  
Protect API keys that will be exposed to the front end  
Reading and Displaying Data  
Implement the useAccount hook to show the user's address, connection state, network, and balance  
Implement an isMounted hook to prevent hydration errors  
Implement wagmi's useReadContract hook to fetch data from a smart contract  
Convert data fetched from a smart contract to information displayed to the user  
Identify the caveats of reading data from automatically-generated getters  
Enable the watch feature of useReadContract to automatically fetch updates from the blockchain  
Describe the costs of using the watch feature, and methods to reduce those costs  
Configure arguments to be passed with a call to a pure or view smart contract function  
Call an instance of useReadContract on demand  
Utilize isLoading and isFetching to improve user experience  
Writing to Contracts  
Implement wagmi's useWriteContract hook to send transactions to a smart contract  
Configure the options in useWriteContract  
Display the execution, success, or failure of a function with button state changes, and data display  
Implement Wagmi's usePrepareContractWrite and useWriteContract to send transactions to a smart contract  
Configure the options in useSimulateContract and useWriteContract  
Call a smart contract function on-demand using the write function from useWriteContract, with arguments and a value  
ON THIS PAGE  
Objectives  
Objectives  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/frontend-setup/wallet-connectors  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Wallet Connectors  
Building an Onchain App  
Connecting to the Blockchain  
Reading and Displaying Data  
Writing to Contracts  
Exercise Contracts  
Wallet Connectors  
One of the most intimidating tasks when building an onchain app is making that initial connection between your users' wallets, and your app. Initial research often surfaces a bewildering number of wallets, each with their own SDKs, and own methods to manage the connection. Luckily, you don't actually need to manage all of this on your own. There are a number of wallet connector libraries specialized in creating a smooth and beautiful user experience to facilitate this connection.  
To further add to the confusion and difficulty, Smart wallets are growing in popularity. These advanced wallets allow users to create and manage wallets with passkeys, and support, or will soon support, a growing array of features including session keys, account recovery, and more!  
RainbowKit, the aggregator you'll use for this lesson, works with the Coinbase Smart Wallet out of the box, but you'll need to do a little bit of extra configuration to support users of both traditional wallets and smart wallets.  
Objectives  
By the end of this guide you should be able to:  
Identify the role of a wallet aggregator in an onchain app  
Debate the pros and cons of using a template  
Scaffold a new onchain app with RainbowKit  
Support users of EOAs and the Coinbase Smart Wallet with the same app  
Connecting to the Blockchain  
One of the many challenging tasks of building a frontend that can interface with your smart contracts is managing the user's connection between your onchain app and their [EOA] wallet. Not only is there an ever-growing suite of different wallets, but users can (and probably should!) use several different addresses within the same wallet app.  
Rainbowkit is one of several options that makes this a little bit easier by serving as an aggregator of wallets, and handling some of the details of connecting them. Alternatives include ConnectKit, and Dynamic, which are both excellent choices as well.  
Each of these include customizable UI/UX components for inviting the user to connect, displaying connection status, and selecting which wallet they wish to use.  
Using the Quick Start  
If you're just trying to get up and running as quickly as possible, you can use RainbowKit's quick start script to scaffold an app from their template, with a single command. If you're using Yarn:  
yarn create @rainbow-me/rainbowkit  
INFO  
The script doesn't accept . as a project name, so you'll want to run this script in your src directory, or wherever you keep your projects. It will create a folder with the same name as your project, and install the project files inside.  
Once it's done, simply run the app with:  
yarn run dev  
Using the script is fast, but it does mean less choice. In this case, it builds the app on top of Next.js, which is great if you want to use it, but not helpful if you prefer to work from a different framework, such as Create React App, or Remix (the React framework, not the Solidity IDE). The script also doesn't help you if you want to add an onchain integration to an existing site.  
INFO  
The Rainbowkit template has been updated to wagmi 2.X, but it does not use the Next.js app router. You'll need to install it manually if you wish to use the latest patterns.  
The Building an Onchain App tutorial will show you how to do this!  
Coinbase Smart Wallet  
If you have the Coinbase Wallet extension, you might be wondering where the smart wallet can be found. By default, the smart wallet will only be invoked if you click the Coinbase Wallet button to log in and you don't have the browser extension. To test, open a private window with extensions disabled and try to log in.  
Selecting Rainbow, MetaMask, or WalletConnect will display a QR code so that the user can log in with their phone. Picking Coinbase Wallet will instead invoke the smart wallet login.  
This flow can be improved upon, as new crypto users won't know that digging for the smart wallet is the best path forward, and existing users who are trying to migrate to the smart wallet don't have that option.  
See our tutorial on how to Use the Coinbase Smart Wallet and EOAs with OnchainKit for more details!  
Conclusion  
In this article, you've learned how libraries such as Rainbowkit, ConnectKit, and Dynamic, aggregate wallets and make it easier for you to connect your app to your users' wallet of choice. You've also learned how you can use a template to quickly create the foundation of your app. Finally, you've learned that the cost of using a template is that it does make some choices for you.  
ON THIS PAGE  
Connecting to the Blockchain  
Conclusion  
Using the Quick Start  
Coinbase Smart Wallet  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-deploy/deployment-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Installing Hardhat Deploy  
Setting up the Deploy Script  
Testing Our Deployment  
Test Network Configuration  
Deployment  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Deployment  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-deploy/hardhat-deploy-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Installing Hardhat Deploy  
Setting up the Deploy Script  
Testing Our Deployment  
Test Network Configuration  
Deployment  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Deploying Smart Contracts  
In this article, you'll learn how to deploy smart contracts to multiple Blockchain networks using Hardhat and Hardhat deploy.  
Objectives  
By the end of this lesson, you should be able to:  
Deploy a smart contract to the Base Sepolia Testnet with hardhat-deploy  
Deploy a smart contract to the Sepolia Testnet with hardhat-deploy  
Use BaseScan to view a deployed smart contract  
Overview  
Hardhat capabilities enable developers to deploy smart contracts easily to any Blockchain by simply creating tasks or scripts. However, due to the Hardhat architecture that enables its extension by creating plugins, you can rely on existing solutions developed by the community.  
Hardhat deploy is a community-developed plugin that enables the deployment of your smart contracts in a simple way.  
Setting up Hardhat deploy  
To install:  
Run npm install -D hardhat-deploy. Then, import hardhat-deploy in hardhat.config.ts:  
import 'hardhat-deploy';  
Create a folder called deploy and inside it create a new file called 001\_deploy\_lock.ts.  
Include the following:  
import { HardhatRuntimeEnvironment } from 'hardhat/types';  
import { DeployFunction } from 'hardhat-deploy/types';  
  
const func: DeployFunction = async function (hre: HardhatRuntimeEnvironment) {  
 // code here  
};  
export default func;  
Modify the tsconfig.json file to look like:  
{  
 "compilerOptions": {  
 "target": "es2020",  
 "module": "commonjs",  
 "esModuleInterop": true,  
 "forceConsistentCasingInFileNames": true,  
 "strict": true,  
 "skipLibCheck": true,  
 "resolveJsonModule": true  
 },  
 "include": ["./hardhat.config.ts", "./scripts", "./deploy", "./test"]  
}  
Before implementing the deploy functionality, configure a deployer account in the hardhat.config.ts file. Hardhat deployment includes a way to name accounts in the config file.  
Run the following, which adds an alias to the account 0 of your environment:  
const config: HardhatUserConfig = {  
 solidity: '0.8.23',  
 namedAccounts: {  
 deployer: 0,  
 },  
};  
Implement the deploy function by including the following in the 001\_deploy\_lock.ts file:  
import { HardhatRuntimeEnvironment } from 'hardhat/types';  
import { DeployFunction } from 'hardhat-deploy/types';  
import { ethers } from 'hardhat';  
  
const func: DeployFunction = async function (hre: HardhatRuntimeEnvironment) {  
 const { deploy } = hre.deployments;  
 // We can now use deployer  
 const { deployer } = await hre.getNamedAccounts();  
  
 // The value we want to lock  
 const VALUE\_LOCKED = hre.ethers.parseEther('0.01');  
  
 // The unlock time after deployment  
 const UNLOCK\_TIME = 10000;  
  
 // We use ethers to get the current time stamp  
 const blockNumber = await ethers.provider.getBlockNumber();  
 const lastBlockTimeStamp = (await ethers.provider.getBlock(blockNumber))?.timestamp as number;  
  
 // We say we want to deploy our Lock contract using the deployer  
 // account and passing the value and arguments.  
 await deploy('Lock', {  
 from: deployer,  
 args: [lastBlockTimeStamp + UNLOCK\_TIME],  
 value: VALUE\_LOCKED.toString(),  
 });  
};  
  
export default func;  
  
// This tag will help us in the next section to trigger this deployment file programmatically  
func.tags = ['DeployAll'];  
Testing your deployment  
The easiest way to test your deployment is by modifying the test.  
Go to Lock.ts and include in the imports the following:  
import { ethers, deployments } from 'hardhat';  
deployments will allow you to execute the deployment files from your test.  
Change the before function to look like the following:  
before(async () => {  
 lastBlockTimeStamp = await time.latest();  
  
 const signers = await ethers.getSigners();  
 ownerSigner = signers[0];  
 otherUserSigner = signers[1];  
  
 await deployments.fixture(['DeployAll']);  
 const lockDeployment = await deployments.get('Lock');  
  
 lockInstance = Lock\_\_factory.connect(lockDeployment.address, ownerSigner);  
});  
Notice how you execute deployments.fixture and pass a tag that matches the one you specified in the deployment file (001\_deploy\_lock.ts).  
The deployment file is then executed and you can then reuse that functionality and simply consume the address of the newly-deployed contract by using:  
const lockDeployment = await deployments.get('Lock');  
Reuse Lock\_\_factory but use the connect function and pass the address of the newly-created contract plus a signer. Then, run npx hardhat test and you should get the same result:  
 Lock  
 ✔ should get the unlockTime value  
 ✔ should have the right ether balance  
 ✔ should have the right owner  
 ✔ shouldn"t allow to withdraw before unlock time (51ms)  
 ✔ shouldn"t allow to withdraw a non owner  
 ✔ should allow to withdraw a owner  
  
 6 passing (2s)  
Deploying to a test network  
Deploying to a real test network involves configuring the network parameters in the hardhat config file. You need to include parameters such as:  
The JSON RPC url  
The account you want to use  
Real test ether or the native Blockchain token for gas costs  
Include the following in the hardhat.config.ts file:  
const config: HardhatUserConfig = {  
 solidity: '0.8.18',  
 namedAccounts: {  
 deployer: 0,  
 },  
 networks: {  
 base\_sepolia: {  
 url: 'https://sepolia.base.org',  
 accounts: {  
 mnemonic: process.env.MNEMONIC ?? '',  
 },  
 },  
 sepolia: {  
 url: `https://eth-sepolia.g.alchemy.com/v2/${process.env.ALCHEMY\_SEPOLIA\_KEY ?? ''}`,  
 accounts: {  
 mnemonic: process.env.MNEMONIC ?? '',  
 },  
 },  
 },  
};  
You've configured 2 networks:  
base\_sepolia  
sepolia  
You also need to create a .env file with the following variables:  
MNEMONIC="<REPLACE WITH YOUR MNEMONIC>"  
ALCHEMY\_SEPOLIA\_KEY=<REPLACE WITH YOUR API KEY>  
In order to ensure the environment variables are loaded, you need to install another package called dotenv:  
npm install -D dotenv  
Then, include the following in the hardhat.config.ts file:  
import dotenv from 'dotenv';  
  
dotenv.config();  
Deploy to base with the following command:  
npx hardhat deploy --network base\_sepolia  
After you run the command, a deployments folder appears with a newly-created deployment for base\_sepolia:  
If you want to deploy to another network, change the network name as follows:  
npx hardhat deploy --network sepolia  
INFO  
Be aware that you must have the correct environment variables for the JSON RPC URLs. For example, for Sepolia use ALCHEMY\_SEPOLIA\_KEY.  
Conclusion  
In this lesson, you've learned how to deploy smart contracts using Hardhat and Hardhat-deploy. You have configured hardhat to easily deploy to multiple networks and you created deployment files to abstract this task.  
See also  
Solidity Docs  
ON THIS PAGE  
Overview  
Setting up Hardhat deploy  
Testing your deployment  
Deploying to a test network  
Conclusion  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-deploy/installing-hardhat-deploy-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Installing Hardhat Deploy  
Setting up the Deploy Script  
Testing Our Deployment  
Test Network Configuration  
Deployment  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Installing Hardhat Deploy  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-deploy/setup-deploy-script-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Installing Hardhat Deploy  
Setting up the Deploy Script  
Testing Our Deployment  
Test Network Configuration  
Deployment  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Setting up the Deploy Script  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-deploy/test-network-configuration-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Installing Hardhat Deploy  
Setting up the Deploy Script  
Testing Our Deployment  
Test Network Configuration  
Deployment  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Test Network Configuration  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-deploy/testing-our-deployment-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Installing Hardhat Deploy  
Setting up the Deploy Script  
Testing Our Deployment  
Test Network Configuration  
Deployment  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Testing Our Deployment  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-forking/  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Forking Mainnet  
Hardhat Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Hardhat Forking  
In this article, you'll learn how to fork smart contracts in Ethereum mainnet using Hardhat.  
Objectives  
By the end of this lesson, you should be able to:  
Use Hardhat Network to create a local fork of mainnet and deploy a contract to it  
Utilize Hardhat forking features to configure the fork for several use cases  
Overview  
Hardhat forking is a powerful feature that allows developers to create a local replica or fork of the Ethereum network or any other EVM-compatible Blockchain. By using this feature, you can develop smart contracts that rely on smart contracts that are already deployed to a particular network.  
You will create a BalanceReader.sol contract that reads the USDC balance of a particular holder.  
In order to achieve that, you need to:  
Create the BalanceReader.sol contract  
Configure Hardhat to support forking  
Create a test for the BalanceReader.sol contract  
Hardhat forking also has other capabilities like:  
hardhat\_impersonateAccount (useful to impersonate an account and others)  
hardhat\_stopImpersonatingAccount  
hardhat\_setNonce  
hardhat\_setBalance  
hardhat\_setCode  
hardhat\_setStorageAt  
Those won't be covered in this guide, however it's recommended to explore them a bit more in the following link:  
https://hardhat.org/hardhat-network/guides/mainnet-forking.html  
Creating the Balance Reader contract  
The BalanceReader contract is created as follows:  
pragma solidity 0.8.9;  
  
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
  
contract BalanceReader {  
 function getERC20BalanceOf(address \_account, address \_tokenAddress)  
 external  
 view  
 returns (uint256)  
 {  
 // we create an instance only using the interface and the address  
 return IERC20(\_tokenAddress).balanceOf(\_account);  
 }  
}  
You simply pass the address of an account and the address of a token, then you get and return the balance.  
You will need to install @openzeppelin by running:  
npm install @openzeppelin/contracts  
Then, check that everything is working correctly by running:  
npx hardhat compile  
You should get:  
Generating typings for: 2 artifacts in dir: typechain-types for target: ethers-v6  
Successfully generated 18 typings!  
Compiled 2 Solidity files successfully  
Configuring Hardhat to support forking  
By default, Hardhat uses a network called hardhat. You must change its default configuration by going to the hardhat.config.ts file and include the following in the network:  
hardhat: {  
 forking: {  
 url: `https://eth-mainnet.g.alchemy.com/v2/${process.env.ALCHEMY\_MAINNET\_KEY ?? ""}`,  
 enabled: true  
 }  
},  
Be aware that you need to have an ALCHEMY\_MAINNET\_KEY in your .env file. You can get one directly from Alchemy.  
Also notice that forking is enabled by specifying enabled: true, however this value can be changed via environment variables.  
Creating a test for the BalanceReader.sol contract  
Create a test file in the test folder called BalanceReader.ts and include the following:  
import { Signer } from 'ethers';  
import { ethers } from 'hardhat';  
  
import { BalanceReader, BalanceReader\_\_factory } from '../typechain-types';  
  
describe('BalanceReader tests', () => {  
 let instance: BalanceReader;  
 let accounts: Signer[];  
  
 // Configure the addresses we can to check balances for  
 const USDC\_MAINNET\_ADDRESS = '0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48'; // https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48  
 const ARBITRUM\_ONE\_GATEWAY = '0xcEe284F754E854890e311e3280b767F80797180d';  
 const USDC\_DECIMALS = 6;  
  
 it('gets arbitrum gateway balance', async () => {  
 // We get signers as in a normal test  
 accounts = await ethers.getSigners();  
 const factory = new BalanceReader\_\_factory(accounts[0]);  
  
 // We deploy the contract to our local test environment  
 instance = await factory.deploy();  
  
 // Our contract will be able to check the balances of the mainnet deployed contracts and address  
 const balance = await instance.getERC20BalanceOf(ARBITRUM\_ONE\_GATEWAY, USDC\_MAINNET\_ADDRESS);  
 const balanceAsString = ethers.formatUnits(balance, USDC\_DECIMALS);  
  
 console.log(  
 'The USDC Balance of Arbitrum Gateway is $',  
 Number(balanceAsString).toLocaleString(),  
 );  
 });  
});  
In this example, the USDC address is used and since USDC is an ERC-20 token, you can explore the token holders of that particular token directly in Etherscan:  
Or, visit https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48#balances, where you can see, at the time or writing, Arbitrum ONE Gateway is the top token holder.  
Then, run the following command:  
npx hardhat test ./test/BalanceReader.ts  
You should get:  
BalanceReader tests  
The USDC Balance of Arbitrum Gateway is $ 1,116,923,836.506  
 ✔ gets arbitrum gateway balance (4345ms)  
  
 1 passing (4s)  
Conclusion  
In this lesson, you've learned how to use hardhat forking capabilities to test smart contracts. You learned how contracts can interact with already-deployed contracts in an easy way.  
See also  
ON THIS PAGE  
Overview  
Creating the Balance Reader contract  
Configuring Hardhat to support forking  
Creating a test for the BalanceReader.sol contract  
Conclusion  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-forking/mainnet-forking-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Forking Mainnet  
Hardhat Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Forking Mainnet  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-setup-overview/creating-a-project-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Overview  
Creating a Project  
Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Creating a Project  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-setup-overview/hardhat-overview-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Overview  
Creating a Project  
Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Overview  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-setup-overview/hardhat-setup-overview-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Overview  
Creating a Project  
Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Setup and Overview  
In this article, you'll learn about Hardhat: a development framework to create, test, and deploy smart contracts to Ethereum and other supported EVM chains.  
Objectives  
By the end of this lesson, you should be able to:  
Install and create a new Hardhat project with Typescript support  
Describe the organization and folder structure of a Hardhat project  
List the use and properties of hardhat.config.ts  
Overview  
Hardhat is a development environment that allows you to develop and test Solidity on your local machine. It includes debugging and unit testing tools, and has an ecosystem of third-party-developed plugins that ease development and deployment.  
Among other things, these plugins can help you deploy contracts, see the size of your compiled byte-code, and even see unit test coverage.  
Installing Hardhat and creating a new project  
As a pre-requisite to start developing smart contracts with Hardhat, Node.js must be installed.  
You can then simply type npx hardhat init, which provides a set of options to bootstrap a Hardhat project:  
888 888 888 888 888  
888 888 888 888 888  
888 888 888 888 888  
8888888888 8888b. 888d888 .d88888 88888b. 8888b. 888888  
888 888 "88b 888P" d88" 888 888 "88b "88b 888  
888 888 .d888888 888 888 888 888 888 .d888888 888  
888 888 888 888 888 Y88b 888 888 888 888 888 Y88b.  
888 888 "Y888888 888 "Y88888 888 888 "Y888888 "Y888  
  
👷 Welcome to Hardhat v2.11.2 👷‍  
  
? What do you want to do? …  
❯ Create a JavaScript project  
 Create a TypeScript project  
 Create an empty hardhat.config.js  
 Quit  
You are encouraged to select Create a TypeScript project, since it provides you with some benefits such as static typing that can reduce the number of errors during development.  
You can then enter 'yes' for the remaining options, which include installing the @nomicfoundation/hardhat-toolbox package that contains some of the most used Hardhat plugins.  
✔ What do you want to do? · Create a TypeScript project  
✔ Hardhat project root: · {any location}  
✔ Do you want to add a .gitignore? (Y/n) · y  
✔ Do you want to install this sample project's dependencies with npm (hardhat @nomicfoundation/hardhat-toolbox)? (Y/n) · y  
Anatomy of a Hardhat project  
After you complete the previous step, the folder structure looks like the following:  
contracts # contracts will go here  
hardhat.config.ts # configuration file for hardhat  
node\_modules # node.js package folder  
package-lock.json # node.js package lock file  
package.json # node.js package file  
scripts # place the scripts here  
test # place the tests here  
tsconfig.json # typescript configuration file  
It is also common to save hardhat tasks in a task folder.  
It is important to mention that all these paths are fully configurable in the hardhat.config.ts file. You can specify a different folder for the contracts, such as src.  
Configuration  
You can configure the Hardhat environment in the hardhat.config.ts file.  
Since the project uses Typescript, you have the benefit of using static typing.  
The following is the default configuration:  
import { HardhatUserConfig } from 'hardhat/config';  
import '@nomicfoundation/hardhat-toolbox';  
  
const config: HardhatUserConfig = {  
 solidity: '0.8.17',  
};  
  
export default config;  
You can configure aspects such as:  
default network  
networks  
solidity  
paths  
mocha  
For example:  
import { HardhatUserConfig } from 'hardhat/config';  
import '@nomicfoundation/hardhat-toolbox';  
  
const config: HardhatUserConfig = {  
 defaultNetwork: 'base',  
 networks: {  
 base\_sepolia: {  
 url: 'https://sepolia.base.org',  
 accounts: ['<private key 1>'],  
 },  
 sepolia: {  
 url: 'https://sepolia.infura.io/v3/<key>',  
 accounts: ['<private key 1>', '<private key 2>'],  
 },  
 },  
 solidity: {  
 version: '0.8.17',  
 settings: {  
 optimizer: {  
 enabled: true,  
 runs: 200,  
 },  
 },  
 },  
 paths: {  
 sources: './contracts',  
 tests: './test',  
 cache: './cache',  
 artifacts: './artifacts',  
 },  
};  
  
export default config;  
Compiling smart contracts  
At this point, you should have a Hardhat project up and running to start developing smart contracts. You may notice Hardhat includes a sample contract called Lock.sol.  
To run your first command, enter npx hardhat compile, which compiles the smart contracts and generates the correct artifacts that includes the bytecode and ABI.  
After running the npx hardhat compile command, you should see a new folder named artifacts. This folder contains each contract name as a folder and a {ContractName}.json file.  
ON THIS PAGE  
Overview  
Installing Hardhat and creating a new project  
Compiling smart contracts  
Anatomy of a Hardhat project  
Configuration  
Compiling smart contracts  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-testing/contract-abi-and-testing-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Testing Overview  
Writing Tests  
Contract ABIs and Testing  
Testing with Hardhat and Typechain  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Contract ABIs and Testing  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-testing/hardhat-testing-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Testing Overview  
Writing Tests  
Contract ABIs and Testing  
Testing with Hardhat and Typechain  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Testing with Hardhat and Typechain  
In this article, you'll learn how to test smart contracts with Hardhat and Typechain.  
Objectives  
By the end of this lesson, you should be able to:  
Set up TypeChain to enable testing  
Write unit tests for smart contracts using Mocha, Chai, and the Hardhat Toolkit  
Set up multiple signers and call smart contract functions with different signers  
Overview  
Testing is an important aspect of software development and developing smart contracts is no different. In fact, you need to be more careful because smart contracts usually manage money and live in an adversarial environment, where anyone can see the code and interact with your smart contract. This means you can expect bad actors to try to exploit your smart contracts.  
Setup Typechain  
In the previous guide, you created a new project using the init command that by default installs @nomicfoundation/hardhat-toolbox. This package already contains Typechain, which is a plugin that generates static types for your smart contracts. This means you can interact with your contracts and get immediate feedback about the parameters received by a particular function and the functions of a smart contract.  
The best way to see its true potential is to start writing tests.  
After compiling the hardhat project in the previous lesson, a new folder called typechain-types was created, which Typechain is already installed and running.  
Writing your first unit test with Typechain  
Hardhat includes a sample smart contract named Lock.sol and a sample test inside the test folder named Lock.ts.  
In the following, you reuse this smart contract but rewrite the test using Typechain.  
To remove the body of the Lock.ts file:  
import { expect } from 'chai';  
import { ethers } from 'hardhat';  
  
describe('Lock', function () {});  
Then, import two files from typechain-types, Lock, and Lock\_\_Factory.  
Typechain always creates two files per contract. The first one Lock refers to the type and functions of a particular contract. Lock\_\_Factory is used to deploy the Lock contract or to create instances of a particular contract.  
The Lock.sol contract allows the creator to lock Ether until an unlock time has passed.  
Notice the constructor has a payable keyword:  
constructor(uint \_unlockTime) payable {  
 require(  
 block.timestamp < \_unlockTime,  
 "Unlock time should be in the future"  
 );  
  
 unlockTime = \_unlockTime;  
 owner = payable(msg.sender);  
 }  
This means the contract is expecting to receive an amount of ether.  
Next, test the following:  
The unlock time value  
The value locked during creation  
The owner address  
The withdraw function  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Testing unlockTime  
Next, you include test cases after the before function.  
The first test case should verify that the unlockTime variable is correct.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
Testing Ether balance  
In order to get the balance of your Lock contract, you simply call ethers.provider.getBalance.  
Create a new test case:  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
Then, run npx hardhat test and you should get:  
 Lock  
 ✔ should get the unlockTime value  
 ✔ should have the right ether balance  
  
 2 passing (1s)  
Testing owner  
Similar to the previous test cases, you can verify that the owner is correct.  
Reveal code  
  
  
  
  
  
Then, run npx hardhat test and you should get:  
 Lock  
 ✔ should get the unlockTime value  
 ✔ should have the right ether balance  
 ✔ should have the right owner  
  
 3 passing (1s)  
Testing withdraw  
Testing withdrawal is more complex because you need to assert certain conditions, such as:  
The owner cannot withdraw before the unlock time.  
Only the owner can withdraw.  
The withdraw function works as expected.  
Hardhat allow you to test reverts with a set of custom matchers.  
For example, the following code checks that an attempt to call the function withdraw reverts with a particular message:  
it('shouldn"t allow to withdraw before unlock time', async () => {  
 await expect(lockInstance.withdraw()).to.be.revertedWith("You can't withdraw yet");  
});  
In addition, Hardhat also allows you to manipulate the time of the environment where the tests are executed. You can think of it as a Blockchain that is running before the tests and then the tests are executed against it.  
You can modify the block.timestamp by using the time helper:  
it('shouldn"t allow to withdraw a non owner', async () => {  
 const newLastBlockTimeStamp = await time.latest();  
  
 // We set the next block time stamp using this helper.  
 // We assign a value further in the future.  
 await time.setNextBlockTimestamp(newLastBlockTimeStamp + UNLOCK\_TIME);  
  
 // Then we try to withdraw using other user signer. Notice the .connect function that is useful  
 // to create and instance but have the msg.sender as the new signer.  
 const newInstanceUsingAnotherSigner = lockInstance.connect(otherUserSigner);  
  
 // We attempt to withdraw, but since the sender is not the owner, it will revert.  
 await expect(newInstanceUsingAnotherSigner.withdraw()).to.be.revertedWith("You aren't the owner");  
});  
Finally, test that the owner can withdraw. You can manipulate the time similarly to the previous test case but you won't change the signer and will assert the new balances.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
You can then run npx hardhat test and you should get:  
 Lock  
 ✔ should get the unlockTime value  
 ✔ should have the right ether balance  
 ✔ should have the right owner  
 ✔ shouldn"t allow to withdraw before unlock time (51ms)  
 ✔ shouldn"t allow to withdraw a non owner  
 ✔ should allow to withdraw a owner  
  
 6 passing (2s)  
Conclusion  
In this lesson, you've learned how to test smart contracts using Hardhat and Typechain.  
See also  
Solidity Docs  
ON THIS PAGE  
Overview  
Setup Typechain  
Conclusion  
Writing your first unit test with Typechain  
Testing unlockTime  
Testing Ether balance  
Testing owner  
Testing withdraw  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-testing/testing-overview-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Testing Overview  
Writing Tests  
Contract ABIs and Testing  
Testing with Hardhat and Typechain  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Testing Overview  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-testing/writing-tests-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Testing Overview  
Writing Tests  
Contract ABIs and Testing  
Testing with Hardhat and Typechain  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Writing Tests  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-verify/hardhat-verify-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Verifying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Verifying Smart Contracts  
In this article, you'll learn how to verify smart contracts in Etherscan with hardhat and the hardhat deploy plugin.  
Objectives  
By the end of this lesson, you should be able to:  
Verify a deployed smart contract on Etherscan  
Connect a wallet to a contract in Etherscan  
Use etherscan to interact with your own deployed contract  
Overview  
Verifying smart contracts plays an important role in providing security and certainty to the users of your decentralized applications. By offering full visibility of the source code of your smart contract, you provide confidence and transparency of the intention of the code that is being executed.  
The way smart contracts are verified is by simply uploading the source code and contract address to services such as Etherscan.  
Once the contract is verified, the Etherscan explorer shows a status like the following image:  
Luckily, Hardhat and Hardhat-deploy already contain a built-in capability to do this task easily on your behalf.  
This process involves the following steps:  
Getting an Etherscan key  
Configuring Hardhat  
Verifying  
Getting an Etherscan key  
In order to obtain an Etherscan API key, visit Etherscan and create an account.  
Then, go to https://etherscan.io/myapikey and create an API key by clicking the Add button:  
Bear in mind that different networks have other Blockchain explorers. For example:  
Base  
Sepolia  
You'll need to go to that particular explorer and get the API Key following a similar process as mentioned previously (except for Sepolia Etherscan, where you can use the Etherscan mainnet one instead).  
Configuring Hardhat  
You can configure the Etherscan API Key for each different network. For example, include the following to the hardhat.config.ts file for Base Sepolia:  
base\_sepolia: {  
 url: "https://sepolia.base.org",  
 accounts: {  
 mnemonic: process.env.MNEMONIC ?? ""  
 },  
 verify: {  
 etherscan: {  
 apiUrl: "https://api-sepolia.basescan.org",  
 apiKey: process.env.ETHERSCAN\_API\_KEY  
 }  
 }  
}  
Include in your .env file the following:  
ETHERSCAN\_API\_KEY=<YOUR\_ETHERSCAN\_API\_KEY>  
Verifying  
You verify in base, and to do so, simply run the following command:  
npx hardhat --network base\_sepolia etherscan-verify  
You should receive the following response:  
verifying Lock ...  
waiting for result...  
 => contract Lock is now verified  
You can now go to Basescan and search for your contract address, where you'll see the following:  
Conclusion  
In this lesson, you've learned how to verify smart contracts using Hardhat and Hardhat-deploy. You learned how to configure Hardhat to support multiple networks and verify by using a simple command.  
See also  
ON THIS PAGE  
Overview  
Getting an Etherscan key  
Configuring Hardhat  
Verifying  
Conclusion  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/hardhat-verify/hardhat-verify-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Hardhat Setup and Overview  
Testing with Typescript  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Verifying Smart Contracts  
Verifying Smart Contracts  
Mainnet Forking  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Verifying Smart Contracts  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/help-on-discord  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Get Help on Discord  
Need help?  
Whether you're confused about something weird in Solidity, having trouble getting through the frontend material, or just stuck on an exercise - we're here to help!  
First, join the Base Discord.  
Then, get help from the community, Based Advocates, and the authors of the program in the [Base Learn] channel!  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/imports/imports-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Imports  
Imports  
Imports Exercise  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Imports Exercise  
Create a contract that adheres to the following specifications.  
Contract  
Create a contract called ImportsExercise. It should import a copy of SillyStringUtils  
// SPDX-License-Identifier: MIT  
  
pragma solidity ^0.8.17;  
  
library SillyStringUtils {  
  
 struct Haiku {  
 string line1;  
 string line2;  
 string line3;  
 }  
  
 function shruggie(string memory \_input) internal pure returns (string memory) {  
 return string.concat(\_input, unicode" 🤷");  
 }  
}  
Add a public instance of Haiku called haiku.  
Add the following two functions.  
Save Haiku  
saveHaiku should accept three strings and save them as the lines of haiku.  
Get Haiku  
getHaiku should return the haiku as a Haiku type.  
INFO  
Remember, the compiler will automatically create a getter for public structs, but these return each member individually. Create your own getters to return the type.  
Shruggie Haiku  
shruggieHaiku should use the library to add 🤷 to the end of line3. It must not modify the original haiku. It should return the modified Haiku.  
Submit your Contract and Earn an NFT Badge! (BETA)  
CAUTION  
Contract Verification Best Practices  
To simplify the verification of your contract on a blockchain explorer like BaseScan.org, consider these two common strategies:  
Flattening: This method involves combining your main contract and all of its imported dependencies into a single file. This makes it easier for explorers to verify the code since they only have to process one file.  
Modular Deployment: Alternatively, you can deploy each imported contract separately and then reference them in your main contract via their deployed addresses. This approach maintains the modularity and readability of your code. Each contract is deployed and verified independently, which can facilitate easier updates and reusability.  
Use Desktop Tools: Forge and Hardhat both have tools to write scripts that both deploy and verify your contracts.  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Submit your Contract and Earn an NFT Badge! (BETA)  
Save Haiku  
Get Haiku  
Shruggie Haiku  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/imports/imports-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Imports  
Imports  
Imports Exercise  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Imports  
In this lesson, we'll learn how to import code written by others into your contracts. We'll also explore the OpenZeppelin library of smart contracts.  
Objectives  
By the end of this lesson you should be able to:  
Import and use code from another file  
Utilize OpenZeppelin contracts within Remix  
OpenZeppelin  
OpenZeppelin has a robust library of well-documented smart contracts. These include a number of standard-compliant token implementations and a suite of utilities. All the contracts are audited and are therefore safer to use than random code you might find on the internet (you should still do your own audits before releasing to production).  
Docs  
The docs start with installation instructions, which we'll return to when we switch over to local development. You do not need to install anything to use these contracts in Remix.  
Find the documentation for the EnumerableSet under Utils. This library will allow you to create sets of bytes32, address, and uint256. Since they're enumerated, you can iterate through them. Neat!  
Implementing the OpenZeppelin EnumeratedSet  
Create a new file to work in and add the pragma and license identifier.  
In Remix, you can import libraries directly from Github!  
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/EnumerableSet.sol";  
You should see EnumerableSet.sol pop into your workspace files, nested deeply in a bunch of folders.  
Trying It Out  
Add a contract called SetExploration. Review the extensive comments within the contract itself.  
To use the EnumerableSet, you need to use the using keyword. This directive attaches all of the library methods to the type. Doing so allows you to call the method on the variable with dot notation, and the variable itself will be supplied as the first argument.  
Follow the pattern in the example in the comments, but name the variable visitors:  
using EnumerableSet for EnumerableSet.AddressSet;  
  
EnumerableSet.AddressSet private visitors;  
Add a function called registerVisitor that makes use of the library's add function to add the sender of the message to the visitors set.  
TIP  
There's also an \_add function, which is private.  
Reveal code  
  
  
  
  
Add another function to return the numberOfVisitors. Thanks to using, this can cleanly call the length function:  
Reveal code  
  
  
  
Conclusion  
In this lesson, you imported a library from OpenZeppelin and implemented some of its functions. You also learned how to use the using keyword.  
ON THIS PAGE  
OpenZeppelin  
Conclusion  
Docs  
Implementing the OpenZeppelin EnumeratedSet  
Trying It Out  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/imports/imports-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Imports  
Imports  
Imports Exercise  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Imports  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/inheritance/abstract-contracts-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Inheritance  
Inheritance  
Multiple Inheritance  
Multiple Inheritance  
Abstract Contracts  
Abstract Contracts  
Inheritance Exercise  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Abstract Contracts  
Abstract contracts can't exist on their own. Their functionality can only be utilized by a contract that inherits from them. In this lesson, you'll learn how to create an abstract contract.  
Objectives  
By the end of this lesson you should be able to:  
Use the virtual, override, and abstract keywords to create and use an abstract contract  
Abstract Contracts  
Continue with your Inheritance.sol file. Add ContractD as an abstract contract. Add a virtual function called whoAreYou function, but do not add any implementation for that function.  
Reveal code  
  
  
  
  
Inheriting from an Abstract Function  
Update ContractA to inherit from ContractD.  
You'll get a slightly confusing error that ContractA needs to be marked as abstract. Doing so is not the correct fix.  
from solidity:  
TypeError: Contract "ContractA" should be marked as abstract.  
 --> contracts/Inheritance.sol:25:1:  
 |  
25 | contract ContractA is ContractB, ContractC, ContractD {  
 | ^ (Relevant source part starts here and spans across multiple lines).  
Note: Missing implementation:  
 --> contracts/Inheritance.sol:6:5:  
 |  
6 | function whoAreYou() public virtual view returns (string memory);  
 | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
The clue for the correct solution is further down: Note: Missing implementation:  
Only abstract contracts can declare functions that are not implemented. To fix this, provide an override implementation for whoAreYou in ContractA:  
Reveal code  
  
  
  
Conclusion  
In this lesson, you've learned how to implement and inherit from an abstract contract.  
ON THIS PAGE  
Abstract Contracts  
Conclusion  
Inheriting from an Abstract Function  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/inheritance/abstract-contracts-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Inheritance  
Inheritance  
Multiple Inheritance  
Multiple Inheritance  
Abstract Contracts  
Abstract Contracts  
Inheritance Exercise  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Abstract Contracts  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/inheritance/inheritance-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Inheritance  
Inheritance  
Multiple Inheritance  
Multiple Inheritance  
Abstract Contracts  
Abstract Contracts  
Inheritance Exercise  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Inheritance Exercise  
Create contracts that adhere to the following specifications.  
Contracts  
Employee  
Create an abstract contract called Employee. It should have:  
A public variable storing idNumber  
A public variable storing managerId  
A constructor that accepts arguments for and sets both of these variables  
A virtual function called getAnnualCost that returns a uint  
Salaried  
A contract called Salaried. It should:  
Inherit from Employee  
Have a public variable for annualSalary  
Implement an override function for getAnnualCost that returns annualSalary  
An appropriate constructor that performs any setup, including setting annualSalary  
Hourly  
Implement a contract called Hourly. It should:  
Inherit from Employee  
Have a public variable storing hourlyRate  
Include any other necessary setup and implementation  
TIP  
The annual cost of an hourly employee is their hourly rate \* 2080 hours.  
Manager  
Implement a contract called Manager. It should:  
Have a public array storing employee Ids  
Include a function called addReport that can add id numbers to that array  
Include a function called resetReports that can reset that array to empty  
Salesperson  
Implement a contract called Salesperson that inherits from Hourly.  
Engineering Manager  
Implement a contract called EngineeringManager that inherits from Salaried and Manager.  
Deployments  
You'll have to do a more complicated set of deployments for this exercise.  
Deploy your Salesperson and EngineeringManager contracts. You don't need to separately deploy the other contracts.  
Use the following values:  
Salesperson  
Hourly rate is 20 dollars an hour  
Id number is 55555  
Manager Id number is 12345  
Manager  
Annual salary is 200,000  
Id number is 54321  
Manager Id is 11111  
Inheritance Submission  
Copy the below contract and deploy it using the addresses of your Salesperson and EngineeringManager contracts.  
contract InheritanceSubmission {  
 address public salesPerson;  
 address public engineeringManager;  
  
 constructor(address \_salesPerson, address \_engineeringManager) {  
 salesPerson = \_salesPerson;  
 engineeringManager = \_engineeringManager;  
 }  
}  
Submit your Contracts and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
CAUTION  
Submit your address for your copy of the InheritanceSubmission contract that contains your other contract addresses.  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Deployments  
Salesperson  
Manager  
Hourly  
Manager  
Salesperson  
Engineering Manager  
Inheritance Submission  
Salesperson  
Manager  
Submit your Contracts and Earn an NFT Badge! (BETA)  
Submit your Contracts and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/inheritance/inheritance-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Inheritance  
Inheritance  
Multiple Inheritance  
Multiple Inheritance  
Abstract Contracts  
Abstract Contracts  
Inheritance Exercise  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Inheritance  
Solidity is an object-oriented language. Contracts can inherit from one another, allowing efficient reuse of code.  
Objectives  
By the end of this lesson you should be able to:  
Write a smart contract that inherits from another contract  
Describe the impact inheritance has on the byte code size limit  
Inheritance  
Create a new contract file in Remix called Inheritance.sol and add two simple contracts, each with a function identifying which contract called it:  
// SPDX-License-Identifier: MIT  
  
pragma solidity ^0.8.17;  
  
contract ContractB {  
 function whoAmI() external pure returns (string memory) {  
 return "contract B";  
 }  
}  
  
contract ContractA {  
 function whoAmI() external pure returns (string memory) {  
 return "contract A";  
 }  
}  
ContractA says that it is "contract A" and ContractB says that it is "contract B".  
Inheriting from Another Contract  
Inheritance between contracts is indicated by the is keyword in the contract declaration. Update ContractA so that it is ContractB, and delete the whoAmI function from ContractA.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
Deploy and test again. Even though ContractA doesn't have any functions in it, the deployment still shows the button to call whoAmI. Call it. ContractA now reports that it is "contract B", due to the inheritance of the function from Contract B.  
Internal Functions and Inheritance  
Contracts can call the internal functions from contracts they inherit from. Add an internal function to ContractB called whoAmIInternal that returns "contract B".  
Add an external function called whoAmIExternal that returns the results of a call to whoAmIInternal.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Deploy and test. Note that in the deployment for ContractB, the whoAmIInternal function is not available, as it is internal. However, calling whoAmIExternal can call the internal function and return the expected result of "contract B".  
Internal vs. Private  
You cannot call a private function from a contract that inherits from the contract containing that function.  
// Bad code example, do not use  
contract ContractB {  
 function whoAmIPrivate() private pure returns (string memory) {  
 return "contract B";  
 }  
}  
  
contract ContractA is ContractB {  
 function whoAmExternal() external pure returns (string memory) {  
 return whoAmIPrivate();  
 }  
}  
The compiler will raise an error:  
from solidity:  
DeclarationError: Undeclared identifier.  
 --> contracts/Inheritance.sol:17:16:  
 |  
17 | return whoAmIPrivate();  
 | ^^^^^^^^^^^^^  
Inheritance and Contract Size  
A contract that inherits from another contract will have that contract's bytecode included within its own. You can view this by opening settings in Remix and turning Artifact Generation back on. The bytecode for each compiled contract will be present in the JSON file matching that contract's name within the artifacts folder.  
Any empty contract:  
contract EmptyContract {  
  
}  
Will compile into something similar to this:  
6080604052600080fdfea2646970667358221220df894b82f904e22617d7e40150306e2d2e8cb2ca5dcacb666a0c3d40f5f988c464736f6c63430008110033  
A slightly more complex contract:  
contract notEmptyContract {  
 function sayHello() public pure returns (string memory) {  
 return "To whom it may concern, I write you after a long period of silence to alert you that after much reflection, it occurs to me that I don't think you have fully considered...";  
 }  
}  
Will have more complex bytecode. In this case, mostly to store the long string present in the return:  
  
However, if the empty contract inherits from the not empty contract:  
contract EmptyContract is notEmptyContract {  
  
}  
The resulting bytecode will include that of the contract inherited from:  
  
Conclusion  
In this lesson, you've learned how to use inheritance to include the functionality of one contract in another. You've also learned that inheriting contracts can call internal functions, but they cannot call private functions. You've also learned that inheriting from a contract adds the size of that contract's bytecode to the total deployed size.  
ON THIS PAGE  
Inheritance  
Conclusion  
Inheriting from Another Contract  
Internal Functions and Inheritance  
Internal vs. Private  
Inheritance and Contract Size  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/inheritance/inheritance-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Inheritance  
Inheritance  
Multiple Inheritance  
Multiple Inheritance  
Abstract Contracts  
Abstract Contracts  
Inheritance Exercise  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Inheritance  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/inheritance/multiple-inheritance  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Inheritance  
Inheritance  
Multiple Inheritance  
Multiple Inheritance  
Abstract Contracts  
Abstract Contracts  
Inheritance Exercise  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Multiple Inheritance  
Contracts can inherit from more than one contract. In this lesson, we'll explore how multiple inheritance works in Solidity.  
Objectives  
By the end of this lesson you should be able to:  
Write a smart contract that inherits from multiple contracts  
Multiple Inheritance  
Continue working with your contracts in Inheritance.sol. Add a new contract called ContractC with another whoAmI function:  
Reveal code  
  
  
  
  
  
  
Inheriting from Two Contracts  
You can inherit from additional contracts by simply adding a comma and that contract's name after the first. Add inheritance from ContractC (an error is expected):  
Reveal code  
  
  
  
  
  
  
  
The error is because both ContractB and ContractC contain a function called whoAmI. As a result, the compiler needs instruction on which to use.  
from solidity:  
TypeError: Derived contract must override function "whoAmI". Two or more base classes define function with same name and parameter types.  
 --> contracts/Inheritance.sol:21:1:  
 |  
21 | contract ContractA is ContractB, ContractC {  
 | ^ (Relevant source part starts here and spans across multiple lines).  
Note: Definition in "ContractC":  
 --> contracts/Inheritance.sol:6:5:  
 |  
6 | function whoAmI() external pure returns (string memory) {  
 | ^ (Relevant source part starts here and spans across multiple lines).  
Note: Definition in "ContractB":  
 --> contracts/Inheritance.sol:12:5:  
 |  
12 | function whoAmI() external pure returns (string memory) {  
 | ^ (Relevant source part starts here and spans across multiple lines).  
Using Virtual and Override  
One method to resolve this conflict is to use the virtual and override keywords to enable you to add functionality to choose which to call.  
Add the virtual keyword to the whoAmI function in both ContractC and ContractB.  
They must also be made public instead of external, because external functions cannot be called within the contract.  
contract ContractC {  
 function whoAmI() public virtual pure returns (string memory) {  
 return "contract C";  
 }  
}  
  
contract ContractB {  
 function whoAmI() public virtual pure returns (string memory) {  
 return "contract B";  
 }  
  
 // ... additional code  
}  
Add an override function called whoAmI to ContractA:  
// Bad code example, do not use  
function whoAmI() public override pure returns (string memory) {  
 return ContractB.whoAmI();  
}  
You'll get another error, telling you to specify which contracts this function should override.  
from solidity:  
TypeError: Function needs to specify overridden contracts "ContractB" and "ContractC".  
 --> contracts/Inheritance.sol:22:32:  
 |  
22 | function whoAmI() public override pure returns (string memory) {  
 | ^^^^^^^^  
Add them both:  
function whoAmI() external override(ContractB, ContractC) pure returns (string memory) {  
 return ContractB.whoAmI();  
}  
Deploy and test. The call will now be back to reporting "contract B".  
Changing Types Dynamically  
Add an enum at the contract level in ContractA with members for None, ContractBType, and ContractCType, and an instance of it called contractType.  
Reveal code  
  
  
  
  
Add a constructor to ContractA that accepts a Type and sets initialType.  
Reveal code  
  
  
  
  
Update whoAmI in ContractA to call the appropriate virtual function based on its currentType.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
You'll get errors because the function now reads from state, so it is no longer pure. Update it to view. You'll also have to update the whoAmI virtual functions to view to match.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
Finally, add a function that allows you to switch currentType:  
Reveal code  
  
  
  
  
Deploy and test. You'll need to use 0, 1, and 2 as values to set contractType, because Remix won't know about your enum.  
Final Code  
After completing this exercise, you should have something similar to:  
// SPDX-License-Identifier: MIT  
  
pragma solidity ^0.8.17;  
  
contract ContractC {  
 function whoAmI() public virtual view returns (string memory) {  
 return "contract C";  
 }  
}  
  
contract ContractB {  
 function whoAmI() public virtual view returns (string memory) {  
 return "contract B";  
 }  
  
 function whoAmIInternal() internal pure returns (string memory) {  
 return "contract B";  
 }  
}  
  
contract ContractA is ContractB, ContractC {  
 enum Type { None, ContractBType, ContractCType }  
  
 Type contractType;  
  
 constructor (Type \_initialType) {  
 contractType = \_initialType;  
 }  
  
 function changeType(Type \_newType) external {  
 contractType = \_newType;  
 }  
  
 function whoAmI() public override(ContractB, ContractC) view returns (string memory) {  
 if(contractType == Type.ContractBType) {  
 return ContractB.whoAmI();  
 }  
 if(contractType == Type.ContractCType) {  
 return ContractC.whoAmI();  
 }  
 return "contract A";  
 }  
  
 function whoAmExternal() external pure returns (string memory) {  
 return whoAmIInternal();  
 }  
}  
Conclusion  
In this lesson, you've explored how to use multiple inheritance to import additional functionality into a contract. You've also implemented one approach to resolving name conflicts between those contracts.  
ON THIS PAGE  
Multiple Inheritance  
Final Code  
Inheriting from Two Contracts  
Using Virtual and Override  
Changing Types Dynamically  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/inheritance/multiple-inheritance-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Inheritance  
Inheritance  
Multiple Inheritance  
Multiple Inheritance  
Abstract Contracts  
Abstract Contracts  
Inheritance Exercise  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Multiple Inheritance  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/interfaces/calling-another-contract-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Intro to Interfaces  
Calling Another Contract  
Testing the Interface  
Contract to Contract Interaction  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Calling Another Contract  
CAUTION  
This tutorial has been moved as part of a reorganization! It assumes you are using Hardhat. Everything in this lesson will work with minor adjustments if you are working in Foundry or Remix.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/interfaces/contract-to-contract-interaction  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Intro to Interfaces  
Calling Another Contract  
Testing the Interface  
Contract to Contract Interaction  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Contract to Contract Interaction  
In this article, you'll learn how to interact with other smart contracts using interfaces and the .call() function, which allows you to interact with other smart contracts without using an interface.  
CAUTION  
This tutorial has been moved as part of a reorganization! It assumes you are using Hardhat. Everything in this lesson will work with minor adjustments if you are working in Foundry or Remix.  
Objectives  
By the end of this lesson you should be able to:  
Use interfaces to allow a smart contract to call functions in another smart contract  
Use the call() function to interact with another contract without using an interface  
Overview  
Interacting with external smart contracts is a very common task in the life of a smart contract developer. This includes interacting with contracts that are already deployed to a particular network.  
Usually the creators of certain smart contracts document their functionality and expose their functions by providing interfaces that can be used to integrate those particular contracts into your own.  
For instance, Uniswap provides documentation on how to interact with their smart contracts and also some packages to easily integrate their protocol.  
In this example, you interact with the Uniswap protocol to create a custom pool for a custom pair of tokens.  
Since the Uniswap protocol is already deployed, you will use Hardhat forking to test your contract.  
You will also use the following two approaches in the example:  
Using interfaces  
Using the .call() function  
Interacting with deployed contracts using interfaces  
You must first install the Uniswap V3 core package by running:  
npm install @uniswap/v3-core  
This package provides access to the Uniswap interfaces of the Core protocol.  
Then, write a custom contract called PoolCreator with the following code:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol";  
  
contract PoolCreator {  
 IUniswapV3Factory public uniswapFactory;  
  
 constructor(address \_factoryAddress) {  
 uniswapFactory = IUniswapV3Factory(\_factoryAddress);  
 }  
  
 function createPool(  
 address tokenA,  
 address tokenB,  
 uint24 fee  
 ) external returns (address poolAddress) {  
 // Check if a pool with the given tokens and fee already exists  
 poolAddress = uniswapFactory.getPool(tokenA, tokenB, fee);  
 if (poolAddress == address(0)) {  
 // If the pool doesn't exist, create a new one  
 poolAddress = uniswapFactory.createPool(tokenA, tokenB, fee);  
 }  
  
 return poolAddress;  
 }  
}  
Notice the following:  
You are importing a IUniswapV3Factory interface. The interface contains function declarations that include getPool and createPool:  
// SPDX-License-Identifier: GPL-2.0-or-later  
pragma solidity >=0.5.0;  
  
/// @title The interface for the Uniswap V3 Factory  
/// @notice The Uniswap V3 Factory facilitates creation of Uniswap V3 pools and control over the protocol fees  
interface IUniswapV3Factory {  
 // ...  
 // ...other function declarations  
  
 /// @notice Returns the pool address for a given pair of tokens and a fee, or address 0 if it does not exist  
 /// @dev tokenA and tokenB may be passed in either token0/token1 or token1/token0 order  
 /// @param tokenA The contract address of either token0 or token1  
 /// @param tokenB The contract address of the other token  
 /// @param fee The fee collected upon every swap in the pool, denominated in hundredths of a bip  
 /// @return pool The pool address  
 function getPool(  
 address tokenA,  
 address tokenB,  
 uint24 fee  
 ) external view returns (address pool);  
  
 /// @notice Creates a pool for the given two tokens and fee  
 /// @param tokenA One of the two tokens in the desired pool  
 /// @param tokenB The other of the two tokens in the desired pool  
 /// @param fee The desired fee for the pool  
 /// @dev tokenA and tokenB may be passed in either order: token0/token1 or token1/token0. tickSpacing is retrieved  
 /// from the fee. The call will revert if the pool already exists, the fee is invalid, or the token arguments  
 /// are invalid.  
 /// @return pool The address of the newly created pool  
 function createPool(  
 address tokenA,  
 address tokenB,  
 uint24 fee  
 ) external returns (address pool);  
The constructor receives the address of the pool factory and creates an instance of IUniswapV3Factory.  
The createPool function includes a validation to ensure the pool doesn't exist.  
The createPool function creates a new pool.  
Then, create a test file called PoolCreator.test.ts with the following content:  
import { ethers } from 'hardhat';  
import { HardhatEthersSigner } from '@nomicfoundation/hardhat-ethers/signers';  
  
import { Token, Token\_\_factory, PoolCreator, PoolCreator\_\_factory } from '../typechain-types';  
  
describe('PoolCreator tests', function () {  
 const UNISWAP\_FACTORY\_ADDRESS = '0x1F98431c8aD98523631AE4a59f267346ea31F984';  
 let tokenA: Token;  
 let tokenB: Token;  
 let poolCreator: PoolCreator;  
 let owner: HardhatEthersSigner;  
  
 before(async () => {  
 const signers = await ethers.getSigners();  
 owner = signers[0];  
 tokenA = await new Token\_\_factory().connect(owner).deploy('TokenA', 'TokenA');  
 tokenB = await new Token\_\_factory().connect(owner).deploy('TokenB', 'TokenB');  
 poolCreator = await new PoolCreator\_\_factory().connect(owner).deploy(UNISWAP\_FACTORY\_ADDRESS);  
 });  
  
 it('should create a pool', async () => {  
 const contractAddress = await poolCreator.createPool.staticCall(tokenA, tokenB, 500);  
 console.log('Contract Address', contractAddress);  
 await poolCreator.createPool(tokenA, tokenB, 500);  
 });  
});  
Notice the following:  
The address 0x1F98431c8aD98523631AE4a59f267346ea31F984 is the address of the Uniswap pool factory deployed to the Ethereum mainnet. This can be verified by looking at the Uniswap documentation that includes the Deployment addresses of the contracts.  
You created two tokens, TokenA and TokenB, by using a Token contract.  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";  
  
contract Token is ERC20 {  
 constructor(string memory name, string memory symbol) ERC20(name, symbol){  
 \_mint(msg.sender, 1000 ether);  
 }  
}  
Finally, run npx hardhat test and you should get a result similar to the following:  
PoolCreator tests  
Contract Address 0xa76662f79A5bC06e459d0a841190C7a4e093b04d  
 ✔ should create a pool (1284ms)  
  
 1 passing (5s)  
Interacting with external contracts using .call()  
In the previous example, you accessed the Uniswap V3 Factory interface, however if you don't have access to the contract interface, you can use a special function called call.  
Using call, you can call any contract as long as you know minimal information of the function signature. In this case, you should at least know that createPool requires three parameters:  
tokenA  
tokenB  
fee  
The newly modified smart contract code looks as follows:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract PoolCreator {  
 address public uniswapFactory;  
  
 constructor(address \_factoryAddress) {  
 uniswapFactory = \_factoryAddress;  
 }  
  
 function createPool(  
 address tokenA,  
 address tokenB,  
 uint24 fee  
 ) external returns (address poolAddress) {  
 bytes memory payload = abi.encodeWithSignature(  
 "createPool(address,address,uint24)",  
 tokenA,  
 tokenB,  
 fee  
 );  
  
 (bool success, bytes memory data) = uniswapFactory.call(payload);  
 require(success, "Uniswap factory call failed");  
  
 // The pool address should be returned as the first 32 bytes of the data  
 assembly {  
 poolAddress := mload(add(data, 32))  
 }  
  
 require(poolAddress != address(0), "Pool creation failed");  
 return poolAddress;  
 }  
}  
Notice the following:  
By using abi.encodeWithSignature, you encode the payload required to make a smart contract call using the .call() function.  
Using .call() doesn't require you to import the interface.  
You load the pool address by using a special assembly operation called mload.  
Try to run again the command npx hardhat test and you should expect the same result:  
PoolCreator tests  
Contract Address 0xa76662f79A5bC06e459d0a841190C7a4e093b04d  
 ✔ should create a pool (1284ms)  
  
 1 passing (5s)  
Conclusion  
Interfaces or the .call function are two ways to interact with external contracts. Using interfaces provides several advantages, including type safety, code readability, and compiler error checking. When interacting with well-documented contracts like Uniswap, using interfaces is often the preferred and safest approach.  
On the other hand, the .call function offers more flexibility but comes with greater responsibility. It allows developers to call functions on contracts even without prior knowledge of their interfaces. However, it lacks the type safety and error checking provided by interfaces, making it more error-prone.  
ON THIS PAGE  
Overview  
Interacting with deployed contracts using interfaces  
Interacting with external contracts using .call()  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/interfaces/intro-to-interfaces-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Intro to Interfaces  
Calling Another Contract  
Testing the Interface  
Contract to Contract Interaction  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Intro to Interfaces  
CAUTION  
This tutorial has been moved as part of a reorganization! It assumes you are using Hardhat. Everything in this lesson will work with minor adjustments if you are working in Foundry or Remix.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/interfaces/testing-the-interface-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Intro to Interfaces  
Calling Another Contract  
Testing the Interface  
Contract to Contract Interaction  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Testing the Interface  
CAUTION  
This tutorial has been moved as part of a reorganization! It assumes you are using Hardhat. Everything in this lesson will work with minor adjustments if you are working in Foundry or Remix.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/intro-to-tokens/intro-to-tokens-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Introduction  
Common Misconceptions  
Overview  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Introduction  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/intro-to-tokens/misconceptions-about-tokens-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Introduction  
Common Misconceptions  
Overview  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Common Misconceptions  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/intro-to-tokens/tokens-overview  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Introduction  
Common Misconceptions  
Overview  
Minimal Tokens  
ERC-20 Tokens  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Overview  
This article will provide an overview of the most popular token standards on Ethereum, including ERC-20, ERC-721, ERC-1155, and a discussion on their properties and various use cases.  
Objectives:  
By the end of this lesson you should be able to:  
Describe the properties of ERC-20 and ERC-721 tokens  
List popular ERC-721 tokens  
List the uses for ERC-20, ERC-721, and ERC-1155 tokens  
ERC Token Standards  
Ethereum Request for Comments (or, ERC) is a term used to describe technical proposals and standards for Ethereum. An ERC is authored by developers and members of the Ethereum community to suggest improvements, new features, or guidelines for creating and managing tokens and smart contracts. Once an ERC is submitted, it undergoes review and discussion by the community. If it gains consensus, it can then be implemented or adopted as a standard in the ecosystem.  
Token standards on Ethereum form the backbone of the digital asset ecosystem. They are a set of predefined rules and guidelines that govern the creation, management, and interaction of tokens on the network. These standards ensure that tokens are compatible with various apps, wallets, and other tokens within the Ethereum ecosystem. Token standards allow developers to create tokens with consistent behavior, facilitating seamless interaction and interoperability within the network.  
ERC-20  
ERC-20 tokens, the most widely-used token standard on Ethereum, possess several key properties that make them versatile and flexible for various applications. One of the defining characteristics of these tokens is their fungibility. Each unit of an ERC-20 token is interchangeable and holds equal value to another unit of the same token, rendering them indistinguishable from one another. In other words, one USDC token is always equal in value and interchangeable with another USDC token.  
Another aspect of ERC-20 tokens is their standardized interface, which includes a set of six mandatory functions: totalSupply(), balanceOf(address), transfer(address, uint256), transferFrom(address, address, uint256), approve(address, uint256), and allowance(address, address). This standardization ensures consistency when interacting with these tokens, irrespective of their specific implementation or use case. For example, a user can easily check their token balance or transfer tokens using the same set of functions, whether they are interacting with a governance token like UNI or a stablecoin like DAI.  
Some notable applications of ERC-20 tokens include utility tokens (FIL, BAT, MANA), governance tokens (UNI, AAVE, COMP), and stablecoins (USDC, USDT, DAI).  
ERC-721  
ERC-721 is a prominent token standard specifically designed for NFTs, allowing for the creation and management of unique, indivisible digital assets that each have their own special properties.  
In contrast to ERC-20 tokens, which are fungible and can be easily exchanged, ERC-721 tokens are non-fungible and can't be swapped on a one-to-one basis. Every token has its own attributes that set it apart from the rest. This one-of-a-kind nature enables the representation of a wide range of digital assets, including digital art, virtual real estate, and collectibles. For example, an artist could mint a one-of-a-kind digital painting, a virtual land parcel could be tokenized in a metaverse, or a rare sports card could be digitized as a collectible NFT.  
ERC-721 tokens, like ERC-20 tokens, follow a standardized interface but employ a unique set of functions designed for non-fungible tokens, which allow developers to interact with NFTs across multiple platforms. For instance, a developer would use the same set of functions to interact with a digital artwork NFT listed on OpenSea as they would with a virtual land parcel NFT in Decentraland.  
Besides their unique qualities, ERC-721 tokens come with metadata properties that offer information about the token's specific features, such as the artwork's title, the artist, and an image preview. This metadata helps users better understand and appreciate the distinct aspects of each NFT, and it is consistent across platforms.  
Some notable applications of ERC-721 tokens include digital art by Beeple, virtual collectibles by NBA Top Shot, virtual real estate in Decentraland, and Ethereum-based domain names like vitalik.eth on the Ethereum Name Service (ENS).  
ERC-1155  
ERC-1155 is an innovative hybrid token standard that merges the best aspects of both fungible and non-fungible tokens, enabling developers to create and manage diverse token types using a single smart contract. This combination of features allows ERC-1155 tokens to provide greater versatility while representing a wide array of assets with different levels of fungibility.  
For example, a video game might use both fungible and non-fungible tokens within its ecosystem. Fungible tokens could represent in-game currencies, consumables, or resources, while non-fungible tokens could represent exclusive and unique items like character skins, weapons, or collectible cards.  
Digital artists can also benefit from ERC-1155, as it allows them to mint limited edition series of their artwork, with each piece in the series having unique attributes. At the same time, they can create fungible tokens that represent ownership of a specific edition number within the series.  
Similar to other token standards, ERC-1155 tokens adhere to a standardized interface with a set of functions that ensure consistency and compatibility across platforms and services. Furthermore, this standard enables efficient batch transfers, simplifying the process and reducing the cost of managing multiple tokens within a single application. For instance, a user who has collected various in-game items in a virtual world can leverage ERC-1155's batch transfer feature to send multiple fungible and non-fungible tokens to another user or marketplace simultaneously. This efficient approach minimizes transaction costs and the complexity typically involved in transferring numerous tokens one by one.  
Other Token Standards  
In addition to the three most prominent token standards that we covered, it is worth mentioning that other standards like ERC-777 and ERC-4626 have been introduced to address specific use cases or challenges. ERC-777 offers enhanced security and functionality over the fungible ERC-20 standard, while ERC-4626 streamlines yield-bearing vault integration by optimizing and unifying technical parameters. These lesser-known standards highlight the ongoing innovation and adaptability of the Ethereum token ecosystem as it continues to grow and evolve.  
Conclusion  
Ethereum's ERC token standards have played a pivotal role in shaping the digital asset ecosystem by providing clear guidelines and rules for the creation, management, and interaction of tokens on the network. From the widely-used ERC-20 standard for fungible tokens to the distinct ERC-721 standard for non-fungible tokens and the versatile hybrid ERC-1155 standard, these token standards empower developers to craft diverse tokens tailored to various use cases and applications. The standardized interfaces ensure seamless interoperability within the Ethereum ecosystem, facilitating token transfers and interactions across different platforms and services. Additional token standards, such as ERC-777 and ERC-4626, address specific challenges and further demonstrate the continuous innovation and adaptability of the Ethereum token ecosystem.  
See Also  
EIP-20  
EIP-721  
EIP-1155  
EIP-777  
EIP-4626  
ON THIS PAGE  
ERC Token Standards  
ERC-20  
ERC-721  
ERC-1155  
Other Token Standards  
Conclusion  
See Also  
See Also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-ethereum/ethereum-applications  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Introduction  
Ethereum Applications  
Ethereum Applications  
Gas Use in Ethereum Transactions  
EVM Diagram  
Guide to Base  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Ethereum Applications  
In this article, we'll explore Ethereum's significance and impact in the crypto ecosystem as well as its role in shaping the Web3 landscape. We'll learn about Ethereum's ethos and goals and also examine the different types of applications developed on Ethereum. Lastly, we'll take a look at the evolution of the web with an emphasis on comparing Web2 and Web3 development.  
Objectives  
By the end of this lesson you should be able to:  
Describe the origin and goals of the Ethereum blockchain  
List common types of applications that can be developed with the Ethereum blockchain  
Compare and contrast Web2 vs. Web3 development  
Compare and contrast the concept of "ownership" in Web2 vs. Web3  
The Ethos and Goals of Ethereum  
Ethereum was originally proposed in 2013 by Vitalik Buterin, who was then a young developer in the Bitcoin community. Vitalik had a vision that the potential of blockchain technology extended far beyond a decentralized digital currency. When his ideas were rejected by the Bitcoin community, he set out to create a platform that could bring his vision to life.  
The ethos of Ethereum is fundamentally different from Bitcoin's. Bitcoin development is conservative; it's focused on maintaining the existing protocol, making only incremental improvements over time rather than implementing radical changes. In other words, changes are slow and deliberate and any unnecessary risk-taking is generally frowned upon. Ethereum development, on the other hand, is focused on innovation and experimentation. There is more of a willingness to take risks and make radical changes to the protocol in order to improve on and expand upon functionality and enable new use cases.  
Ethereum's primary goal is to be a general, all-purpose blockchain that allows developers to create any type of decentralized application that their minds can conjure up. One of the most important features that unlock all of these possibilities is smart contracts. Without smart contract functionality, most applications built on the platform today would be nonexistent.  
Applications on Ethereum  
Before delving into the different types of applications built on Ethereum, let's review one of the underlying forces of the smart contracts that make them possible.  
Scripting  
As we've learned, one of Bitcoin's limitations when it comes to complex applications being built on the protocol is scripting. Its simple stack-based, left-to-right scripting system lacks the flexibility to support smart contracts. In the Ethereum whitepaper, Vitalik pointed out several limitations to Bitcoin scripting. A couple of key ones to note are:  
Lack of Turing-completeness. Although this is an intentional feature of Bitcoin to avoid infinite loops during transaction verification, without loops or recursion, running a complex program that a decentralized application demands, is not possible.  
Lack of state. Bitcoin's UTXO model only allows for simple contracts and not the complex stateful contracts needed for most decentralized applications. What that means is that there is no internal state beyond a UTXO being spent or unspent.  
Ethereum's scripting languages, most notably Solidity, are Turing-complete and stateful, among other features. These features allow smart contracts to be executed deterministically, meaning that the outcome of the contract is predictable and can be enforced automatically and autonomously. They also allow developers to write much more complex programs that can execute a much wider range of operations.  
It is this flexibility and versatility in Ethereum's scripting that ultimately powers the decentralized applications that we know.  
Decentralized Finance (DeFi)  
DeFi is one of the most popular use cases for Ethereum. In DeFi Summer 2020, we saw an explosion in the usage and popularity of DeFi applications built on Ethereum. During this brief period alone, the total value locked in DeFi protocols increased from less than $1 billion to more than $10 billion. This period marked a turning point, as it brought more attention to the space, and it solidified Ethereum as the de facto smart contract platform of the DeFi and greater Web3 ecosystem.  
DeFi applications are designed to provide traditional financial services, such as lending, borrowing, trading, and much more, in a transparent, open and accessible manner. All of these services are facilitated by smart contracts.  
How exactly does this work? Let's take a look at a simple example in the context of a DeFi lending platform, such as Aave or Compound.  
Suppose Alice wants to borrow 5 ETH but doesn't want to sell her 10,000 USDC. She can deposit that USDC as collateral and borrow 5 ETH against it.  
First, Alice must interact with the smart contract of the platform. The smart contract will check if Alice has 10,000 USDC in her wallet and, if so, will lock it up as collateral for the loan. The smart contract then transfers 5 ETH to Alice's wallet. The smart contract will also define the terms of the loan, such as the interest rate and the repayment date.  
Alice now has 5 ETH that she can use for whatever she wants. However, she must repay the loan within the specified period. If Alice repays the loan on time, the smart contract will release her deposited collateral back to her. Otherwise, it will automatically liquidate her collateral and transfer it to the lender's address in exchange for the 5 ETH that was lent to her. The repayment will also include any accrued interest based on the interest rate set by the smart contract.  
In this way, smart contracts enable DeFi platforms to operate autonomously without a centralized entity. The smart contract provides security and transparency to both the borrower and the lender, as the terms of the loan are defined in the code and enforced automatically.  
Of course, this does not mean that DeFi comes without risks. Although it's beyond the scope of this article, it's worth mentioning that DeFi has been the target of numerous smart contract exploits involving hundreds of millions of dollars in value.  
Non-Fungible Token (NFT)  
NFTs are another application of Ethereum that has gained significant attention in recent years. NFTs are unique digital assets that represent ownership of a specific item. They can be used to represent just about anything, but most notably digital artwork, sports collectibles, and in-game items.  
Smart contracts play a crucial role in NFTs by providing a way to represent and enforce ownership of the digital asset. When a new NFT is created, a smart contract is deployed on Ethereum that defines the unique characteristics of that asset as well as the ownership information and rules for transferring ownership.  
Ethereum's ERC-721 standard was the first to introduce NFTs in 2017, and it has since become the most popular standard for creating and trading NFTs on Ethereum.  
We'll cover more on tokens and token standards for fungible and non-fungible assets later on in the course.  
Decentralized Autonomous Organization (DAO)  
DAOs are another common use case for Ethereum and also one of the earliest use cases implemented on the network. In simple words, DAOs are software-enabled, community-led organizations. They allow a community to pool resources toward a shared goal, such as buying one of the original copies of the U.S. Constitution or determining the future of a protocol. Because DAOs aren't tied to a physical location, they are able to mobilize quickly and attract resources and capital from all over the world.  
The rules of a DAO are established by community members through the use of smart contracts, which lay the groundwork for how a DAO operates. When a new DAO is created, a contract is deployed to the network. It contains the rules that govern the organization, including how its resources are managed. Members of the DAO can then interact with the contract by sending transactions to the blockchain.  
DAOs typically use a token-based system to govern voting and decision-making. Members of the DAO are issued tokens that represent their ownership and influence within the organization. These tokens can be used to vote on proposals and allocate resources.  
When a proposal is submitted to the DAO, members can vote on whether to accept or reject it. The smart contract tracks the votes and automatically executes the proposal if it receives enough support from the members. This process allows members of the DAO to collectively make decisions and take actions in a decentralized way.  
Other Applications  
While the above use cases have been the most prominent applications on Ethereum, there are a plethora of others, including:  
Identity Management is one use case that has come to the forefront in recent years. The most notable example is the Ethereum Name Service (ENS). It allows users to register human-readable domain names, similar to the traditional DNS system, but with the added functionality of being able to associate Ethereum and other blockchain addresses with a domain name, such as vitalik.eth. This makes it easier for users to send and receive transactions without having to remember or type in long and complex addresses.  
Gaming is another common use case. Axie Infinity and Decentraland are both popular examples of decentralized games that make use of fungible and non-fungible tokens for a variety of purposes.  
Prediction markets are another use case where users can bet on the outcome of real-world events, such as forecasting election results or predicting which team will win a game. Augur and Gnosis are popular examples of this application.  
There are many other use cases from supply chain, energy, and intellectual property management to decentralized storage and content management to governance and voting systems. The list goes on and on, and it's worth taking some time to explore these types of applications on your own.  
Evolution of the Web  
Opinions abound in the history, divisions, and eras of the development and evolution of the internet. One popular explanation divides the web into three major eras (so far).  
Web1  
The web has come a long way since its humble read-only beginnings in the early 1990s. Web1 is often referred to as the static web, meaning it was primarily a collection of static web pages that provided information to users with limited interaction and dynamic content.  
Web2  
In the early 2000s, a dynamic web emerged. Web2 introduced more interactive and dynamic content, such as social media and e-commerce. It's characterized by the use of centralized servers that store and control user data. In other words, with Web2, users can interact with content, share information, and collaborate with others, but they have limited control over their data. A vast majority of the web today operates in this paradigm.  
Web3  
Web3 or the decentralized web is the next phase of the web that has started to emerge in recent years with the rise in popularity of Ethereum. This iteration of the web is focused on user ownership and control over data, providing a more private, decentralized, and secure web experience.  
The Limitations of Web2  
While Web2 brought many benefits beyond its predecessor, there are some key limitations to consider.  
Privacy & Control: Users have limited control over their data and how it is used. Companies often have broad or even complete control over user data on a given platform.  
Censorship: Due to centralized control of user data, corporations or governments can censor content they deem as inappropriate or dangerous, which can limit free speech and expression or block access to certain online services. This is especially concerning in countries with authoritarian regimes, which often use censorship as a tool to control citizens and maintain power.  
Lack of transparency: Users cannot always verify how their data is being used or who has access to it.  
Security vulnerabilities: Because Web2 relies on centralized servers, it is more vulnerable to hacking and data breaches, which can expose sensitive user information and compromise online safety.  
Limited interoperability: Most Web2 platforms are not interoperable, meaning that different platforms may not be compatible with each other. Data is generally confined to one system.  
The Limitations of Web3  
While many of the limitations of Web2 have spurred the development of Web3 by aiming to provide a more decentralized, secure, and private web, Web3 does not come without its own set of limitations.  
Speed: The reliance on decentralized networks and consensus mechanisms result in much slower processing times compared to centralized systems.  
Storage: Storing data onchain can be very expensive, which can make it challenging for developers to create apps that require large amounts of storage.  
Smart contract limitations:  
Smart contracts on Ethereum are currently limited to a maximum size of 24 KB, which can limit the complexity of the logic that can be programmed into them.  
Once a smart contract is deployed, it cannot be updated or changed. If there is a bug or a flaw in the contract, it cannot be fixed unless a new contract is deployed.  
All data is public: The transparency of blockchain means that all data is public and visible to anyone. While this can be an advantage in terms of transparency and accountability, it can also be a limitation for applications that may require privacy or confidentiality.  
Web2 vs Web3 Development  
While there are many general distinctions to be made between Web2 and Web3, these characteristics are even more apparent when examining their development approaches.  
Web2  
In Web2, engineering is centered around a client-server architecture, and development is focused on building applications for specific platforms and using APIs and tools provided by those platforms to create user interfaces and access data. There is a top-down corporate approach to development processes, and code is generally proprietary and closed-sourced. As a result, there tends to be very limited collaboration with developers outside of a company and there is little integration between different platforms.  
Web2 developers rely on centralized infrastructure, such as servers and cloud-computing services provided by large tech companies to host their applications and store data. This creates a centralized system where the platforms and companies that control the infrastructure have significant power and control over the applications and data that are built on top of them.  
Web3  
In contrast, the Web3 development paradigm is centered around a distributed architecture, where developers build applications that run on decentralized protocols and smart contracts. There is a bottom-up community approach to development processes, and there is an emphasis on open-source code and open standards. Web3 development culture is collaborative, and there is strong integration and interoperability between platforms.  
Web3 development requires a different set of engineering skills and tools. Developers need to have a strong understanding of blockchain technology, cryptography, and distributed systems, and they also need to be proficient in programming languages like Solidity.  
There is also a different approach to testing and deployment. Because onchain apps run on distributed systems, developers need to consider factors like network latency and the possibility of network partitions. They also need to ensure that their applications are extremely secure and resistant to a variety of attacks because the stakes can often be very high when it comes to dealing with millions and even billions of dollars of value that cannot be recovered in the event of a hack. Developers also have to consider concepts like immutability because once code is deployed to a blockchain, it cannot be edited.  
Overall, Web3 development requires a different set of engineering skills and tools as well as a deeper understanding of distributed systems and cryptography. Developers also need the ability to think creatively about how to build applications that are constrained by the technical limitations of the Web3 paradigm, such as speed, storage, and scalability.  
Conclusion  
Ethereum was created to extend the potential of blockchain technology beyond just a decentralized digital currency platform. Its ethos of innovation and experimentation has made a major impact on shaping the crypto ecosystem and has played a significant role in shaping the landscape of Web3. The use of smart contracts has enabled a wide range of new web applications, including DeFi, NFTs, DAOs, and many more.  
The evolution of the web has brought us from a static Web1 to a dynamic Web2, and now to a decentralized Web3. While Web2 brought many key benefits, it also came with many drawbacks regarding privacy, censorship, and security vulnerabilities. Web3 aims to address these challenges but has its own set of limitations. Lastly, the development paradigms of Web2 and Web3 are distinct in their architecture, infrastructure, and their development approaches. Web3 development requires a different set of skills and a different mental framework from its predecessor.  
See also  
Ethereum Whitepaper  
Decentralized Applications (Dapps)  
Web2 vs Web3  
The Architecture of a Web 3.0 Application  
ON THIS PAGE  
The Ethos and Goals of Ethereum  
Applications on Ethereum  
Evolution of the Web  
The Limitations of Web2  
The Limitations of Web3  
Web2 vs Web3 Development  
Decentralized Autonomous Organization (DAO)  
Other Applications  
Conclusion  
The Limitations of Web2  
The Limitations of Web3  
Web2 vs Web3 Development  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-ethereum/ethereum-dev-overview-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Introduction  
Ethereum Applications  
Ethereum Applications  
Gas Use in Ethereum Transactions  
EVM Diagram  
Guide to Base  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Ethereum Applications  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-ethereum/evm-diagram  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Introduction  
Ethereum Applications  
Ethereum Applications  
Gas Use in Ethereum Transactions  
EVM Diagram  
Guide to Base  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
EVM Diagram  
In this article, we'll examine the inner workings of the EVM, its components, and its role within the Ethereum network.  
Objectives  
By the end of this lesson you should be able to:  
Diagram the EVM  
What is the EVM?  
The Ethereum Virtual Machine (EVM) is the core engine of Ethereum. It is a Turing-complete, sandboxed virtual machine designed to execute smart contracts on the network. The term "sandboxed" means that the EVM operates in an isolated environment, ensuring that each smart contract's execution does not interfere with others or the underlying blockchain. As we've learned, the EVM's Turing-complete nature allows developers to write complex programs that can perform any computationally feasible task.  
The EVM employs a sophisticated resource management system using gas to regulate computation costs and prevent network abuse. It also supports a rich ecosystem of onchain apps by providing a versatile set of opcodes for smart contract logic, and fostering interoperability with various programming languages, tools, and technologies. This adaptability has made the EVM a fundamental component in the advancement and growth of the Ethereum network.  
EVM Components  
The EVM has several key components that enable it to process and manage smart contracts. Let's define them:  
World State: Represents the entire Ethereum network, including all accounts and their associated storage.  
Accounts: Entities that interact with the Ethereum network, including Externally Owned Accounts (EOAs) and Contract Accounts.  
Storage: A key-value store associated with each contract account, containing the contract's state and data.  
Gas: A mechanism for measuring the cost of executing operations in the EVM, which protects the network from spam and abuse.  
Opcodes: Low-level instructions that the EVM executes during smart contract processing.  
Execution Stack: A last-in, first-out (LIFO) data structure for temporarily storing values during opcode execution.  
Memory: A runtime memory used by smart contracts during execution.  
Program Counter: A register that keeps track of the position of the next opcode to be executed.  
Logs: Events emitted by smart contracts during execution, which can be used by external systems for monitoring or reacting to specific events.  
EVM Execution Model  
In simple terms, when a transaction is submitted to the network, the EVM first verifies its validity. If the transaction is deemed valid, the EVM establishes an execution context that incorporates the current state of the network and processes the smart contract's bytecode using opcodes. As the EVM runs the smart contract, it modifies the blockchain's world state and consumes gas accordingly. However, if the transaction is found to be invalid, it will be dismissed by the network without further processing. Throughout the smart contract's execution, logs are generated that provide insights into the contract's performance and any emitted events. These logs can be utilized by external systems for monitoring purposes or to respond to specific events.  
Gas and Opcode Execution  
While we have already delved into the concept of gas in a previous lesson, it is worth reiterating its critical role within the EVM and as a fundamental component of Ethereum. Gas functions as a metric for quantifying the computational effort needed to carry out operations in the EVM. Every opcode in a smart contract carries a specific gas cost, which reflects the computational resources necessary for its execution.  
Opcodes are the low-level instructions executed by the EVM. They represent elementary operations that allow the EVM to process and manage smart contracts.  
During execution, the EVM reads opcodes from the smart contract, and depending on the opcode, it may update the world state, consume gas, or revert the state if an error occurs. Some common opcodes include:  
ADD: Adds two values from the stack.  
SUB: Subtracts two values from the stack.  
MSTORE: Stores a value in memory.  
SSTORE: Stores a value in contract storage.  
CALL: Calls another contract or sends ether.  
Stack and Memory  
The EVM stack and memory are critical components of the EVM architecture, as they enable smart contracts to manage temporary data during opcode execution. The stack is a last-in, first-out (LIFO) data structure that is used for temporarily storing values during opcode execution. It is managed by the EVM and is separate from the contract's storage. The stack supports two primary operations: push and pop.  
The push operation adds a value to the top of the stack, while the pop operation removes the top value from the stack. These operations are used to manage temporary data during opcode execution. For example, an opcode that performs an addition operation might push the two operands onto the stack, perform the addition, and then pop the result off the top of the stack.  
During contract execution, memory serves as a collection of bytes, organized in an array, for the purpose of temporarily storing data. It can be read from and written to by opcodes. Memory is often used to store temporary data during opcode execution, such as when working with dynamically sized data like strings or arrays that are being manipulated or computed within the smart contract before being stored in the contract's storage. When a smart contract needs to store temporary data during opcode execution, it can use the memory to store that data.  
EVM Architecture and Execution Context  
To understand the inner workings of the EVM, the following diagram offers a streamlined visualization of its transaction execution process. It begins with the transaction initiation, and progresses to the gas computations for each operation. Integral to the process are the EVM's stack, memory, and storage, which are engaged to manage and persist data throughout the lifecycle of a transaction. Checks and validations at each step ensure the validity of operations, safeguarding the network's integrity. This systemized sequence of actions forms the bedrock of transaction and smart contract execution, ensuring Ethereum's consistent and secure operation.  
INFO  
Data Bytecode in the EVM  
Every transaction or smart contract call within the EVM uses "bytecode", which is a sequence of instructions that guides the EVM's actions. Bytecode is primarily presented in a compact hexadecimal format.  
Decoding the example sequence: 0x6080604052  
60 // PUSH1: Pushes the next byte (0x80) onto the stack.  
80 // The byte to be pushed onto the stack by the previous PUSH1.  
60 // PUSH1: Pushes the next byte (0x40) onto the stack.  
40 // The byte to be pushed onto the stack by the previous PUSH1.  
52 // MSTORE: Stores the second stack item in memory at the address of the first.  
This bytecode sequence is not a random set of characters. Each segment corresponds to specific operations or data in the EVM. Opcodes dictate actions, while subsequent data provides specifics.  
Conclusion  
The EVM plays a vital role within the Ethereum network. By examining the EVM's key components as well as its architecture and execution model, we've gained insight into the engine of Ethereum and how it enables the smooth execution of smart contracts on the platform.  
See Also  
The Ethereum Virtual Machine (Mastering Ethereum)  
Ethereum Virtual Machine (Ethereum docs)  
ON THIS PAGE  
What is the EVM?  
EVM Components  
EVM Execution Model  
Gas and Opcode Execution  
Stack and Memory  
EVM Architecture and Execution Context  
Conclusion  
See Also  
See Also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-ethereum/gas-use-in-eth-transactions  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Introduction  
Ethereum Applications  
Ethereum Applications  
Gas Use in Ethereum Transactions  
EVM Diagram  
Guide to Base  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Gas Use in Ethereum Transactions  
In this article, we'll delve into the concept of gas and its importance in the Ethereum ecosystem. You'll learn why Ethereum relies on a system of gas to regulate the execution of transactions and smart contracts, and how it plays a crucial role in the proper functioning of the network.  
Objectives  
By the end of this lesson you should be able to:  
Explain what gas is in Ethereum  
Explain why gas is necessary in Ethereum  
Understand how gas works in Ethereum transactions  
What is gas?  
Gas is a term used in Ethereum to describe a computational unit that measures the amount of computational work needed to perform specific operations on the network. Unlike Bitcoin, where transaction fees only consider the size of a transaction, Ethereum accounts for every computational step performed by transactions and smart contract code execution. In other words, every single operation that is performed on Ethereum requires a certain amount of gas.  
Complexity  
The amount of gas required for an operation depends on its complexity. More complex operations require more computational resources and therefore require more gas to be executed. For example, a simple transaction that involves sending ETH from one address to another may require less gas than a complex smart contract that executes multiple operations or interacts with multiple other contracts.  
State of the Network  
Gas costs can also vary depending on the state of the network, or more specifically, how congested it is. When there are more transactions waiting to be processed than the network can handle, it will prioritize transactions based on the gas price that was set by the user, meaning that higher gas prices are more likely to get processed first. When the network is congested, gas prices increase to encourage more efficient use of the network's resources and decrease when network usage is lower. This dynamic pricing mechanism ensures that the Ethereum network remains accessible and functional for all users, while also incentivizing responsible and efficient use of the network's resources.  
Why is gas necessary?  
Turing Completeness  
As we've learned, Ethereum is a Turing-complete platform, which means that any program that can be represented in code can theoretically be expressed and executed on the network. This opens up the door to countless different types of applications that can be built, but it also creates the possibility that malicious or inefficient code can clog up the network, potentially leading to denial-of-service attacks, network spam, and other problems.  
Preventing Infinite Loops  
Gas to the rescue! To prevent accidental or intentional infinite loops in smart contract code, Ethereum requires that every transaction specify a gas limit. The gas limit establishes the maximum amount of gas that the transaction can consume, and they ensure that transactions are executed within a predetermined amount of computational resources, preventing the execution of code that might consume too much computation power and potentially cause the network to freeze or crash. Without gas, Ethereum's Turing completeness would be insecure and inefficient.  
Autonomous Execution  
It's also important to note that gas enables the execution of smart contracts without the need for a central authority to monitor their execution. The gas system provides a mechanism for regulating the resources required to execute the code of these contracts as well. In other words, without gas, it would be difficult to guarantee that smart contracts could operate autonomously, fairly and efficiently.  
How does gas work?  
Ethereum Denominations  
Before diving into the inner workings of gas, it's important to understand a few of the most common denominations used in Ethereum.  
Ether (ETH)  
Ether is the native cryptocurrency of the Ethereum network. Gas fees are paid in ETH.  
Wei  
Wei is the smallest denomination of Ethereum and is equivalent to 10^-18 ETH. It is used to represent very small amounts of ETH, usually gas prices and transaction fees. To put 10^-18 into perspective:  
1 ETH = 1,000,000,000,000,000,000 wei  
1 wei = 0.000000000000000001 ETH  
Gwei  
Gwei is commonly used to express the price of gas. One gwei is equivalent to one billionth of one ETH or 10^-9 ETH.  
1 ETH = 1,000,000,000 gwei  
1 gwei = 0.000000001 ETH  
Gas Price  
Gas price on the network is denominated in gwei, and the gas fee is calculated as the product of the gas price and the amount of gas required for an operation. For example, if the gas price is 50 gwei, and an operation requires 100,000 units of gas, the gas fee would be 0.005 ETH (50 gwei x 100,000 gas = 0.005 ETH).  
Gas Limit  
Gas limit is an essential component of the gas system in Ethereum. It defines the maximum amount of gas a user is willing to spend for a transaction to be processed. This gas limit is set by the sender of the transaction and represents the upper limit of computational resources that the transaction can consume. The Ethereum Virtual Machine (EVM) starts deducting the amount of gas used from the gas limit as soon as it starts processing the transaction.  
Consider Alice wants to send some ETH to Bob. Alice specifies a gas limit of 100,000 units and a gas price of 10 gwei (0.00000001 ETH) per unit of gas. So, she's willing to spend a maximum of 0.001 ETH for this transaction (1,000,000 gwei).  
The EVM, upon receiving Alice's transaction, starts executing it. As the transaction is processed, the EVM deducts the used gas from the gas limit. If the transaction completes before reaching the gas limit, the remaining unused gas is refunded to Alice's account.  
Let's illustrate this with a couple scenarios:  
Suppose the transaction used 80,000 units of gas, leaving 20,000 units unused. Since the gas price was set at 10 gwei per unit, Alice would receive a refund of 0.0002 ETH (200,000 gwei) for the unused gas.  
In a different scenario, suppose Alice sends a transaction with a gas limit of 100,000 units. After processing all the opcodes in the transaction except for the last one, Alice's transaction has consumed 99,998 units of gas. The EVM checks and sees that the last opcode will initiate because there are 2 units of gas remaining, enough to start it. However, as the opcode executes, it becomes clear that it actually requires more than 2 units of gas. At this point, the EVM throws an "Out of Gas" exception and halts the transaction. In this scenario, Alice loses all 100,000 units of gas, as they are consumed in the attempted execution. All state changes that might have occurred during the execution are rolled back, and the ETH Alice tried to send to Bob is returned to her.  
Gas Estimation  
Gas estimation is another key concept to understand. It refers to the process of predicting the amount of gas that will be required to execute a transaction. This is important because as we've seen in our example, the gas limit of a transaction needs to be set before it can be broadcasted to the network. If the gas limit is set too low, the transaction may fail to execute, while if it is set too high, the sender may end up paying more in transaction fees than is necessary.  
There are several methods that can be used for gas estimation. One common method is to use historical gas prices and gas limits as a reference point, and to estimate the gas needed for a new transaction based on the gas used in similar past transactions. Another method is to simulate the execution of the transaction in a test environment to determine the actual amount of gas that would be used.  
Thankfully, most Ethereum wallet applications have built-in gas estimation algorithms that can automatically calculate an appropriate gas limit for a transaction based on the network conditions at the time the transaction is initiated. This helps to prevent a transaction from failing from the gas limit being too low while optimizing for the best possible cost for the sender.  
Conclusion  
Gas is a vital component of Ethereum. It's what regulates the execution of all transactions and smart contracts, and it plays a significant role in the proper functioning and security of the network. Without gas, Ethereum's Turing-complete architecture would be inefficient and vulnerable to attacks. Gas also ensures that smart contracts can operate autonomously, fairly, and efficiently without the need for a central authority to monitor their execution. Understanding how gas works is essential for anyone who wants to develop applications or smart contracts on the Ethereum network.  
See also  
Gas and Fees (Ethereum Docs)  
Transaction Gas (Mastering Ethereum)  
Turing Completeness and Gas (Mastering Ethereum)  
Gas (Mastering Ethereum)  
ON THIS PAGE  
What is gas?  
Why is gas necessary?  
Turing Completeness  
Preventing Infinite Loops  
How does gas work?  
Ethereum Denominations  
Gas Price  
Gas Limit  
Conclusion  
Ethereum Denominations  
Gas Price  
Gas Limit  
Gas Estimation  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-ethereum/intro-to-ethereum-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Introduction  
Ethereum Applications  
Ethereum Applications  
Gas Use in Ethereum Transactions  
EVM Diagram  
Guide to Base  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Introduction  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/anatomy-of-a-smart-contract-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Anatomy of a Smart Contract  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/deployment-in-remix  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Introduction  
Solidity Overview  
Introduction to Remix  
Introduction to Remix  
Deployment in Remix  
Deployment in Remix  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Deployment in Remix  
Remix contains a simulation of the blockchain that allows you to easily and safely deploy and interact with contracts, for free.  
Objectives  
By the end of this lesson you should be able to:  
Deploy and test the Storage.sol demo contract in Remix  
Deploy the Storage Contract  
Deploying a contract is easy, but remember that if the contract doesn't compile, the deploy button will instead deploy the last good compiled version of your contract. Verify that you see a green checkmark on the icon for the Compiler contract on the left side of the editor, then select the Deploy & Run Transactions plugin.  
If you've already deployed any contracts, press the trash can button to the right of the Deployed Contracts label. Then, press the orange button to deploy the Storage contract.  
Contract Addresses  
After your contract deploys, it will appear in the Deployed contracts section as STORAGE AT followed by an address. Addresses are used for both contracts and wallets in EVM-compatible blockchains and serve a similar purpose to an IP address. You can copy the address to see what it looks like. It's 20 characters of hexadecimal, similar to 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8.  
The address is what you will use to find your contract with tools such as Etherscan, or to connect to it with a front end.  
However, when you deploy using the Remix VM simulation, it will only exist in your browser.  
Deployments and Test Accounts  
The result of any transactions, including deployments, will appear in the Remix terminal. Click the chevron next to the blue Debug button to expand the log.  
Doing so will show the full transaction log, which contains all of the details of the transaction, such as its amount, the address to and from, and the inputs and outputs provided to the transaction.  
In this case, the sender (from) matches the first listed account in the panel, which has spent a small amount of simulated Ether to deploy the contract.  
You can access a list of 15 test wallets here, each with 100 Ether to spend. Among other uses, you can use these accounts to compare behavior between wallets that are and are not the owner of a deployed contract.  
Interacting with the Contract  
Click the chevron to expand your contract in the Deployed Contracts section of the left panel. You'll see two buttons, one for each public function in the Storage contract. Notice how the Store button also has a field to pass a uint256, matching the parameter for uint256 num.  
Let's click the retrieve button first. Before clicking, make a prediction: given that the number variable was instantiated without a value, what do you thing the return will be?  
Go ahead and click – the result will appear below the button as:  
0: uint256: 0  
Outputs from the EVM are in the form of an array, so in this case, the only return is in the 0th element and it is a uint256 of 0. Were you expecting undefined or an error?  
Unlike many languages, variables in Solidity have a default value if not assigned. For uint and int, that value is 0.  
You can also review the results of your transaction in the console.  
The screenshot above is from a newer version of Remix than the video. Outputs are now often decoded for you!  
Storing and Retrieving a Value  
Use the input to store and retrieve a value. Which costs more gas? Storing or retrieving? This isn't a trick question, but it is a bit nuanced. Both cost about 23500 gas, but there is only a gas cost for the retrieve function if it is called by another contract. Calling it from the web is free, because you're only reading data that is on the blockchain and are not asking the EVM to perform a computational task.  
Disabling Artifact Generation  
Return to the File Explorer by clicking the double document icon in the upper left. You should now see a folder called artifacts that has been added to your project. This folder contains a number of build artifacts, such as the ABI for your contract, that will be useful to you later, but currently just cause clutter.  
You can disable artifact generation by clicking the settings gear in the bottom left corner, then deselecting the first checkbox to Generate contract metadata...  
Conclusion  
Remix makes it easy to write, deploy, and test contracts. Contracts are deployed by a wallet address to their own address. These addresses are similar to how IP addresses work, in that they enable connections across the network. You can test deployed contracts directly in Remix and use the console to see detailed information about each transaction.  
ON THIS PAGE  
Deploy the Storage Contract  
Disabling Artifact Generation  
Contract Addresses  
Deployments and Test Accounts  
Interacting with the Contract  
Storing and Retrieving a Value  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/deployment-in-remix-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Introduction  
Solidity Overview  
Introduction to Remix  
Introduction to Remix  
Deployment in Remix  
Deployment in Remix  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Deployment in Remix  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/introduction-to-remix  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Introduction  
Solidity Overview  
Introduction to Remix  
Introduction to Remix  
Deployment in Remix  
Deployment in Remix  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Introduction to Remix  
In this lesson, you'll be introduced to an online Solidity IDE called Remix. You'll tour the workspace and explore a sample smart contract.  
Objectives  
By the end of this lesson you should be able to:  
List the features, pros, and cons of using Remix as an IDE  
Deploy and test the Storage.sol demo contract in Remix  
Remix Window Overview  
Begin by opening a browser window and navigating to remix.ethereum.org. Open the project you created and cleaned up at the end of the last reading, and open 1\_Storage.sol. The editor should be organized in a way that is familiar to you. It is divided into three areas:  
Editor Pane  
Terminal/Output  
Left Panel  
Editor Pane  
The editor pane loads with the Remix home screen, which contains news, helpful links, and warnings about common scams. Double-click on 1\_Storage.sol to open it in the editor. You can close the home tab if you'd like.  
You'll edit your code in the editor pane. It also has most of the features you're expecting, such as syntax and error highlighting. Note that in Remix, errors are not underlines. Instead, you'll see an❗to the left of the line number where the error is present.  
At the top, you'll see a large green arrow similar to the Run button in other editors. In Solidity, this compiles your code, but it does not run it because you must first deploy your code to the simulated blockchain.  
Terminal/Output  
Below the editor pane, you'll find the terminal:  
You'll primarily use this panel to observe transaction logs from your smart contracts. It's also one way to access Remix's very powerful debugging tools.  
Left Panel  
As with many other editors, the left panel in Remix has a number of vertical tabs that allow you to switch between different tools and functions. You can explore the files in your current workspace, create and switch between workspaces, search your code, and access a number of plugins.  
Plugins  
Most of the features in Remix are plugins and the ones you'll use the most are active by default. You can view and manage plugins by clicking the plug button in the lower-left corner, right above the settings gear. You can turn them off and on by clicking activate/deactivate, and some, such as the Debug plugin will be automatically activated through other parts of the editor.  
Solidity Compiler  
The first default plugin (after the search function) is the Solidity Compiler. Be sure to check the Auto compile option. Smart contracts are almost always very small files, so this shouldn't ever cause a performance problem while editing code.  
The Compile and Run script button in this plugin is a little misleading. This is not how you will usually run your contract through testing. You can click the I button for more information on this feature.  
Finally, if you have errors in your contracts, the complete text for each error will appear at the bottom of the pane. Try it out by introducing some typos to 1\_Storage.sol.  
Deploy & Run Transactions  
The Deploy & Run Transactions plugin is what you'll use to deploy your contracts and then interact with them. At the top are controls to select which virtual machine to use, mock user wallets with test Ether, and a drop-down menu to select the contract you wish to deploy and test.  
Fix any errors you introduced to 1\_Storage.sol and then click the orange Deploy button. You'll see your contract appear below as STORAGE AT \<address>.  
CAUTION  
There are two common gotchas that can be very confusing when deploying contracts in Remix.  
Each time you hit the Deploy button, a new copy of your contract is deployed but the previous deployments remain. Unless you are comparing or debugging between different versions of a contract, or deploying multiple contracts at once, you should click the Trash button to erase old deployments before deploying again.  
If your code will not compile, clicking the deploy button will not generate an error! Instead, the last compiled version will be deployed. Visually check and confirm that there are no errors indicated by a number in a red circle on top of the Compiler plugin.  
Conclusion  
Remix is a robust editor with many features and one or two gotchas. It is an excellent tool to use at the beginning of your journey because you can jump right in and start writing code for smart contracts.  
See also  
Remix  
ON THIS PAGE  
Remix Window Overview  
Plugins  
Solidity Compiler  
Deploy & Run Transactions  
Left Panel  
Conclusion  
Solidity Compiler  
Deploy & Run Transactions  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/introduction-to-remix-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Introduction  
Solidity Overview  
Introduction to Remix  
Introduction to Remix  
Deployment in Remix  
Deployment in Remix  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Introduction to Remix  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/introduction-to-solidity-overview  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Overview  
The course you are about to begin is designed to rapidly and thoroughly teach web3 concepts and language to web2 developers. It specifically highlights similarities and differences found in web3 vs. web2 and contains background information, guided coding practices, and independent exercises.  
This program is not suitable for people who are new to programming in general. While the explanations are thorough, they often rely on an expectation that you are familiar with the underlying concepts. We will not teach you what arrays are and how they are used, but we will show you how they work in this environment.  
Prerequisites  
Before these lessons, you should:  
Have several years of experience as a programmer in an object-oriented language  
Be familiar with the uses and properties of the Ethereum blockchain and the EVM  
Ideally, be familiar with at least one curly-bracket programming language  
Objectives  
By the end of this module, you should be able to:  
Introduction to Solidity  
Describe why languages like Solidity are used to write smart contracts  
Relate an overview of the history (and pace of change) of Solidity and its strengths and weaknesses  
Deploy and test the Storage.sol demo contract in Remix  
Contracts and Basic Functions  
Construct a simple ""Hello World"" contract  
Categorize basic data types  
List the major differences between data types in Solidity as compared to other languages  
Compare and contrast signed and unsigned integers  
Write a pure function that accepts argument and returns a value  
Deploying Smart Contracts to a Testnet  
Describe the uses and properties of the Ethereum testnet  
Compare and contrast Ropsten, Rinkeby, Goerli, and Sepolia  
Deploy a contract to the Sepolia testnet and interact with it in Etherscan  
Control Structures  
Control code flow with if, else, while, and for  
List the unique constraints for control flow in Solidity  
Storage in Solidity  
Diagram how a contract's data is stored on the blockchain (Contract -> Blockchain)  
Order variable declarations to use storage efficiently  
Diagram how variables in a contract are stored (Variable -> Contract)  
Arrays in Solidity  
Construct then store and retrieve values in storage and memory arrays  
Describe the difference between storage and memory arrays  
Diagram how arrays are stored  
Write a function that can return a filtered subset of an array  
The Mapping Type  
Construct a Map (dictionary) data type  
Diagram the storage of the Mapping data type  
Recall that assignment of the Map data type is not as flexible as for other data types/in other languages  
Restrict function calls with the msg.sender global variable  
Recall that there is no collision protection in the EVM and why this (probably) ok  
Advanced Functions  
Describe how pure and view functions are different than functions that modify storage  
Categorize functions as public, private, internal, or external based on their usage  
Use modifiers to efficiently add functionality to multiple functions  
Utilize require to write a function that can only be used when a variable is set to 'True'  
Structs  
Construct a struct (user-defined type) that contains several different data types  
Declare members of the struct to maximize storage efficiency  
Describe constraints related to assignment of structs depending on the types they contain  
Inheritance  
Write a smart contract that inherits from another contract  
Imports  
Import and use code from another file  
Errors  
Debug common solidity errors including execution reverted, out of gas, stack overflow, value overflow/underflow, index out of range, and so on  
New Keyword  
Write a contract that creates a new contract with the new keyword  
ON THIS PAGE  
Objectives  
Objectives  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/introduction-to-solidity-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Introduction  
Solidity Overview  
Introduction to Remix  
Introduction to Remix  
Deployment in Remix  
Deployment in Remix  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Introduction  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/introduction-to-solidity/solidity-overview  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Introduction  
Solidity Overview  
Introduction to Remix  
Introduction to Remix  
Deployment in Remix  
Deployment in Remix  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Solidity Overview  
In this article, you'll learn about the origins and history of Solidity, where to find the docs, and review some of the considerations that make programming in Solidity relatively unique. You'll also learn about how to get started with development!  
Objectives  
By the end of this lesson you should be able to:  
Describe why languages like Solidity are used to write smart contracts  
Relate an overview of the history (and pace of change) of Solidity and its strengths and weaknesses  
Introduction to Solidity  
Solidity is a high-level language used to develop smart contracts compatible with the Ethereum Virtual Machine. It is object-oriented, strongly typed, and allows variadic (more than one argument) returns. Solidity was inspired by a number of languages, particularly C++. Compared to other languages, Solidity changes very rapidly. Review the releases to see just how rapid!  
The Docs  
The Solidity Docs are thorough and helpful. This guide will regularly reference them and they should be your first source for specific information related to any of the components in the language. As with any versioned doc source, always double-check that the version you're referencing matches the version you are developing with.  
Origins TL;DR  
Solidity was developed by the Ethereum Project's Solidity team and was first previewed in 2014 at DevCon0. The original goal was to create an easy-to-use language for smart contract development. A great history overview can be found in the team's blog.  
What it Actually Does  
Solidity is very similar to the programming languages you are familiar with in that it's a high-level language that is relatively human-readable, which is then compiled into byte-code that can be read by the EVM. For example, this:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.9;  
  
contract Hello {  
 function HelloWorld() public pure returns (string memory) {  
 return "Hello World!";  
 }  
}  
compiles into this:  
  
As you can see, the first example is a little easier to read!  
Programming for Ethereum with Solidity  
On the surface, writing code for the EVM using Solidity isn't particularly different from other programming languages. You write code organized into functions, and those functions get executed when called, often accepting arguments and returning values. However, there are a number of unusual traits that will require you to think a little differently. Additionally, the EVM is a much smaller, slower, and less-powerful computer than a desktop, or even a mobile device.  
Gas Fees  
Every single operation your code performs costs gas, which your users pay for. You're probably already well-versed in time complexity and know how to get an operation down to O(log(n)), when you have no choice but to run something that is O(2^n), and that sometimes, nested for-loops go brrrrr. These constraints and practices still apply, but in Solidity, every inefficiency directly costs your users money, which can make your app more expensive, and less appealing, than needed.  
When you were learning about time complexity, you probably heard the term space complexity once, and then it was never mentioned again. This is because normally, computation is expensive, and storage is practically free. The opposite is true on the EVM. It costs a minimum of 20,000 gas to initialize a variable, and a minimum of 5,000 to change it. Meanwhile, the cost to add two numbers together is 3 gas. This means it is often much cheaper to repeatedly derive a value that is calculated from other values than it is to calculate it once and save it.  
You also have to be careful to write code with predictable execution paths. Each transaction is sent with a gas limit and which various frameworks, such as ethers.js, in order to do their best to estimate. If this estimate is wrong, the transaction will fail, but it will still cost the gas used up until the point it failed!  
Contract Size Limit  
EIP-170 introduced a compiled byte-code size limit of 24 KiB (24,576 B) to Ethereum Smart Contracts. Read that sentence again, as you're probably not used to thinking in this small of a number!  
While there isn't an exact ratio of lines of code to compiled byte-code size, you're limited to deploying contracts that are approximately 300-500 lines of Solidity.  
Luckily, there are a few ways around this limitation. Contracts can expose their functions to be called by other contracts, although there is an additional cost. Using this, you can write a suite of contracts designed to work together, or even make use of contracts already deployed by others. You can also use more advanced solutions, such as EIP-2535.  
Stack Limit  
Programs written for computers or mobile devices often work with hundreds of variables at the same time. The EVM operates with a stack that can hold 1,024 values, but it can only access the top 16.  
There are many implications of this limit, but the one you'll run into most commonly is the "Stack too Deep" error because you're trying to work with too many variables at once.  
In Solidity/EVM, your functions are limited to a total of 16 variables that are input, output, or initialized by the function.  
Permanence  
Once deployed, smart contracts are permanent and cannot be changed by anyone, even their creator(s)! It is literally not possible to edit them. If the creators of a contract discover a vulnerability, they can't do anything about it except withdraw the funds - if the contract allows them to!  
As a result, standard practice is to have a smart contract audited by an expert, before deployment.  
Pace of Change  
Solidity files always start with a license and a version:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.17;  
One of the reasons for this is that the pace of development for Solidity is very fast, and changes are not always backwards-compatible. As a result, the compiler needs to know which version to use when converting the Solidity code to byte-code.  
Review the changelog to see some of the recent additions and fixes.  
Development Environments  
We'll be covering two tools that can be used to develop in Solidity.  
Remix  
We'll start with Remix, an online IDE similar to Codepen, Replit, or CodeSandbox. Remix is a great place to get started because it works out of the box, has a number of demo contracts, and has great debugging tools. More information can be found at the Remix Project website.  
DANGER  
BE VERY CAREFUL while using Remix, as it can also be used by scammers. Remix itself will warn you about this, so take heed! One common scam is for the scammer to convince you to paste and deploy code that is allegedly some sort of automated moneymaker, such as a staking tool, or a bot.  
If you paste and run code that you don't understand, you may lose all assets from your currently connected wallet. You should also be careful to always navigate directly to remix.ethereum.org. More experienced developers prefer to use static versions of Remix deployed to IPFS, but be careful. There are also deployments that are compromised and used as a part of a scam!  
Hardhat  
Hardhat is a development environment that allows you to develop and test Solidity on your local machine. It includes debugging and unit testing tools, and has an ecosystem of third-party-developed plugins that ease development and deployment. Among other things, these plugins can help you deploy contracts, see the size of your compiled byte-code, and even see unit test coverage.  
We'll introduce Hardhat and local development after the basics.  
Remix Setup  
The next lesson will explore one of the demo contracts within Remix. Open it up and review the quickstart information if this is your first time on the site. Then, open or create a new workspace using the Default template.  
Delete everything except the contracts folder and the 1\_Storage.sol contract within that folder. You can also leave .prettierrc.json if you'd like.  
Conclusion  
On the surface, Solidity is very similar to other programming languages; most developers won't struggle to write familiar operations. However, there are some critically important properties to keep in mind. Operations are much more expensive than in other environments, particularly storage. You can use most of the practices you are accustomed to, but you are limited to very small contract sizes and by the size of the stack. Finally, remember that you should always use a separate wallet for development. If you make a mistake, you could lose anything in it!  
See also  
Solidity Docs  
ON THIS PAGE  
Introduction to Solidity  
Programming for Ethereum with Solidity  
Gas Fees  
Contract Size Limit  
Stack Limit  
Development Environments  
Remix  
Hardhat  
Stack Limit  
Permanence  
Pace of Change  
Remix Setup  
Remix  
Hardhat  
Conclusion  
See also  
See also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/learning-objectives  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
learning-objectives  
Ethereum Applications  
Describe the origin and goals of the Ethereum blockchain  
List common types of applications that can be developed with the Ethereum blockchain  
Compare and contrast Web2 vs. Web3 development  
Compare and contrast the concept of "ownership" in Web2 vs. Web3  
Gas Use in Ethereum Transactions  
Explain what gas is in Ethereum  
Explain why gas is necessary in Ethereum  
Understand how gas works in Ethereum transactions  
EVM Diagram  
Diagram the EVM  
Setup and Overview  
Install and create a new Hardhat project with Typescript support  
Describe the organization and folder structure of a Hardhat project  
List the use and properties of hardhat.config.ts  
Testing with Hardhat and Typechain  
Set up TypeChain to enable testing  
Write unit tests for smart contracts using Mocha, Chai, and the Hardhat Toolkit  
Set up multiple signers and call smart contract functions with different signers  
Etherscan  
List some of the features of Etherscan  
Read data from the Bored Apes Yacht Club contract on Etherscan  
Write data to a contract using Etherscan.  
Deploying Smart Contracts  
Deploy a smart contract to the Base Sepolia Testnet with hardhat-deploy  
Deploy a smart contract to the Sepolia Testnet with hardhat-deploy  
Use BaseScan to view a deployed smart contract  
Verifying Smart Contracts  
Verify a deployed smart contract on Etherscan  
Connect a wallet to a contract in Etherscan  
Use etherscan to interact with your own deployed contract  
Hardhat Forking  
Use Hardhat Network to create a local fork of mainnet and deploy a contract to it  
Utilize Hardhat forking features to configure the fork for several use cases  
'Introduction to Remix'  
List the features, pros, and cons of using Remix as an IDE  
Deploy and test the Storage.sol demo contract in Remix  
Deployment in Remix  
Deploy and test the Storage.sol demo contract in Remix  
Hello World  
Construct a simple "Hello World" contract  
List the major differences between data types in Solidity as compared to other languages  
Select the appropriate visibility for a function  
Basic Types  
Categorize basic data types  
List the major differences between data types in Solidity as compared to other languages  
Compare and contrast signed and unsigned integers  
Test Networks  
Describe the uses and properties of the Base testnet  
Compare and contrast Ropsten, Rinkeby, Goerli, and Sepolia  
Deployment to Base Sepolia  
Deploy a contract to the Base Sepolia testnet and interact with it in [BaseScan]  
Contract Verification  
Verify a contract on the Base Sepolia testnet and interact with it in [BaseScan]  
Control Structures  
Control code flow with if, else, while, and for  
List the unique constraints for control flow in Solidity  
Utilize require to write a function that can only be used when a variable is set to true  
Write a revert statement to abort execution of a function in a specific state  
Utilize error to control flow more efficiently than with require  
Storing Data  
Use the constructor to initialize a variable  
Access the data in a public variable with the automatically generated getter  
Order variable declarations to use storage efficiently  
How Storage Works  
Diagram how a contract's data is stored on the blockchain (Contract -> Blockchain)  
Order variable declarations to use storage efficiently  
Diagram how variables in a contract are stored (Variable -> Contract)  
Arrays  
Describe the difference between storage, memory, and calldata arrays  
Filtering an Array  
Write a function that can return a filtered subset of an array  
Mappings  
Construct a Map (dictionary) data type  
Recall that assignment of the Map data type is not as flexible as for other data types/in other languages  
Restrict function calls with the msg.sender global variable  
Recall that there is no collision protection in the EVM and why this is (probably) ok  
Function Visibility and State Mutability  
Categorize functions as public, private, internal, or external based on their usage  
Describe how pure and view functions are different than functions that modify storage  
Function Modifiers  
Use modifiers to efficiently add functionality to multiple functions  
Structs  
Construct a struct (user-defined type) that contains several different data types  
Declare members of the struct to maximize storage efficiency  
Describe constraints related to the assignment of structs depending on the types they contain  
Inheritance  
Write a smart contract that inherits from another contract  
Describe the impact inheritance has on the byte code size limit  
Multiple Inheritance  
Write a smart contract that inherits from multiple contracts  
Abstract Contracts  
Use the virtual, override, and abstract keywords to create and use an abstract contract  
Imports  
Import and use code from another file  
Utilize OpenZeppelin contracts within Remix  
Error Triage  
Debug common solidity errors including transaction reverted, out of gas, stack overflow, value overflow/underflow, index out of range, etc.  
The New Keyword  
Write a contract that creates a new contract with the new keyword  
'Contract to Contract Interaction'  
Use interfaces to allow a smart contract to call functions in another smart contract  
Use the call() function to interact with another contract without using an interface  
Events  
Write and trigger an event  
List common uses of events  
Understand events vs. smart contract storage  
Address and Payable in Solidity  
Differentiate between address and address payable types in Solidity  
Determine when to use each type appropriately in contract development  
Employ address payable to send Ether and interact with payable functions  
Minimal Token  
Construct a minimal token and deploy to testnet  
Identify the properties that make a token a token  
The ERC-20 Token Standard  
Analyze the anatomy of an ERC-20 token  
Review the formal specification for ERC-20  
ERC-20 Implementation  
Describe OpenZepplin  
Import the OpenZepplin ERC-20 implementation  
Describe the difference between the ERC-20 standard and OpenZeppelin's ERC20.sol  
Build and deploy an ERC-20 compliant token  
The ERC-721 Token Standard  
Analyze the anatomy of an ERC-721 token  
Compare and contrast the technical specifications of ERC-20 and ERC-721  
Review the formal specification for ERC-721  
ERC-721 Token  
Analyze the anatomy of an ERC-721 token  
Compare and contrast the technical specifications of ERC-20 and ERC-721  
Review the formal specification for ERC-721  
Build and deploy an ERC-721 compliant token  
Use an ERC-721 token to control ownership of another data structure  
Wallet Connectors  
Identify the role of a wallet aggregator in an onchain app  
Debate the pros and cons of using a template  
Scaffold a new onchain app with RainbowKit  
Support users of EOAs and the Coinbase Smart Wallet with the same app  
Building an Onchain App  
Identify the role of a wallet aggregator in an onchain app  
Debate the pros and cons of using a template  
Add a wallet connection to a standard template app  
The useAccount Hook  
Implement the `useAccount`` hook to show the user's address, connection state, network, and balance  
Implement an isMounted hook to prevent hydration errors  
The useReadContract Hook  
Implement wagmi's useReadContract hook to fetch data from a smart contract  
Convert data fetched from a smart contract to information displayed to the user  
Identify the caveats of reading data from automatically-generated getters  
Configuring useReadContract  
Use useBlockNumber and the queryClient to automatically fetch updates from the blockchain  
Describe the costs of using the above, and methods to reduce those costs  
Configure arguments to be passed with a call to a pure or view smart contract function  
Call an instance of useReadContract on demand  
Utilize isLoading and isFetching to improve user experience  
The useWriteContract hook  
Implement wagmi's useWriteContract hook to send transactions to a smart contract  
Configure the options in useWriteContract  
Display the execution, success, or failure of a function with button state changes, and data display  
The useSimulateContract hook  
Implement wagmi's useSimulateContract and useWriteContract to send transactions to a smart contract  
Configure the options in useSimulateContract and useWriteContract  
Call a smart contract function on-demand using the write function from useWriteContract, with arguments and a value  
ON THIS PAGE  
Gas Use in Ethereum Transactions  
EVM Diagram  
Setup and Overview  
Testing with Hardhat and Typechain  
Etherscan  
Deploying Smart Contracts  
Verifying Smart Contracts  
Hardhat Forking  
'Introduction to Remix'  
Deployment in Remix  
Hello World  
Basic Types  
Test Networks  
Deployment to Base Sepolia  
Contract Verification  
Control Structures  
Storing Data  
How Storage Works  
Arrays  
Filtering an Array  
Mappings  
Function Visibility and State Mutability  
Function Modifiers  
Structs  
Inheritance  
Multiple Inheritance  
Abstract Contracts  
Imports  
Error Triage  
The New Keyword  
'Contract to Contract Interaction'  
Events  
Address and Payable in Solidity  
Minimal Token  
The ERC-20 Token Standard  
ERC-20 Implementation  
The ERC-721 Token Standard  
ERC-721 Token  
Wallet Connectors  
Building an Onchain App  
The useAccount Hook  
The useReadContract Hook  
Configuring useReadContract  
The useWriteContract hook  
The useSimulateContract hook  
The useSimulateContract hook  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/mappings/how-mappings-are-stored-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Mappings  
Using `msg.sender`  
Mappings  
How Mappings are Stored  
Mappings Exercise  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
How Mappings are Stored  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/mappings/mappings-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Mappings  
Using `msg.sender`  
Mappings  
How Mappings are Stored  
Mappings Exercise  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Mappings Exercise  
Create a contract that adheres to the following specifications.  
Contract  
Create a single contract called FavoriteRecords. It should not inherit from any other contracts. It should have the following properties:  
State Variables  
The contract should have the following state variables. It is up to you to decide if any supporting variables are useful.  
A public mapping approvedRecords, which returns true if an album name has been added as described below, and false if it has not  
A mapping called userFavorites that indexes user addresses to a mapping of string record names which returns true or false, depending if the user has marked that album as a favorite  
Loading Approved Albums  
Using the method of your choice, load approvedRecords with the following:  
Thriller  
Back in Black  
The Bodyguard  
The Dark Side of the Moon  
Their Greatest Hits (1971-1975)  
Hotel California  
Come On Over  
Rumours  
Saturday Night Fever  
Get Approved Records  
Add a function called getApprovedRecords. This function should return a list of all of the names currently indexed in approvedRecords.  
Add Record to Favorites  
Create a function called addRecord that accepts an album name as a parameter. If the album is on the approved list, add it to the list under the address of the sender. Otherwise, reject it with a custom error of NotApproved with the submitted name as an argument.  
Users' Lists  
Write a function called getUserFavorites that retrieves the list of favorites for a provided address memory.  
Reset My Favorites  
Add a function called resetUserFavorites that resets userFavorites for the sender.  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Contract  
State Variables  
Loading Approved Albums  
Get Approved Records  
Add Record to Favorites  
Users' Lists  
Reset My Favorites  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/mappings/mappings-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Mappings  
Using `msg.sender`  
Mappings  
How Mappings are Stored  
Mappings Exercise  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Mappings  
In Solidity, the hashtable/hashmap/dictionary-comparable type used to store key-value pairs is called a mapping. mappings are a powerful tool with many uses, but they also have some unexpected limitations. They also aren't actually hash tables!  
Objectives  
By the end of this lesson you should be able to:  
Construct a Map (dictionary) data type  
Recall that assignment of the Map data type is not as flexible as for other data types/in other languages  
Restrict function calls with the msg.sender global variable  
Recall that there is no collision protection in the EVM and why this is (probably) ok  
Mappings in Solidity vs. Hash Tables  
On the surface, the mapping data type appears to be just another hash table implementation that stores pairs of any hashable type as a key, to any other type as a value. The difference is in implementation.  
In a more traditional implementation, the data is stored in memory as an array, with a hash-to-index (hashmod) function used to determine which spot in the array to store a given value, based on the key. Sometimes, the hashmod function for two different keys results in the same index, causing a collision.  
Collisions are resolved via additional solutions, such as linked list chaining; when the underlying array starts to get full, a bigger one is created, all the keys are re-hash-modded, and all the values moved over to the new array.  
In the EVM, mappings do not have an array as the underlying data structure. Instead, the keccak256 hash of the key plus the storage slot for the mapping itself is used to determine which storage slot out of all 2\*\*256 will be used for the value.  
There is no collision-handling, for the same reason that makes wallets work at all - 2\*\*256 is an unimaginably large number. One of the biggest numbers you might encounter regularly is the number of possible configurations for a shuffled deck of cards, which is:  
80658175170943878571660636856403766975289505440883277824000000000000  
Meanwhile, the number of variations of a keccak256 hash are:  
115792089237316195423570985008687907853269984665640564039457584007913129639935  
Collisions are very unlikely.  
As a result, there are a few special characteristics and limitations to keep in mind with the mapping data type:  
Mappings can only have a data location of storage  
They can't be used as parameters or returns of public functions  
They are not iterable and you cannot retrieve a list of keys  
All possible keys will return the default value, unless another value has been stored  
Creating a Mapping  
Create a contract called Mappings. In it, add a mapping from an address to a uint called favoriteNumbers.  
Reveal code  
  
  
  
  
Writing to the Mapping  
Add a function called saveFavoriteNumber that takes an address and uint, then saves the uint in the mapping, with the address as the key.  
Reveal code  
  
  
  
  
Deploy and test it out. Does it work? Probably...  
You don't have a way to read the data in favoriteNumber, but this problem is easy to correct. Similar to arrays, if you mark a mapping as public, the Solidity compiler will automatically create a getter for values in that mapping.  
Update the declaration of favoriteNumbers and deploy to test again.  
Utilizing msg.sender  
Another issue with this contract is that a public function can be called by anyone and everyone with a wallet and funds to pay gas fees. As a result, anyone could go in after you and change your favorite number from lucky number 13 to anything, even 7!  
That won't do at all!  
Luckily, you can make use of a global variable called msg.sender to access the address of the wallet that sent the transaction. Use this to make it so that only the owner of an address can set their favorite number.  
Reveal code  
  
  
  
  
Deploy and test again. Success!  
Retrieving All Favorite Numbers  
One challenging limitation of the mapping data type is that it is not iterable - you cannot loop through and manipulate or return all values in the mapping.  
At least not with any built in features, but you can solve this on your own. A common practice in Solidity with this and similar problems is to use multiple variables or data types to store the right combination needed to address the issue.  
Using a Helper Array  
For this problem, you can use a helper array to store a list of all the keys present in favoriteNumbers. Simply add the array, and push new keys to it when saving a new favorite number.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
To return all of the favorite numbers, you can then iterate through addressesOfFavs, look up that addresses' favorite number, add it to a return array, and then return the array when you're done.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
On the surface, this solution works, but there is a problem: What happens if a user updates their favorite number? Their address will end up in the list twice, resulting in a doubled entry in the return.  
A solution here would be to check first if the address already has a number as a value in favoriteNumbers, and only push it to the array if not.  
Reveal code  
  
  
  
  
  
  
  
You should end up with a contract similar to this:  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Conclusion  
In this lesson, you've learned how to use the mapping data type to store key-value pairs in Solidity. You've also explored one strategy for solving some of the limitations found in the mapping type when compared to similar types in other languages.  
ON THIS PAGE  
Mappings in Solidity vs. Hash Tables  
Retrieving All Favorite Numbers  
Using a Helper Array  
Writing to the Mapping  
Utilizing msg.sender  
Conclusion  
Using a Helper Array  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/mappings/mappings-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Mappings  
Using `msg.sender`  
Mappings  
How Mappings are Stored  
Mappings Exercise  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Mappings  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/mappings/using-msg-sender-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Mappings  
Using `msg.sender`  
Mappings  
How Mappings are Stored  
Mappings Exercise  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Using `msg.sender`  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/minimal-tokens/creating-a-minimal-token-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
Create a Minimal Token  
Transferring a Minimal Token  
Minimal Token  
Minimal Tokens Exercise  
ERC-20 Tokens  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Create a Minimal Token  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/minimal-tokens/minimal-token-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
Create a Minimal Token  
Transferring a Minimal Token  
Minimal Token  
Minimal Tokens Exercise  
ERC-20 Tokens  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Minimal Token  
At their core, tokens are very simple. The technology powering famous NFT collections and fungible tokens worth vast amounts of money simply uses the EVM to keep track of who owns what, and provides a permissionless way for the owner to transfer what they own to someone new.  
Objectives  
By the end of this lesson you should be able to:  
Construct a minimal token and deploy to testnet  
Identify the properties that make a token a token  
Implementing a Token  
The minimal elements needed for a token are pretty basic. Start by creating a contract called MinimalToken. Add a mapping to relate user addresses to the number of tokens they possess. Finally, add a variable to track totalSupply:  
Reveal code  
  
  
  
  
  
Add a constructor that initializes the totalSupply at 3000 and assigns ownership to the contract creator:  
Reveal code  
  
  
  
  
  
  
Deploy and test to confirm that the total supply is 3000, and the balance of the first account is as well.  
Update the constructor and hardcode a distribution of the tokens to be evenly split between the first three test accounts:  
Reveal code  
  
  
  
  
  
  
  
  
Redeploy and test again. Now, each of the first three accounts should have 1000 tokens.  
Transferring Tokens  
We can set an initial distribution of tokens and we can see balances, but we're still missing a way to allow the owners of these tokens to share them or spend them.  
To remediate this, all we need to do is add a function that can update the balances of each party in the transfer.  
Add a function called transfer that accepts an address of \_to and a uint for the \_amount. You don't need to add anything for \_from, because that should only be msg.sender. The function should subtract the \_amount from the msg.sender and add it to \_to:  
Reveal code  
  
  
  
  
  
Double-check that you've switched back to the first address and redeploy. Then, try sending 500 tokens to the second address.  
What happens if you try to transfer more tokens than an account has? Give it a try!  
transact to MinimalToken.transfer pending ...  
transact to MinimalToken.transfer errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
Note: The called function should be payable if you send value and the value you send should be less than your current balance.  
Debug the transaction to get more information.  
You won't be able to do it, though the Note: here is misleading. In the EVM, payable only refers to transfers of the primary token used to pay gas fees: ETH, Base ETH, Sepolia ETH, Matic, etc. It does not refer to the balance of our simple token.  
Instead, the transaction is reverting because of the built-in overflow/underflow protection. It's not a great programming practice to depend on this, so add an error for InsufficientTokens that returns the newSenderBalance.  
function transfer(address \_to, uint \_amount) public {  
 int newSenderBalance = int(balances[msg.sender] - \_amount);  
 if (newSenderBalance < 0) {  
 revert InsufficientTokens(newSenderBalance);  
 }  
  
 balances[msg.sender] = uint(newSenderBalance);  
 balances[\_to] += \_amount;  
}  
Try spending too much again. You'll get the same error in Remix:  
transact to MinimalToken.transfer pending ...  
transact to MinimalToken.transfer errored: VM error: revert.  
  
revert  
 The transaction has been reverted to the initial state.  
Note: The called function should be payable if you send value and the value you send should be less than your current balance.  
Debug the transaction to get more information.  
However, you can use the debug tool to review the error in memory to see that it now matches your custom error.  
Destroying Tokens  
Tokens can be effectively destroyed by accident, or on purpose. Accidental destruction happens when someone sends a token to an unowned wallet address. While it's possible that some day, some lucky person will create a new wallet and find a pleasant surprise, the most likely outcome is that any given randomly chosen address will never be used, thus no one will ever have the ability to use or transfer those tokens.  
Luckily, there are some protections here. Similar to credit card numbers, addresses have a built-in checksum that helps protect against typos. Try it out by trying to transfer tokens to the second Remix address, but change the first character in the address from A to B. You'll get an error:  
transact to MinimalToken.transfer errored: Error encoding arguments: Error: bad address checksum (argument="address", value="0xBb8483F64d9C6d1EcF9b849Ae677dD3315835cb2", code=INVALID\_ARGUMENT, version=address/5.5.0) (argument=null, value="0xBb8483F64d9C6d1EcF9b849Ae677dD3315835cb2", code=INVALID\_ARGUMENT, version=abi/5.5.0)  
A more guaranteed way to destroy, or burn a token, is to transfer it to the default address 0x0000000000000000000000000000000000000000. This address is unowned and unownable, making it mathematically impossible to retrieve any tokens that are sent to it. Redeploy and try it out by sending 1000 tokens to the zero address.  
The totalSupply remains unchanged, and the balance of the zero address are visible, but those tokens are stuck there forever.  
INFO  
The zero address currently has a balance of more than 11,000 ETH, worth over 20 million dollars! Its total holding of burned assets is estimated to be worth more than 200 million dollars!!!  
Conclusion  
In this lesson, you've learned to implement a simple token, which is really just a system to store the balance of each address, and a mechanism to transfer them from one wallet to another. You've also learned how to permanently destroy tokens, whether by accident, or on purpose.  
ON THIS PAGE  
Implementing a Token  
Transferring Tokens  
Destroying Tokens  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/minimal-tokens/minimal-tokens-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
Create a Minimal Token  
Transferring a Minimal Token  
Minimal Token  
Minimal Tokens Exercise  
ERC-20 Tokens  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Minimal Tokens Exercise  
Create a contract that adheres to the following specifications.  
Contract  
Create a contract called UnburnableToken. Add the following in storage:  
A public mapping called balances to store how many tokens are owned by each address  
A public uint to hold totalSupply  
A public uint to hold totalClaimed  
Other variables as necessary to complete the task  
Add the following functions.  
Constructor  
Add a constructor that sets the total supply of tokens to 100,000,000.  
Claim  
Add a public function called claim. When called, so long as a number of tokens equalling the totalSupply have not yet been distributed, any wallet that has not made a claim previously should be able to claim 1000 tokens. If a wallet tries to claim a second time, it should revert with TokensClaimed.  
The totalClaimed should be incremented by the claim amount.  
Once all tokens have been claimed, this function should revert with an error AllTokensClaimed. (We won't be able to test this, but you'll know if it's there!)  
Safe Transfer  
Implement a public function called safeTransfer that accepts an address \_to and an \_amount. It should transfer tokens from the sender to the \_to address, only if:  
That address is not the zero address  
That address has a balance of greater than zero Base Sepolia Eth  
A failure of either of these checks should result in a revert with an UnsafeTransfer error, containing the address.  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
CAUTION  
The contract specification contains actions that can only be performed once by a given address. As a result, the unit tests for a passing contract will only be successful the first time you test.  
You may need to submit a fresh deployment to pass  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Contract  
Constructor  
Claim  
Safe Transfer  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/minimal-tokens/transferring-a-minimal-token-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Introduction to Tokens  
Minimal Tokens  
Create a Minimal Token  
Transferring a Minimal Token  
Minimal Token  
Minimal Tokens Exercise  
ERC-20 Tokens  
ERC-721 Tokens  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Transferring a Minimal Token  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/new-keyword/creating-a-new-contract-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Creating a `new` Contract  
The New Keyword  
New Exercise  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Creating a `new` Contract  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/new-keyword/new-keyword-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Creating a `new` Contract  
The New Keyword  
New Exercise  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
New Exercise  
For this exercise, we're challenging you to build a solution requiring you to use a number of the concepts you've learned so far. Have fun and enjoy!  
Contracts  
Build a contract that can deploy copies of an address book contract on demand, which allows users to add, remove, and view their contacts.  
You'll need to develop two contracts for this exercise and import at least one additional contract.  
Imported Contracts  
Review the Ownable contract from OpenZeppelin. You'll need to use it to solve this exercise.  
You may wish to use another familiar contract to help with this challenge.  
AddressBook  
Create an Ownable contract called AddressBook. In it include:  
A struct called Contact with properties for:  
id  
firstName  
lastName  
a uint array of phoneNumbers  
Additional storage for contacts  
Any other necessary state variables  
It should include the following functions:  
Add Contact  
The addContact function should be usable only by the owner of the contract. It should take in the necessary arguments to add a given contact's information to contacts.  
Delete Contact  
The deleteContact function should be usable only by the owner and should delete the contact under the supplied \_id number.  
If the \_id is not found, it should revert with an error called ContactNotFound with the supplied id number.  
Get Contact  
The getContact function returns the contact information of the supplied \_id number. It reverts with ContactNotFound if the contact isn't present.  
QUESTION  
For bonus points (that only you will know about), explain why we can't just use the automatically generated getter for contacts?  
Get All Contacts  
The getAllContacts function returns an array with all of the user's current, non-deleted contacts.  
CAUTION  
You shouldn't use onlyOwner for the two get functions. Doing so won't prevent a third party from accessing the information, because all information on the blockchain is public. However, it may give the mistaken impression that information is hidden, which could lead to a security incident.  
AddressBookFactory  
The AddressBookFactory contains one function, deploy. It creates an instance of AddressBook and assigns the caller as the owner of that instance. It then returns the address of the newly-created contract.  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
ON THIS PAGE  
Imported Contracts  
AddressBook  
AddressBookFactory  
Add Contact  
Delete Contact  
Get Contact  
Get All Contacts  
Submit your Contract and Earn an NFT Badge! (BETA)  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/new-keyword/new-keyword-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Creating a `new` Contract  
The New Keyword  
New Exercise  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
The New Keyword  
You've seen the new keyword and used it to instantiate memory arrays with a size based on a variable. You can also use it to write a contract that creates other contracts.  
Objectives  
By the end of this lesson you should be able to:  
Write a contract that creates a new contract with the new keyword  
Creating a Simple Contract Factory  
A contract factory is a contract that creates other contracts. To start, let's create and interact with a very simple one. Create a new project in Remix and add a file called ContractFactory.sol.  
Adding the Template  
Imagine you want to create a contract that can store its owner's name and compliment them upon request. You can create this contract fairly easily.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
Deploy and test.  
Creating a Factory  
The Complimenter contract is a huge success! People love how it makes them feel and you've got customers banging on the doors and windows. Awesome!  
The only problem is that it takes time and effort to manually deploy a new version of the contract for each customer. Luckily, there's a way to write a contract that will act as a self-service portal for your customers.  
Start by adding a contract called ComplimenterFactory. The Remix interface makes things easier if you leave the factory in the same file as Complimenter.  
Add a function called CreateComplimenter that is public, accepts a string called \_name, and returns an address.  
Creating a new contract is simple: new Complimenter(\_name)  
You can also save the return from that instantiation into a variable. This reference can be used to call public functions in the deployed contract, and can be cast to an address. We can use it to get an easy reference to find the copies made by the factory. The end result should look similar to:  
Reveal code  
  
  
  
  
  
  
  
Testing  
Clear the environment if you haven't already, then start by deploying ComplimenterFactory. You've been working hard and deserve nice things, so call CreateComplimenter with your name.  
In the terminal, the decoded output will be the address of the new contract.  
{  
 "0": "address: 0x9e0BC6DB02E5aF99b8868f0b732eb45c956B92dD"  
}  
Copy only the address.  
Switch the CONTRACT to be deployed to Complimenter, then paste the address you copied in the field next to the At Address button which is below the Deploy button.  
Click At Address and the instance of Complimenter should appear below ComplimenterFactory. Test to confirm it works, then try deploying more instances with the factory.  
TIP  
If the deployed contract appears, but is instead a broken copy of the factory, it's because you didn't change the contract in the CONTRACT dropdown above the deploy button.  
Remix is trying to interact with Complimenter using the ABI from the factory contract, which won't work.  
Conclusion  
In this lesson, you learned how to deploy contracts from another contract by using the new keyword. You also learned that you look great today!  
ON THIS PAGE  
Creating a Simple Contract Factory  
Conclusion  
Adding the Template  
Creating a Factory  
Testing  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/reading-and-displaying-data/configuring-useReadContract  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Connecting to the Blockchain  
Reading and Displaying Data  
The `useAccount` Hook  
The `useReadContract` Hook  
Configuring `useReadContract`  
Writing to Contracts  
Exercise Contracts  
Configuring `useReadContract`  
The useReadContract hook has a number of configurable properties that will allow you to adapt it to your needs. You can combine the functionality of TanStack queries with [useBlockNumber] to watch the blockchain for changes, although doing so will consume a number of API calls.  
Objectives  
By the end of this guide you should be able to:  
Use useBlockNumber and the queryClient to automatically fetch updates from the blockchain  
Describe the costs of using the above, and methods to reduce those costs  
Configure arguments to be passed with a call to a pure or view smart contract function  
Call an instance of useReadContract on demand  
Utilize isLoading and isFetching to improve user experience  
Fetching Updates from the Blockchain  
You'll continue with the project you've been building and last updated while learning about the useReadContract hook.  
Once the excitement of your accomplishment of finally reading from your own contract subsides, try using BaseScan to add another issue, or vote on an existing issue. You'll notice that your frontend does not update. There are a few ways to handle this.  
The Watch Feature  
The easiest is to use useBlockNumber with the watch feature to automatically keep track of the block number, then use the queryClient to update when that changes. Make sure you decompose the queryKey from the return of useReadContract.  
import { useEffect, useState } from 'react';  
import { useReadContract, useBlockNumber } from 'wagmi';  
import { useQueryClient } from '@tanstack/react-query';  
  
// Other Code  
  
export function IssueList() {  
 // Other Code  
  
 const queryClient = useQueryClient();  
 const { data: blockNumber } = useBlockNumber({ watch: true });  
  
 const {  
 data: issuesData,  
 isError: issuesIsError,  
 isPending: issuesIsPending,  
 queryKey: issuesQueryKey,  
 } = useReadContract({  
 address: contractData.address as `0x${string}`,  
 abi: contractData.abi,  
 functionName: 'getAllIssues',  
 });  
  
 // Note that this is a separate `useEffect` from the one that handles the  
 // update after the list of issues is returned  
 useEffect(() => {  
 queryClient.invalidateQueries({ queryKey: issuesQueryKey });  
 }, [blockNumber, queryClient, issuesQueryKey]);  
  
 // Return code  
}  
Try adding a new issue and it will automatically appear on the list, although it may take more time than you are used to. Blockchain is still slower than the web.  
It works! Unfortunately, you can't really stop here, unless you're working on a hackathon prototype or a very early stage demo. The catch is that wagmi has a default pollingInterval of 4 seconds, so having this watch causes it to call eth\_blocknumber constantly, which then triggers an eth\_call, both of which use api credits.  
If you were to take the obvious approach of adding a useReadContract for every function you wanted data from, and set it to watch, things would quickly get out of hand. A single open web page with 15 functions watched in this way will hit rate-limiting in as short as an hour.  
INFO  
Don't do this, either use multi-call via useReadContracts, or consolidate your views into a single function that fetches all the data you need in one call.  
Luckily, you have options to control these calls a little better.  
Pausing On Blur  
Once quick improvement is to simply stop watching the blockchain if the website doesn't have focus. To see this in action, add a state variable to count how many times the function has settled, and one for if the page is focused. You'll also need to set up event listeners to set the state of the latter when the page is focused or blurred.  
const [timesCalled, setTimesCalled] = useState(0);  
const [pageIsFocused, setPageIsFocused] = useState(true);  
  
useEffect(() => {  
 const onFocus = () => setPageIsFocused(true);  
 const onBlur = () => setPageIsFocused(false);  
  
 window.addEventListener('focus', onFocus);  
 window.addEventListener('blur', onBlur);  
  
 return () => {  
 window.removeEventListener('focus', onFocus);  
 window.removeEventListener('blur', onBlur);  
 };  
}, []);  
Then, update the watch for useBlockNumber so that it only does so if pageIsFocused.  
const { data: blockNumber } = useBlockNumber({ watch: pageIsFocused });  
Add a line to the useEffect for blockNumber increment your counter as well.  
useEffect(() => {  
 setTimesCalled((prev) => prev + 1);  
 queryClient.invalidateQueries({ queryKey: issuesQueryKey });  
}, [blockNumber, queryClient]);  
Finally, surface your counter in the component.  
return (  
 <div>  
 <h2>Number of times called</h2>  
 <p>{timesCalled.toString()}</p>  
 <p>{'Has focus: ' + pageIsFocused}</p>  
 <h2>All Issues</h2>  
 <div>{renderIssues()}</div>  
 </div>  
);  
Now, when you watch the page, the count will go up every four seconds. When you switch to another tab or window, the counter will pause until you switch back.  
Adjusting the Polling Rate  
You likely need to share timely updates with your users, but how timely do those updates need to be to meet the requirements of your app? If you're doing instant messaging, 4 seconds may even be too long (though any faster is running into the speed blocks are added in most L2s).  
A more robust DAO is going to have a voting period of at least a day or two, so those users probably don't need to see that there is a new issue within 4 seconds of it hitting the chain.  
Adjust your pollingInterval by setting it in getDefaultConfig in \_app.tsx:  
const config = getDefaultConfig({  
 appName: 'RainbowKit App',  
 projectId: 'YOUR\_PROJECT\_ID',  
 chains: [baseSepolia],  
 ssr: true,  
 pollingInterval: 30\_000,  
});  
Setting it to 30 seconds, or 30,000 milliseconds, will reduce your API calls dramatically, without negatively impacting members of the DAO.  
You can also set pollingInterval if you're using createConfig instead of the default.  
Updating on Demand  
You can use a similar system to call your update function on demand. First, add a button, a handler for that button, and a state variable for it to set:  
const [triggerRead, setTriggerRead] = useState(false);  
  
const handleTriggerRead = () => {  
 setTriggerRead(true);  
};  
return (  
 <div>  
 <button onClick={handleTriggerRead}>Read Now</button>  
 <h2>Number of times called</h2>  
 <p>{timesCalled.toString()}</p>  
 <p>{'Has focus: ' + pageIsFocused}</p>  
 <h2>All Issues</h2>  
 <div>{renderIssues()}</div>  
 </div>  
);  
Finally, set watch to equal triggerRead, instead of pageIsFocused, and reset triggerRead in the useEffect.  
const { data: blockNumber } = useBlockNumber({ watch: triggerRead });  
  
// Other code...  
  
useEffect(() => {  
 setTriggerRead(false);  
 queryClient.invalidateQueries({ queryKey: issuesQueryKey });  
}, [blockNumber, queryClient]);  
Now, when the user clicks the button, the hook will call the read function a single time, then set watch back to false.  
Setting UI Elements  
You can use the "is" return values to set UI elements depending on the status of the hook as it attempts to call a function on the blockchain.  
Try to modify your button to provide feedback to the user that the function has been called.  
// Bad code example, do not use  
<button disabled={issuesIsLoading} onClick={handleTriggerRead}>  
 {issuesIsLoading ? 'Loading' : 'Read Now'}  
</button>  
The above code won't break anything, but nothing will appear to happen. This happens because isLoading is only true in circumstances where data is loading for the first time, but no data is present. You could use this to show a spinning wheel in place of the list of issues.  
Instead, try decomposing isFetching in your useReadContract. This property is true while data is being fetched, even if data has already been loaded once.  
// Imperfect code example, do not use  
<button disabled={issuesIsFetching} onClick={handleTriggerRead}>  
 {issuesIsFetching ? 'Loading' : 'Read Now'}  
</button>  
You'll probably see the button flicker very quickly since the call doesn't take very long. For a production app, you'd need to add additional handling to smooth out the experience.  
Passing Arguments  
Arguments are passed into a useReadContract hook by adding an array of arguments, in order, to the args property. Common practice is to use React state variables set by UI elements to enable the arguments to be set and modified. For example, you might create a drop-down to set issueNumber, then fetch that issue with:  
// Incomplete code stub  
const [issueNumber, setIssueNumber] = useState(0);  
  
const { isLoading: getIssueIsLoading } = useReadContract({  
 address: contractData.address as `0x${string}`,  
 abi: contractData.abi,  
 functionName: 'getIssue',  
 args: [issueNumber],  
});  
Depending on your design needs, you can use the techniques above to either watch constantly for updates, or fetch them on user action.  
Conclusion  
In this guide, you've learned how to use the watch feature of useBlockNumber combined with useEffect and queryClient.invalidateQueries to enable your frontend to see updates to your smart contract. You've also learned the costs of doing so, and some strategies for mitigation. You've learned how to pass arguments to your functions. Finally, you've learned how to use the properties returned by useReadContract to adjust your UI to improve the experience for your users.  
ON THIS PAGE  
Fetching Updates from the Blockchain  
Setting UI Elements  
The Watch Feature  
Pausing On Blur  
Adjusting the Polling Rate  
Updating on Demand  
Passing Arguments  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/reading-and-displaying-data/useAccount  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Connecting to the Blockchain  
Reading and Displaying Data  
The `useAccount` Hook  
The `useReadContract` Hook  
Configuring `useReadContract`  
Writing to Contracts  
Exercise Contracts  
The `useAccount` Hook  
wagmi is a library that provides React hooks that trade a somewhat complex setup process for a great developer experience when building a frontend around the constraints and quirks of onchain building. One of the hooks, useAccount, provides access to information about your users' wallet and connection information.  
You can use this for connection-status-based rendering, to enable or disable controls or views based on address, and many other useful tasks.  
Objectives  
By the end of this guide you should be able to:  
Implement the `useAccount`` hook to show the user's address, connection state, network, and balance  
Implement an isMounted hook to prevent hydration errors  
Displaying Connection Information  
We'll be working from an app generated by RainbowKit's quick start. Either open the one you created when we were exploring Wallet Connectors, or create a new one for this project.  
Either way, change the list of chains to only include baseSepolia as the network option. You don't want to accidentally spend real money while developing!  
You can set up your providers as described in Introduction to Providers, or use the default from RainbowKit:  
const config = getDefaultConfig({  
 appName: 'RainbowKit App',  
 projectId: 'YOUR APP ID',  
 chains: [baseSepolia],  
 ssr: true,  
});  
Either way, be sure to set ssr to true, or you will get a hydration error from Next.js.  
The useAccount Hook  
The useAccount hook allows you to access account and connection data from within any of your components.  
Add a folder for components and a file called ConnectionWindow.tsx in that folder. Add the below component to the file, and replace the boilerplate text in index.tsx with an instance of it.  
// ConnectionWindow.tsx  
export function ConnectionWindow() {  
 return (  
 <div>  
 <p>Connection Status</p>  
 </div>  
 );  
}  
// index.tsx  
import { ConnectButton } from '@rainbow-me/rainbowkit';  
import type { NextPage } from 'next';  
import Head from 'next/head';  
import styles from '../styles/Home.module.css';  
import { ConnectionWindow } from '../components/ConnectionWindow';  
  
const Home: NextPage = () => {  
 return (  
 <div className={styles.container}>  
 <main className={styles.main}>  
 <ConnectButton />  
 <ConnectionWindow />  
 </main>  
 </div>  
 );  
};  
  
export default Home;  
For the purposes of this exercise, open styles/Home.module.css and delete or comment out .main. Doing so will move the content to the top of the page, which will prevent the RainbowKit modal from blocking your ability to see changes.  
Return to ConnectionWindow.tsx and add the useAccount hook to the top, where you'd add any state variables. The general pattern for wagmi hooks is you decompose the properties you want to use from a function call of the name of the hook. For some, you'll add a config object to that call, but it's not needed for this one.  
import { useAccount } from 'wagmi';  
  
export function ConnectionWindow() {  
 const { address, isConnected, isConnecting, isDisconnected } = useAccount();  
  
 return (  
 <div>  
 <h2>Connection Status</h2>  
 </div>  
 );  
}  
You can see all the deconstructable return options in the UseAccountReturnType:  
Update your <div> to show the address of the connected wallet:  
<div>  
 <h2>Connection Status</h2>  
 <div>  
 <p>{'Address: ' + address}</p>  
 </div>  
</div>  
Test it out by connecting and disconnecting with your wallet. You should see your full address when you are connected, and the address will be undefined when you are disconnected.  
Connection Status Conditional Rendering  
It isn't very nice to display a value of undefined to the user, so let's use the connection status values for conditional rendering depending on if the user is disconnected, connected, or connecting.  
A common pattern is to use the conditional directly in the html return of a component or render function. For example, we could add a line to show that we're connecting as demonstrated:  
<div>  
 <h2>Connection Information</h2>  
 <div>  
 {!isConnecting && <p>Please click Connect in your wallet...</p>}  
 <p>{"Address: " + address}</p>  
 </div>  
</div>  
Connect and disconnect your wallet a few times. The isConnecting state is true while the Connect to website wallet UI is open.  
Autoconnect is enabled by default, so you'll need to clear the connection from your wallet settings to see this more than once. Otherwise, it will briefly flash as the autoconnect processes.  
Use the connected property in the same way to only render the wallet address if there is a wallet connected. Similarly, use the isDisconnected property to show a message asking the user to connect.  
<div>  
 <h2>Connection Information</h2>  
 <div>  
 {isConnecting && <p>Please click Connect in your wallet...</p>}  
 {isConnected && <p>{"Address: " + address}</p>}  
 {isDisconnected && <p>Please connect your wallet to use this app.</p>}  
 </div>  
</div>  
Conclusion  
In this guide, you've learned how the useAccount hook gives you access to information about the user's connection status and wallet. It can be used in any part of your app that is wrapped by the wagmi context provider. You've also learned a technique for conditional rendering based on connection status. Finally, you've learned to set the ssr flag to prevent hydration errors due to the client and server possessing different information about the user's connection status.  
ON THIS PAGE  
Displaying Connection Information  
Conclusion  
The useAccount Hook  
Connection Status Conditional Rendering  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/reading-and-displaying-data/useReadContract  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Connecting to the Blockchain  
Reading and Displaying Data  
The `useAccount` Hook  
The `useReadContract` Hook  
Configuring `useReadContract`  
Writing to Contracts  
Exercise Contracts  
The `useReadContract` Hook  
The useReadContract hook is wagmi's method of calling pure and view functions from your smart contracts. As with useAccount, useReadContract contains a number of helpful properties to enable you to manage displaying information to your users.  
Objectives  
By the end of this guide you should be able to:  
Implement wagmi's useReadContract hook to fetch data from a smart contract  
Convert data fetched from a smart contract to information displayed to the user  
Identify the caveats of reading data from automatically-generated getters  
Contract Setup  
For this guide, you'll be continuing from the project you started for the useAccount hook. You'll work with an upgrade to the contract that you may have created if you completed the ERC 20 Tokens Exercise. See below for an example you can use if you don't already have your own!  
The contract creates a very simple DAO, in which users can create issues and vote for them, against them, or abstain. Anyone can claim 100 tokens. This is an insecure system for demonstration purposes, since it would be trivial to claim a large number of tokens with multiple wallets, then transfer them to a single address and use that to dominate voting.  
But it makes it much easier to test!  
CAUTION  
If you're using your own contract, please redeploy it with the following view functions:  
function numberOfIssues() public view returns(uint) {  
 return issues.length;  
}  
  
function getAllIssues() public view returns(ReturnableIssue[] memory) {  
 ReturnableIssue[] memory allIssues = new ReturnableIssue[](issues.length);  
  
 for(uint i = 0; i < issues.length; i++) {  
 allIssues[i] = getIssue(i);  
 }  
  
 return allIssues;  
}  
You also need to make the getIssue function public. The original spec called for it to be external.  
Create Demo Issues  
To start, you'll need to put some data into your contract so that you can read it from your frontend. Open Sepolia BaseScan, find your contract, connect with your wallet, and call the claim function.  
Add the following two issues:  
\_issueDesc: We should enable light mode by default.  
\_quorom: 2  
\_issueDesc: We should make inverted mouse controls the default selection.  
\_quorom: 2  
Switch to a different wallet address. Claim your tokens with the new address, and add one more issue:  
\_issueDesc: Two spaces, not four, not tabs!  
\_quorom: 2  
Call the getAllIssues function under the Read Contract tab to make sure all three are there.  
Reading from your Smart Contract  
To be able to read from your deployed smart contract, you'll need two pieces of information: the address and ABI. These are used as parameters in the useReadContract hook.  
If you're using Hardhat, both of these can be conveniently found in a json file in the deployments/<network> folder, named after your contract. For example, our contract is called FEWeightedVoting, so the file is deployments/base-sepolia/FEWeightedVoting.json.  
If you're using something else, it should produce a similar artifact, or separate artifacts with the ABI and address. If this is the case, make the adjustments you need when you import this data.  
Either way, add a folder called deployments and place a copy of the artifact file(s) inside.  
Using the useReadContract Hook  
Add a file for a new component called IssueList.tsx. You can start with:  
import { useReadContract } from 'wagmi';  
  
export function IssueList() {  
 return (  
 <div>  
 <h2>All Issues</h2>  
 <div>{/\* TODO: List each issue \*/}</div>  
 </div>  
 );  
}  
You'll need to do some prepwork to enable Typescript to more easily interpret the data returned from your contract. Add an interface called Issue that matches with the ReturnableIssue type:  
interface Issue {  
 voters: string[];  
 issueDesc: string;  
 votesFor: bigint;  
 votesAgainst: bigint;  
 votesAbstain: bigint;  
 totalVotes: bigint;  
 quorum: bigint;  
 passed: boolean;  
 closed: boolean;  
}  
DANGER  
Be very careful here! bigint is the name of the type, BigInt is the name of the constructor for that type. If you incorrectly use the constructor as the type, much of your code will still work, but other parts will express very confusing bugs.  
Now, import useState and add a state variable to hold your list of Issues.  
const [issues, setIssues] = useState<Issue[]>([]);  
You'll also need to import your contract artifact:  
import contractData from '../deployments/FEWeightedVoting.json';  
Finally, the moment you've been waiting for: Time to read from your contract! Add an instance of the useReadContract hook. It works similarly to the useAccount hook. Configure it with:  
const {  
 data: issuesData,  
 isError: issuesIsError,  
 isPending: issuesIsPending,  
} = useReadContract({  
 address: contractData.address as `0x${string}`,  
 abi: contractData.abi,  
 functionName: 'getAllIssues',  
});  
You can use useEffect to do something when the call completes and the data. For now, just log it to the console:  
useEffect(() => {  
 if (issuesData) {  
 const issuesList = issuesData as Issue[];  
 console.log('issuesList', issuesList);  
 setIssues(issuesList);  
 }  
}, [issuesData]);  
Add in instance of your new component to index.tsx:  
<main className={styles.main}>  
 <ConnectButton />  
 <ConnectionWindow />  
 <IssueList />  
</main>  
Run your app, and you should see your list of issues fetched from the blockchain and displayed in the console!  
Breaking down the hook, you've:  
Renamed the properties decomposed from useReadContract to be specific for our function. Doing so is helpful if you're going to read from more than one function in a file  
Configured the hook with the address and ABI for your contract  
Made use of useEffect to wait for the data to be returned from the blockchain, log it to the console, and set the list of Issues in state.  
Displaying the Data  
Now that you've got the data in state, you can display it via your component. One strategy to display a list of items is to compile a ReactNode array in a render function.  
function renderIssues() {  
 return issues.map((issue) => (  
 <div key={issue.issueDesc}>  
 <h3>{issue.issueDesc}</h3>  
 <p>{'Voters: ' + issue.voters.toString()}</p>  
 <p>{'Votes For: ' + issue.votesFor.toString()}</p>  
 <p>{'Votes Against: ' + issue.votesAgainst.toString()}</p>  
 <p>{'Votes Abstain: ' + issue.votesAbstain.toString()}</p>  
 <p>{'Quorum: ' + issue.quorum.toString()}</p>  
 <p>{'Passed: ' + issue.passed}</p>  
 <p>{'Closed: ' + issue.closed}</p>  
 </div>  
 ));  
}  
Then, call the render function in the return for your component:  
return (  
 <div>  
 <h2>All Issues</h2>  
 <div>{renderIssues()}</div>  
 </div>  
);  
You'll now see your list of issues rendered in the browser! Congrats, you've finally made a meaningful connection between your smart contract and your frontend!  
A Caveat with Automatic Getters  
Remember how the Solidity compiler creates automatic getters for all of your public state variables? This feature is very helpful, but it can create bewildering results when you use it for structs that contain mappings. Remember, nesting mappings cannot be returned outside the blockchain. The enumerableSet protects you from this problem, because it has private variables inside it, which prevents setting issues as public. Had we instead used a mapping, we'd lose this protection:  
 // Code for demo only  
 struct Issue {  
 mapping(address => bool) voters;  
 string issueDesc;  
 uint votesFor;  
 uint votesAgainst;  
 uint votesAbstain;  
 uint totalVotes;  
 uint quorum;  
 bool passed;  
 bool closed;  
 }  
Redeploy with the above change, and add a second useReadContract to fetch an individual issue using the getter:  
// Bad code for example, do not use  
const {  
 data: getOneData,  
 isError: getOneIsError,  
 isPending: getOneIsPending,  
} = useReadContract({  
 address: contractData.address as `0x${string}`,  
 abi: contractData.abi,  
 functionName: 'issues',  
 args: [1],  
});  
  
useEffect(() => {  
 if (getOneData) {  
 console.log('getOneData', getOneData);  
 const issueOne = getOneData as Issue;  
 console.log('Issue One', issueOne);  
 }  
}, [getOneData]);  
Everything appears to be working just fine, but how is issueOne.desc undefined? You can see it right there in the log!  
If you look closely, you'll see that voters is missing from the data in the logs. What's happening is that because the nested mapping cannot be returned outside the blockchain, it simply isn't. TypeScript then gets the data and does the best it can to cast it as an Issue. Since voters is missing, this will fail and it instead does the JavaScript trick of simply tacking on the extra properties onto the object.  
Take a look at the working example above where you retrieved the list. Notice that the keys in your Issue type are in that log, but are missing here. The assignment has failed, and all of the Issue properties are null.  
Conclusion  
In this guide, you've learned how to use the useReadContract hook to call pure and view functions from your smart contracts. You then converted this data into React state and displayed it to the user. Finally, you explored a tricky edge case in which the presence of a nested mapping can cause a confusing bug when using the automatically-generated getter.  
Simple DAO Contract Example  
Use this contract if you don't have your own from the ERC 20 Tokens Exercise. You can also use this if you want to cheat to get that badge. Doing so would be silly though!  
CAUTION  
If you use your own contract, redeploy it with the numberOfIssues and getAllIssues functions from the bottom of the contract below. We'll need this for our first pass solution for getting all the Issues in the contract.  
You also need to make the getIssue function public. The original spec called for it to be external.  
// SPDX-License-Identifier: MIT  
  
pragma solidity ^0.8.17;  
  
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";  
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";  
  
contract FEWeightedVoting is ERC20 {  
 using EnumerableSet for EnumerableSet.AddressSet;  
  
 mapping(address => bool) claimed;  
 uint public maxSupply = 1000000;  
 uint totalClaimed;  
  
 uint constant claimAmount = 100;  
  
 error TokensClaimed();  
 error AllTokensClaimed();  
 error NoTokensHeld();  
 error QuorumTooHigh(uint);  
 error AlreadyVoted();  
 error VotingClosed();  
  
 enum Vote {  
 AGAINST,  
 FOR,  
 ABSTAIN  
 }  
  
 struct Issue {  
 EnumerableSet.AddressSet voters;  
 string issueDesc;  
 uint votesFor;  
 uint votesAgainst;  
 uint votesAbstain;  
 uint totalVotes;  
 uint quorum;  
 bool passed;  
 bool closed;  
 }  
  
 // EnumerableSets are mappings and cannot be returned outside a contract  
 struct ReturnableIssue {  
 address[] voters;  
 string issueDesc;  
 uint votesFor;  
 uint votesAgainst;  
 uint votesAbstain;  
 uint totalVotes;  
 uint quorum;  
 bool passed;  
 bool closed;  
 }  
  
 Issue[] issues;  
  
 constructor(  
 string memory \_name,  
 string memory \_symbol  
 ) ERC20(\_name, \_symbol) {  
 // Burn Issue 0  
 issues.push();  
 }  
  
 function claim() public {  
 if (claimed[msg.sender] == true) {  
 revert TokensClaimed();  
 }  
  
 if (totalSupply() >= maxSupply) {  
 revert AllTokensClaimed();  
 }  
  
 \_mint(msg.sender, claimAmount);  
 claimed[msg.sender] = true;  
 }  
  
 function createIssue(  
 string memory \_issueDesc,  
 uint \_quorum  
 ) public returns (uint) {  
 if (balanceOf(msg.sender) == 0) {  
 revert NoTokensHeld();  
 }  
  
 if (\_quorum > totalSupply()) {  
 revert QuorumTooHigh(\_quorum);  
 }  
  
 Issue storage newIssue = issues.push();  
 newIssue.issueDesc = \_issueDesc;  
 newIssue.quorum = \_quorum;  
 return issues.length - 1;  
 }  
  
 function getIssue(uint \_id) public view returns (ReturnableIssue memory) {  
 Issue storage issue = issues[\_id];  
 return  
 ReturnableIssue(  
 issue.voters.values(),  
 issue.issueDesc,  
 issue.votesFor,  
 issue.votesAgainst,  
 issue.votesAbstain,  
 issue.totalVotes,  
 issue.quorum,  
 issue.closed,  
 issue.passed  
 );  
 }  
  
 function vote(uint \_issueId, Vote \_vote) public {  
 Issue storage issue = issues[\_issueId];  
 if (issue.voters.contains(msg.sender)) {  
 revert AlreadyVoted();  
 }  
 if (issue.closed) {  
 revert VotingClosed();  
 }  
 issue.voters.add(msg.sender);  
  
 if (\_vote == Vote.FOR) {  
 issue.votesFor += balanceOf(msg.sender);  
 } else if (\_vote == Vote.AGAINST) {  
 issue.votesAgainst += balanceOf(msg.sender);  
 } else if (\_vote == Vote.ABSTAIN) {  
 issue.votesAbstain += balanceOf(msg.sender);  
 } else {  
 revert("Error...");  
 }  
  
 issue.totalVotes += balanceOf(msg.sender);  
  
 if (issue.totalVotes >= issue.quorum) {  
 issue.closed = true;  
 if (issue.votesFor > issue.votesAgainst) {  
 issue.passed = true;  
 }  
 }  
 }  
  
 function numberOfIssues() public view returns(uint) {  
 return issues.length;  
 }  
  
 function getAllIssues() public view returns(ReturnableIssue[] memory) {  
 ReturnableIssue[] memory allIssues = new ReturnableIssue[](issues.length);  
  
 for(uint i = 0; i < issues.length; i++) {  
 allIssues[i] = getIssue(i);  
 }  
  
 return allIssues;  
 }  
}  
ON THIS PAGE  
Contract Setup  
Reading from your Smart Contract  
Using the useReadContract Hook  
Conclusion  
Using the useReadContract Hook  
Displaying the Data  
A Caveat with Automatic Getters  
Simple DAO Contract Example  
Simple DAO Contract Example  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/storage/how-storage-works  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Simple Storage  
Storing Data  
How Storage Works  
How Storage Works  
Storage Exercise  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
How Storage Works  
In this article, we will delve into the workings of Ethereum storage, explore the nuances of variable declaration ordering, and provide examples of efficient and inefficient storage practices to create optimized smart contracts.  
Objectives:  
By the end of this lesson you should be able to:  
Diagram how a contract's data is stored on the blockchain (Contract -> Blockchain)  
Order variable declarations to use storage efficiently  
Diagram how variables in a contract are stored (Variable -> Contract)  
Introduction  
Creating smart contracts that can operate efficiently requires a thorough understanding of how storage works in Ethereum. When designing a contract, you need to consider the storage requirements of the contract, including the types of storage needed, the gas costs associated with storage operations, and how to manage storage effectively. Poor storage management practices can lead to bloated contracts that consume excessive gas, making them more expensive to execute. By following best practices for storage management, you'll be equipped to create contracts that are lean, efficient, and cost-effective.  
Smart Contract Data Storage  
Key-Value Store  
Smart contracts on Ethereum store and manage data utilizing a key-value store model, where each piece of data is identified by a unique key and accompanied by its corresponding value.  
In this diagram, the keys (user addresses) are unique identifiers used to index the corresponding values (balances):  
This model can be compared to a dictionary or a map where the key serves as the index and the value represents the data associated with that index. However, the key-value store has distinct characteristics that set it apart from these traditional data structures, which make it a more optimal choice for smart contracts on Ethereum.  
Simplicity: It is simple and straightforward, which allows for easier implementation and maintenance within a contract.  
Scalability: It is highly scalable, making it well-suited for managing vast amounts of data typically associated with apps and smart contracts. This scalability helps maintain performance levels even as data storage requirements grow.  
Fixed-size chunks: Storing data in fixed-size 32-byte chunks optimizes storage space and ensures that data location calculations are more efficient. This feature is particularly beneficial in the context of Ethereum, where storage costs are a significant concern.  
Efficient storage and retrieval: It is optimized for storing and retrieving large volumes of data efficiently, which is essential for quick access to stored information.  
Security and immutability: Unlike other storage models that may allow direct data manipulation, key-value stores within Ethereum's environment ensure data integrity and security through transaction-based modifications. This feature aligns with the decentralized and trustless nature of blockchain technology.  
Gas-efficiency: In Ethereum, every operation within a smart contract execution consumes gas. The key-value store model is designed to be gas-efficient, minimizing the gas consumption for storage and retrieval operations, thus reducing the overall cost of contract execution.  
Compatibility with decentralized environments: It is particularly suitable for decentralized environments, where data consistency, integrity, and security are crucial. The model's design inherently addresses the challenges posed by multi-threaded or concurrent environments where multiple processes or functions may attempt to access or modify the same data simultaneously.  
Types of Storage  
There are three primary types of storage in Ethereum smart contracts: storage, memory, and stack. Each type has its specific use case and characteristics, which make them suitable for different aspects of smart contract execution.  
Storage  
Storage is the most persistent and expensive form of data storage. Data stored in the contract's storage persists across transaction executions and is accessible to any function within the smart contract. This storage is also visible on the blockchain and can be read by external sources, making it suitable for storing important and long-lasting information related to the contract's state.  
Key attributes of storage:  
Persistent: Data remains in storage even after the contract execution finishes, allowing for state continuity across multiple transactions.  
Expensive: Storing and modifying data in storage consumes more gas compared to other data locations, making it costly in terms of transaction fees.  
Visible on the blockchain: Storage data is publicly available and can be read by external parties.  
Consider the following contract:  
contract StorageDemo {  
 // Declare a state variable to store data in storage  
 uint256 public storedData;  
  
 // Function to update the storedData variable in storage  
 function updateData(uint256 newData) public {  
 storedData = newData;  
 }  
}  
The contract includes a state variable called storedData, which is stored in the contract's storage. The public visibility modifier allows anyone to access this variable. The contract also includes a public function called updateData, which can be called by anyone to modify the value of storedData in storage.  
Any changes made to storedData in storage will persist across multiple transactions and will be visible to anyone who reads the blockchain. Please note that storage is more expensive than other data locations, so it is important to use it judiciously to minimize gas costs.  
Memory  
Memory is a temporary and more affordable data location. It's used to save data during the execution of a single transaction. Once the transaction is complete, the memory is wiped clean and any data within it is lost. Memory is suitable for storing intermediate variables and temporary data that does not need to persist across multiple transactions.  
Key attributes of memory:  
Temporary: Data in memory is only available during a single transaction execution and is lost afterward.  
Less expensive: Saving and modifying data in memory consumes less gas compared to storage, making it more cost-effective for temporary data.  
Not visible on the blockchain: Memory data is not accessible to external parties and remains confined to the transaction execution.  
Consider the following contract:  
contract MemoryDemo {  
 // Declare a state variable to store data in storage  
 uint256 public storedData;  
  
 // Function to update the storedData variable in memory  
 function updateData(uint256 newData) public {  
 // Declare a memory variable to store the new data  
 uint256 tempData = newData;  
  
 // Assign the value of the memory variable to the storage variable  
 storedData = tempData;  
 }  
}  
In the contract, we declare a memory variable called tempData and assign the input parameter newData to it to update its value. The tempData variable is then assigned to the storedData variable to update its value in storage.  
Unlike storage, data stored in memory is not persisted across transactions and is only accessible during the execution of the function. However, accessing and modifying data in memory is less expensive than doing so in storage, making it a more efficient option when dealing with temporary data. Additionally, any data stored in memory is not visible on the blockchain and cannot be read by external parties.  
Stack  
The stack is another form of temporary data storage, specifically used for holding function arguments, local variables, and intermediate values during function execution. The stack follows a Last-In-First-Out (LIFO) structure, meaning that the most recently added item is the first to be removed. This storage type is highly efficient but has limited space, making it suitable for small-scale data manipulation during function execution.  
The stack is an internal data structure used by the EVM (Ethereum Virtual Machine) for computation during the execution of transactions. When a transaction is executed by the EVM, the bytecode of the smart contract is loaded into memory, and the EVM uses the stack to keep track of intermediate results and execute operations.  
In Solidity, developers do not interact with the stack directly, but can optimize their code to make the best use of it and minimize the amount of gas used during transaction execution. This can include using more efficient algorithms or data structures, or avoiding unnecessary operations that can increase the depth of the stack.  
Key attributes of the stack:  
Temporary: Like memory, stack data is only available during a single transaction execution and is lost afterward.  
Highly efficient: Stack operations consume minimal gas, making it the most cost-effective storage option for small-scale data manipulation.  
LIFO structure: The stack follows the Last-In-First-Out order, which allows for efficient management of function arguments, local variables, and intermediate values.  
Limited space: The stack has a maximum depth of 1024, restricting the number of elements it can hold at a given time.  
Limited visibility: Only the top 16 elements in the stack are accessible, limiting how many variables and other elements can be in scope at one time.  
Let's compare two versions of a function and analyze their gas efficiency with regard to stack usage and gas consumption:  
contract GasEfficiencyDemo {  
 uint256 public result;  
  
 // Less efficient  
 function sumLessEfficient(uint256 a, uint256 b) public {  
 uint256 temp = a + b;  
 result = temp;  
 }  
  
 // More efficient  
 function sumMoreEfficient(uint256 a, uint256 b) public {  
 result = a + b;  
 }  
}  
In the sumLessEfficient function, the sum of the two input arguments a and b is first assigned to the temporary variable temp before being assigned to the state variable result. This additional step introduces an extra variable on the stack, which requires more gas for stack operations and consumes more gas overall.  
In contrast, the sumMoreEfficient function directly assigns the sum of the input arguments a and b to the state variable result. This eliminates the need for the temporary variable and reduces the stack usage, leading to lower gas consumption for stack operations and a more gas-efficient execution.  
Although the difference in gas consumption between these two functions may not be significant for such a simple example, the principle of minimizing stack usage and optimizing code to reduce gas consumption is essential for developing efficient smart contracts. By avoiding unnecessary variables and operations, you can improve the gas efficiency of your functions and reduce the cost of executing them on the EVM.  
Variable Storage  
Variable Packing  
As we've learned, minimizing the storage footprint of a contract can substantially reduce gas costs. To make storage more efficient, Ethereum employs a concept called variable packing.  
Variable packing is the process of placing multiple smaller variables into a single storage slot to optimize storage usage. A storage slot is a fixed-size container that can hold up to 32 bytes of data. Ethereum's Solidity compiler automatically packs smaller variables together if they can fit into a single storage slot.  
Ordering Variable Declarations  
When declaring variables in a contract, their order can impact a contract's gas usage. You can optimize storage by declaring variables of similar sizes together, such that they can be packed into the same storage slot.  
Let's illustrate how this works:  
contract StoragePackingExample {  
 uint8 a; // 1 byte  
 uint8 b; // 1 byte  
 uint256 c; // 32 bytes  
}  
In this example, the compiler will automatically pack a and b into the same storage slot, as they are both 1-byte variables and can fit into a single 32-byte storage slot. However, c requires a separate storage slot due to its size (32 bytes).  
If these variables were not in the correct order, the contract would not take advantage of variable packing. The variables would take up more storage and would potentially consume more gas to execute the contract.  
Let's consider an inefficient example:  
contract StoragePackingBadExample {  
 uint8 a; // 1 byte  
 uint256 b; // 32 bytes  
 uint8 c; // 1 byte  
}  
In this contract, the variables are not declared in the optimal order, and the compiler would store these variables in the following way:  
To make the most of variable packing, it's important to group variables of the same size together and avoid mixing variable sizes. By doing this, the compiler can store them more efficiently, reducing the overall storage usage of the contract. This optimization will not only reduce the gas costs associated with storage, but it will also improve the contract's execution speed.  
Conclusion  
Creating efficient and optimized smart contracts on Ethereum requires a thorough understanding of how storage works. Smart contracts use a key-value store model to manage and store data, which is simple, scalable, gas-efficient, and suitable for decentralized environments. There are three types of storage in Ethereum smart contracts: storage, memory, and stack, each with specific characteristics. Developers can optimize storage usage by using variable packing and ordering variable declarations based on their size. By following best practices for storage management, developers can create contracts that are lean, efficient, cost-effective, and improve their execution speed.  
See Also  
Understanding Ethereum Smart Contract Storage  
What is Smart Contract Storage Layout  
ON THIS PAGE  
Introduction  
Smart Contract Data Storage  
Variable Storage  
Variable Packing  
Ordering Variable Declarations  
Conclusion  
Variable Packing  
Ordering Variable Declarations  
See Also  
See Also  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/storage/how-storage-works-video  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Simple Storage  
Storing Data  
How Storage Works  
How Storage Works  
Storage Exercise  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
How Storage Works  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/storage/simple-storage-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Simple Storage  
Storing Data  
How Storage Works  
How Storage Works  
Storage Exercise  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Storing Data  
Ultimately, the power of the blockchain is that anyone can store their data on it via the storage in a smart contract. In this step by step guide, you'll learn how to access and use the storage data location.  
Objectives  
By the end of this lesson you should be able to:  
Use the constructor to initialize a variable  
Access the data in a public variable with the automatically generated getter  
Order variable declarations to use storage efficiently  
Simple Storage Contract  
Create a contract called SimpleStorage.  
Add a Storage Variable  
In Solidity, variables declared at the class level are automatically storage variables. Create a variable to store the age of a person and another to store the number of cars that they own. Give age an initial value of your choosing, but don't make an assignment for cars;  
Reveal code  
  
  
  
  
  
Because the age of a person, or the number of cars they own, is unlikely to be greater than 255, we can use a uint8 for each of these. For types that are smaller than 32 bytes, multiple variables of the same type will be packed in the same storage slot. For this to work, the variables must be declared together.  
// These variables take advantage of packing  
uint8 first;  
uint8 second;  
uint third;  
  
// These variables DO NOT take advantage of packing and should be reordered  
uint8 fourth;  
uint fifth;  
uint8 sixth;  
Initializing a Value with the Constructor  
You may add a constructor function to your contract. Similar to other languages, this function is called exactly once, when the contract is deployed. The constructor may have parameters, but it does not require them.  
You can use the constructor to perform various setup tasks. For example, the constructor for the ERC-721 token that is the underlying mechanism for most NFTs uses the constructor to set up the name and symbol for the token.  
Create a constructor function and use it to assign the value of your choosing to cars.  
Reveal code  
  
  
  
Accessing State Variables  
Deploy your contract in Remix. It should work fine, but you'll have one problem: there isn't a way to see if the variables have the expected values!  
You could solve this by writing functions that return the values in your state variables, but you don't need to. The Solidity compiler automatically creates getters for all public variables.  
Add the public keyword to both variables. Unlike most languages, public goes after the type declaration. Your contract should now be similar to:  
Reveal code  
  
  
  
  
  
  
  
  
Redeploy your contract and test to confirm.  
Setting a State Variable with a Function  
Good news! Our user bought a second car! The only problem is that we don't have a way to update the number of cars stored.  
Add a Function to Update cars  
Before writing the function, let's think about design considerations for this feature. At any point in time, a user could:  
Buy or otherwise acquire a new car  
Get several new cars all at once (Woohoo!)  
Sell or give away one or more cars (😞)  
Given this wide variety of conditions, a good approach would be to handle calculating the correct number of cars on the front end, and passing the updated value to the back end.  
To meet this need, we can write a public function that takes a uint8 for \_numberOfCars and then simply assigns that value to the state variable cars. Because this function modifies state, it does not need pure or view. It isn't either of those.  
Reveal code  
  
  
  
  
Deploy and test to make sure it works as expected.  
DANGER  
While packing variables can save on gas costs, it can also increase them. The EVM operates on 32 bytes at a time, so it will take additional steps to reduce the size of the element for storage.  
Furthermore, the savings in writing to storage only apply when writing multiple values in the same slot at the same time.  
Review the Warning in the layout section of the docs for more details!  
Add a Function to Update age  
It would also be good to be able update the age value. This problem has slightly different considerations. Sadly, age will never go down. It should also probably only go up by one year for each update. The ++ operator works in Solidity, so we can use that to create a function that simple increments age when called.  
Reveal code  
  
  
  
  
But what if a user calls this function by mistake? Good point!  
On your own, add a function called adminSetAge that can set the age to a specified value.  
Refactor the Constructor to Accept Arguments  
We've got one problem remaining with this contract. What if your user has a different age or number of cars than what you've hardcoded into the contract?  
As mentioned above, the constructor can take arguments and use them during deployment. Let's refactor the contract to set the two state variables in the constructor based on provided values.  
Reveal code  
  
  
  
  
  
  
  
  
  
Redeploy your contract. Note that now you have added parameters to the constructor, you'll have to provide them during deployment.  
Once completed, your contract should be similar to:  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Conclusion  
In this lesson, you've explored how to persistently store values on the blockchain. You've also practiced updating them from functions. Finally, you've learned how to use the constructor to perform setup functionality during deployment, with and without parameters.  
ON THIS PAGE  
Simple Storage Contract  
Setting a State Variable with a Function  
Add a Function to Update cars  
Add a Function to Update age  
Refactor the Constructor to Accept Arguments  
Conclusion  
Add a Function to Update cars  
Add a Function to Update age  
Refactor the Constructor to Accept Arguments  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/storage/simple-storage-video  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Simple Storage  
Storing Data  
How Storage Works  
How Storage Works  
Storage Exercise  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Simple Storage  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/storage/storage-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Simple Storage  
Storing Data  
How Storage Works  
How Storage Works  
Storage Exercise  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Storage Exercise  
Create a contract that adheres to the following specifications:  
Contract  
Create a single contract called EmployeeStorage. It should not inherit from any other contracts. It should have the following functions:  
State Variables  
The contract should have the following state variables, optimized to minimize storage:  
A private variable shares storing the employee's number of shares owned  
Employees with more than 5,000 shares count as directors and are stored in another contract  
Public variable name which stores the employee's name  
A private variable salary storing the employee's salary  
Salaries range from 0 to 1,000,000 dollars  
A public variable idNumber storing the employee's ID number  
Employee numbers are not sequential, so this field should allow any number up to 2^256-1  
Constructor  
When deploying the contract, utilize the constructor to set:  
shares  
name  
salary  
idNumber  
For the purposes of the test, you must deploy the contract with the following values:  
shares - 1000  
name - Pat  
salary - 50000  
idNumber - 112358132134  
View Salary and View Shares  
DANGER  
In the world of blockchain, nothing is ever secret!\* private variables prevent other contracts from reading the value. You should use them as a part of clean programming practices, but marking a variable as private does not hide the value. All data is trivially available to anyone who knows how to fetch data from the chain.  
\*You can make clever use of encryption though!  
Write a function called viewSalary that returns the value in salary.  
Write a function called viewShares that returns the value in shares.  
Grant Shares  
Add a public function called grantShares that increases the number of shares allocated to an employee by \_newShares. It should:  
Add the provided number of shares to the shares  
If this would result in more than 5000 shares, revert with a custom error called TooManyShares that returns the number of shares the employee would have with the new amount added  
If the number of \_newShares is greater than 5000, revert with a string message, "Too many shares"  
Check for Packing and Debug Reset Shares  
Add the following function to your contract exactly as written below.  
/\*\*  
\* Do not modify this function. It is used to enable the unit test for this pin  
\* to check whether or not you have configured your storage variables to make  
\* use of packing.  
\*  
\* If you wish to cheat, simply modify this function to always return `0`  
\* I'm not your boss ¯\\_(ツ)\_/¯  
\*  
\* Fair warning though, if you do cheat, it will be on the blockchain having been  
\* deployed by your wallet....FOREVER!  
\*/  
function checkForPacking(uint \_slot) public view returns (uint r) {  
 assembly {  
 r := sload (\_slot)  
 }  
}  
  
/\*\*  
\* Warning: Anyone can use this function at any time!  
\*/  
function debugResetShares() public {  
 shares = 1000;  
}  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connecting...  
ON THIS PAGE  
Contract  
State Variables  
Constructor  
View Salary and View Shares  
Grant Shares  
Check for Packing and Debug Reset Shares  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/structs/structs-exercise  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Structs  
Structs  
Structs Exercise  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Structs Exercise  
Create a contract that adheres to the following specifications.  
Contract  
Create a contract called GarageManager. Add the following in storage:  
A public mapping called garage to store a list of Cars (described below), indexed by address  
Add the following types and functions.  
Car Struct  
Implement a struct called Car. It should store the following properties:  
make  
model  
color  
numberOfDoors  
Add Car Garage  
Add a function called addCar that adds a car to the user's collection in the garage. It should:  
Use msg.sender to determine the owner  
Accept arguments for make, model, color, and number of doors, and use those to create a new instance of Car  
Add that Car to the garage under the user's address  
Get All Cars for the Calling User  
Add a function called getMyCars. It should return an array with all of the cars owned by the calling user.  
Get All Cars for Any User  
Add a function called getUserCars. It should return an array with all of the cars for any given address.  
Update Car  
Add a function called updateCar. It should accept a uint for the index of the car to be updated, and arguments for all of the Car types.  
If the sender doesn't have a car at that index, it should revert with a custom error BadCarIndex and the index provided.  
Otherwise, it should update that entry to the new properties.  
Reset My Garage  
Add a public function called resetMyGarage. It should delete the entry in garage for the sender.  
Submit your Contract and Earn an NFT Badge! (BETA)  
INFO  
Hey, where'd my NFT go!?  
Testnets are not permanent! Base Goerli will soon be sunset, in favor of Base Sepolia.  
As these are separate networks with separate data, your NFTs will not transfer over.  
Don't worry! We've captured the addresses of all NFT owners on Base Goerli and will include them when we release the mechanism to transfer these NFTs to mainnet later this year! You can also redeploy on Sepolia and resubmit if you'd like!  
Connect Wallet  
Please connect your wallet.  
If you need a wallet, you can get the Coinbase Wallet here.  
You can also use MetaMask and other popular wallets.  
  
⚠️ Spoiler Alert: Open only if tests fail  
ON THIS PAGE  
Contract  
Car Struct  
Add Car Garage  
Get All Cars for the Calling User  
Get All Cars for Any User  
Update Car  
Reset My Garage  
Submit your Contract and Earn an NFT Badge! (BETA)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/structs/structs-sbs  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Structs  
Structs  
Structs Exercise  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Structs  
The struct type allows you to organize related data of different types.  
Objectives  
By the end of this lesson you should be able to:  
Construct a struct (user-defined type) that contains several different data types  
Declare members of the struct to maximize storage efficiency  
Describe constraints related to the assignment of structs depending on the types they contain  
Creating a Struct  
In the last exercise, we used a mapping to create a relationship between an address and a uint. But what if your users have favorite colors too? Or favorite cars? You could create a mapping for each of these, but it would quickly get awkward. Instead, a struct can be used to create a custom type that can store all of a user's favorites within one data type.  
Create a new contract called Structs.  
Setting up the Struct  
Instantiate a struct with the keyword, followed by a name for the type, curly brackets, and the variables that make up the type. Add a stub for Favorites:  
struct Favorites {  
  
}  
After consulting with the designers, we need to store the following for each address's favorites:  
Favorite number  
Birth Day of Month  
Favorite color  
Lucky Lottery numbers  
Let's pause for a moment and do some technical design around how to save our favorites.  
The product team has confirmed for us that we can safely expect that no users have a favorite number greater than 65,536, and of course, everyone is born on a day of the month between 1-31.  
Variable packing also works inside structs, so we could potentially save on storage by using smaller uints for those variables. However, people don't change their favorite number very often, and the day of the month that they were born on never changes.  
Therefore, it's probably more gas-efficient and less cumbersome to write other parts of the code, if we just use uint for both variables.  
Favorite color can be a string.  
For Lucky Lottery Numbers, we need a collection. We could use a dynamic array, since this will be in storage, but we already know that the lottery has 5 numbers.  
Try to use this information to build the struct on your own. You should end up with something similar to:  
Reveal code  
  
  
  
  
  
  
  
Instantiating a Struct with Its Name  
There are two ways to instantiate a struct using its name. The first is similar to instantiating a new object in JavaScript:  
Favorites memory myFavorites = Favorites({  
 favoriteNumber: 29,  
 birthDay: 14,  
 favoriteColor: "red",  
 lotteryNumbers: [uint(1), 2, 3, 4, 5]  
});  
You can also use a shorthand method where you skip the member names and just list a value for each one. Note that the curly brackets are not included in this format:  
Favorites memory myFavorites = Favorites(  
 29,  
 14,  
 "red",  
 [uint(1), 2, 3, 4, 5]  
);  
There's no difference in gas costs with either of these methods. Use the one that makes the most sense for the given situation.  
Saving Multiple Instances to Storage  
Next, we need to figure out the best way to organize the Favorites in storage. There are a few options, as always, each with tradeoffs. You could match the pattern you used for favorite numbers and utilize a mapping to match addresses to Favorites.  
Another popular method is to use an array, which takes advantage of .push returning a reference to the newly added element, and the fact that the concept of undefined does not exist in Solidity.  
First, instantiate an array of Favorites:  
Favorites[] public userFavorites;  
Next, add a public function to add submitted favorites to the list. It should take each of the members as an argument. Then, assign each argument to the new element via the reference returned by push().  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Alternatively, you can create an instance in memory, then push it to storage.  
Reveal code  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
The gas cost is similar for each of these methods.  
Unexpected Behavior in Structs  
Structs in Solidity exhibit some properties that are unexpected, or even frustrating. Working with them often includes untangling a set of mutually-exclusive properties and needs.  
Dynamic Storage Arrays in Structs  
The product team has contacted you to let you know that the beta testers are complaining about the lotteryNumbers. As it turns out, not every locality has lotteries where 5 numbers are drawn. Some have 3, 4, or even 6!.  
You might think this is an easy enough change. After all, you can just remove the size from the array declaration inside Favorites. Go ahead and try it:  
struct Favorites {  
 uint favoriteNumber;  
 uint birthDay;  
 string favoriteColor;  
 uint[] lotteryNumbers; // Removed the '5'  
}  
You'll get an error if you're using the memory method shown above.  
from solidity:  
TypeError: Invalid type for argument in function call. Invalid implicit conversion from uint256[5] memory to uint256[] memory requested.  
 --> contracts/mappings\_exercise.sol:70:13:  
 |  
70 | [uint(1), 2, 3, 4, 5]  
 | ^^^^^^^^^^^^^^^^^^^^^  
The simplest resolution here is to switch back to using push() to create an empty instance of Favorites, then assigning the values.  
The reason this works is a little obtuse. In the failing example, an unsized uint array is the expected type for the argument, but a sized uint array is provided. Solidity cannot perform implicit conversions like this most of the time and you'll get a compiler error if you provide the wrong type for an argument, even if it is convertible.  
One exception to this rule is that Solidity can perform an implicit conversion during assignment if the variable on the right side "fits" into the variable on the left side.  
uint[5] fits in uint[], so Solidity will allow it to sit 🐈.  
But what happens if you use the getter for userFavorites to retrieve your entry?  
{  
 "0": "uint256: favoriteNumber 29",  
 "1": "uint256: birthDay 14",  
 "2": "string: favoriteColor red"  
}  
What happened to the array? It's not there, and it turns out that this is on purpose.  
Mappings Inside of Structs  
You may add mappings inside of structs, subject to a few quirks and restrictions. Add mapping (uint => uint) numberPairs; to Favorites.  
In addFavorites, assign newFavorite.numberPairs[33] = 66;  
Deploy and test. So far, so good!  
Déjà vu ahead: But what happens if you use the getter for userFavorites to retrieve your entry?  
{  
 "0": "uint256: favoriteNumber 29",  
 "1": "uint256: birthDay 14",  
 "2": "string: favoriteColor red"  
}  
It's not there, and it turns out that this is on purpose.  
Another issue emerges if you try to return the struct from a public function. What if you wanted your addFavorite function to return a reference to the new favorite?  
// Bad code example, will not work  
function addFavorite(  
 uint \_favoriteNumber,  
 uint \_birthDay,  
 string calldata \_favoriteColor,  
 uint[] calldata \_lotteryNumbers  
) public returns (newFavorite memory) {  
 // .push() returns a reference to the new element  
 Favorites storage newFavorite = userFavorites.push();  
 newFavorite.favoriteNumber = \_favoriteNumber;  
 newFavorite.birthDay = \_birthDay;  
 newFavorite.favoriteColor = \_favoriteColor;  
 newFavorite.lotteryNumbers = \_lotteryNumbers;  
 newFavorite.numberPairs[33] = 66;  
  
 return newFavorite;  
}  
You'll get an error. The mapping type cannot be returned by a public or external function, so neither can a struct that contains one.  
from solidity:  
TypeError: Types containing (nested) mappings can only be parameters or return variables of internal or library functions.  
 --> contracts/mappings\_exercise.sol:64:23:  
 |  
64 | ) public returns (Favorites memory) {  
 | ^^^^^^^^^^^^^^^^  
Finally, what happens if you try to assign newFavorite to a memory variable? Again, an error occurs because mappings can only be in storage.  
// Bad code example, will not work  
Favorites memory secondFavorite = newFavorite;  
from solidity:  
TypeError: Type struct Structs.Favorites memory is only valid in storage because it contains a (nested) mapping.  
 --> contracts/mappings\_exercise.sol:82:9:  
 |  
82 | Favorites memory secondFavorite = newFavorite;  
 | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
Automatic Getters for Public Structs  
As with other types, if you put a public struct in storage at the contract level, the compiler will generate a getter automatically. However, these don't work quite the way you might expect. For example, imagine:  
  
struct MyStruct {  
 uint first;  
 uint second;  
 uint third;  
}  
  
MyStruct myStruct;  
  
The automatic getter for myStruct will not be:  
// Approximate example, not real code  
function myStruct() public view returns (MyStruct memory) {  
 return myStruct;  
}  
Instead, it returns the members individually:  
// Approximate example, not real code  
function myStruct() public view returns (uint, uint, uint) {  
 return (myStruct.first, myStruct.second, myStruct.third);  
}  
Create your own getter to return the data as a tuple, which will be interpreted as the appropriate type if it's called from another contract via an interface.  
function getMyStruct() public view returns (MyStruct memory) {  
 return myStruct;  
}  
Conclusion  
In this lesson, you've learned how to use the struct keyword to create a custom type that stores related data. You've also learned three methods of instantiating them and common patterns for storing structs in storage. Finally, you've explored some of the constraints that emerge when working with more complex data types within a struct.  
ON THIS PAGE  
Creating a Struct  
Unexpected Behavior in Structs  
Dynamic Storage Arrays in Structs  
Mappings Inside of Structs  
Automatic Getters for Public Structs  
Conclusion  
Dynamic Storage Arrays in Structs  
Mappings Inside of Structs  
Automatic Getters for Public Structs  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/structs/structs-vid  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Overview  
Anatomy of a Smart Contract  
Introduction to Solidity  
Contracts and Basic Functions  
Deploying to a Testnet  
Control Structures  
Storage in Solidity  
Arrays in Solidity  
The Mapping Type  
Advanced Functions  
Structs  
Structs  
Structs  
Structs Exercise  
Inheritance  
Imports  
Errors  
The `new` Keyword  
Contract to Contract Interactions  
Events  
Address and Payable  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Structs  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/welcome  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Exercise Contracts  
Learn to Build Smart Contracts and Onchain Apps  
Introduction  
Welcome to Base Learn, your guide to learning smart contract development. Base Learn's curriculum has been expertly crafted to equip you with the skills and knowledge needed to build and deploy smart contracts on Base, or any EVM-compatible chain, including Ethereum, Optimism, and many more. Plus, you'll be eligible to earn NFTs as you complete each module, showcasing your mastery of the material.  
Whether you're a curious novice or a seasoned pro looking to stay ahead of the game, our dynamic lessons cater to all levels of experience. You can start with the basics and work your way up, or dive straight into the more advanced concepts and push your limits to new heights.  
Begin your journey today!  
What you can learn in this program  
Base Learn covers the following topics. If you're looking for quickstarts, or deeper guides on advanced topics, check out our Base Builder Tutorials!  
Ethereum Applications  
Describe the origin and goals of the Ethereum blockchain  
List common types of applications that can be developed with the Ethereum blockchain  
Compare and contrast Web2 vs. Web3 development  
Compare and contrast the concept of "ownership" in Web2 vs. Web3  
Gas Use in Ethereum Transactions  
Explain what gas is in Ethereum  
Explain why gas is necessary in Ethereum  
Understand how gas works in Ethereum transactions  
EVM Diagram  
Diagram the EVM  
Setup and Overview  
Install and create a new Hardhat project with Typescript support  
Describe the organization and folder structure of a Hardhat project  
List the use and properties of hardhat.config.ts  
Testing with Hardhat and Typechain  
Set up TypeChain to enable testing  
Write unit tests for smart contracts using Mocha, Chai, and the Hardhat Toolkit  
Set up multiple signers and call smart contract functions with different signers  
Etherscan  
List some of the features of Etherscan  
Read data from the Bored Apes Yacht Club contract on Etherscan  
Write data to a contract using Etherscan.  
Deploying Smart Contracts  
Deploy a smart contract to the Base Sepolia Testnet with hardhat-deploy  
Deploy a smart contract to the Sepolia Testnet with hardhat-deploy  
Use BaseScan to view a deployed smart contract  
Verifying Smart Contracts  
Verify a deployed smart contract on Etherscan  
Connect a wallet to a contract in Etherscan  
Use etherscan to interact with your own deployed contract  
Hardhat Forking  
Use Hardhat Network to create a local fork of mainnet and deploy a contract to it  
Utilize Hardhat forking features to configure the fork for several use cases  
'Introduction to Remix'  
List the features, pros, and cons of using Remix as an IDE  
Deploy and test the Storage.sol demo contract in Remix  
Deployment in Remix  
Deploy and test the Storage.sol demo contract in Remix  
Hello World  
Construct a simple "Hello World" contract  
List the major differences between data types in Solidity as compared to other languages  
Select the appropriate visibility for a function  
Basic Types  
Categorize basic data types  
List the major differences between data types in Solidity as compared to other languages  
Compare and contrast signed and unsigned integers  
Test Networks  
Describe the uses and properties of the Base testnet  
Compare and contrast Ropsten, Rinkeby, Goerli, and Sepolia  
Deployment to Base Sepolia  
Deploy a contract to the Base Sepolia testnet and interact with it in [BaseScan]  
Contract Verification  
Verify a contract on the Base Sepolia testnet and interact with it in [BaseScan]  
Control Structures  
Control code flow with if, else, while, and for  
List the unique constraints for control flow in Solidity  
Utilize require to write a function that can only be used when a variable is set to true  
Write a revert statement to abort execution of a function in a specific state  
Utilize error to control flow more efficiently than with require  
Storing Data  
Use the constructor to initialize a variable  
Access the data in a public variable with the automatically generated getter  
Order variable declarations to use storage efficiently  
How Storage Works  
Diagram how a contract's data is stored on the blockchain (Contract -> Blockchain)  
Order variable declarations to use storage efficiently  
Diagram how variables in a contract are stored (Variable -> Contract)  
Arrays  
Describe the difference between storage, memory, and calldata arrays  
Filtering an Array  
Write a function that can return a filtered subset of an array  
Mappings  
Construct a Map (dictionary) data type  
Recall that assignment of the Map data type is not as flexible as for other data types/in other languages  
Restrict function calls with the msg.sender global variable  
Recall that there is no collision protection in the EVM and why this is (probably) ok  
Function Visibility and State Mutability  
Categorize functions as public, private, internal, or external based on their usage  
Describe how pure and view functions are different than functions that modify storage  
Function Modifiers  
Use modifiers to efficiently add functionality to multiple functions  
Structs  
Construct a struct (user-defined type) that contains several different data types  
Declare members of the struct to maximize storage efficiency  
Describe constraints related to the assignment of structs depending on the types they contain  
Inheritance  
Write a smart contract that inherits from another contract  
Describe the impact inheritance has on the byte code size limit  
Multiple Inheritance  
Write a smart contract that inherits from multiple contracts  
Abstract Contracts  
Use the virtual, override, and abstract keywords to create and use an abstract contract  
Imports  
Import and use code from another file  
Utilize OpenZeppelin contracts within Remix  
Error Triage  
Debug common solidity errors including transaction reverted, out of gas, stack overflow, value overflow/underflow, index out of range, etc.  
The New Keyword  
Write a contract that creates a new contract with the new keyword  
'Contract to Contract Interaction'  
Use interfaces to allow a smart contract to call functions in another smart contract  
Use the call() function to interact with another contract without using an interface  
Events  
Write and trigger an event  
List common uses of events  
Understand events vs. smart contract storage  
Address and Payable in Solidity  
Differentiate between address and address payable types in Solidity  
Determine when to use each type appropriately in contract development  
Employ address payable to send Ether and interact with payable functions  
Minimal Token  
Construct a minimal token and deploy to testnet  
Identify the properties that make a token a token  
The ERC-20 Token Standard  
Analyze the anatomy of an ERC-20 token  
Review the formal specification for ERC-20  
ERC-20 Implementation  
Describe OpenZepplin  
Import the OpenZepplin ERC-20 implementation  
Describe the difference between the ERC-20 standard and OpenZeppelin's ERC20.sol  
Build and deploy an ERC-20 compliant token  
The ERC-721 Token Standard  
Analyze the anatomy of an ERC-721 token  
Compare and contrast the technical specifications of ERC-20 and ERC-721  
Review the formal specification for ERC-721  
ERC-721 Token  
Analyze the anatomy of an ERC-721 token  
Compare and contrast the technical specifications of ERC-20 and ERC-721  
Review the formal specification for ERC-721  
Build and deploy an ERC-721 compliant token  
Use an ERC-721 token to control ownership of another data structure  
Wallet Connectors  
Identify the role of a wallet aggregator in an onchain app  
Debate the pros and cons of using a template  
Scaffold a new onchain app with RainbowKit  
Support users of EOAs and the Coinbase Smart Wallet with the same app  
Building an Onchain App  
Identify the role of a wallet aggregator in an onchain app  
Debate the pros and cons of using a template  
Add a wallet connection to a standard template app  
The useAccount Hook  
Implement the `useAccount`` hook to show the user's address, connection state, network, and balance  
Implement an isMounted hook to prevent hydration errors  
The useReadContract Hook  
Implement wagmi's useReadContract hook to fetch data from a smart contract  
Convert data fetched from a smart contract to information displayed to the user  
Identify the caveats of reading data from automatically-generated getters  
Configuring useReadContract  
Use useBlockNumber and the queryClient to automatically fetch updates from the blockchain  
Describe the costs of using the above, and methods to reduce those costs  
Configure arguments to be passed with a call to a pure or view smart contract function  
Call an instance of useReadContract on demand  
Utilize isLoading and isFetching to improve user experience  
The useWriteContract hook  
Implement wagmi's useWriteContract hook to send transactions to a smart contract  
Configure the options in useWriteContract  
Display the execution, success, or failure of a function with button state changes, and data display  
The useSimulateContract hook  
Implement wagmi's useSimulateContract and useWriteContract to send transactions to a smart contract  
Configure the options in useSimulateContract and useWriteContract  
Call a smart contract function on-demand using the write function from useWriteContract, with arguments and a value  
ON THIS PAGE  
What you can learn in this program  
What you can learn in this program  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/writing-to-contracts/useSimulateContract  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Connecting to the Blockchain  
Reading and Displaying Data  
Writing to Contracts  
The `useWriteContract` hook  
The `useSimulateContract` hook  
Exercise Contracts  
The `useSimulateContract` hook  
The useSimulateContract hook simulates and validates a contract interaction without actually sending a transaction to the blockchain. Using it allows you to detect and respond to potential errors before the user tries to send a transaction.  
Objectives  
By the end of this guide you should be able to:  
Implement wagmi's useSimulateContract and useWriteContract to send transactions to a smart contract  
Configure the options in useSimulateContract and useWriteContract  
Call a smart contract function on-demand using the write function from useWriteContract, with arguments and a value  
Refining the Claim Component  
In the previous step-by-step, you used useWriteContract to set up a hook you can use to call the claim function in your smart contract when the user clicks a button. The component works well enough, but it can take a long time for the wallet to pop up, particularly if there is network congestion. You also have no way of responding to a problem with the transaction inputs until after the user tries to initiate a transaction.  
Using useSimulateContract  
The useSimulateContract can be used in partnership with useWriteContract. To do so, you set up the transaction parameters in useSimulateContract, then use the data?.request returned by it as an argument in the call to write to the contract. Modify your TokenInfo component to test it:  
// Bad code for example. See below for fix.  
const {  
 data: claimData,  
 isFetching: claimIsFetching,  
 isError: claimIsError,  
} = useSimulateContract({  
 address: contractData.address as `0x${string}`,  
 abi: contractData.abi,  
 functionName: 'claim',  
});  
  
useEffect(() => {  
 if (claimIsError) {  
 alert('Unable to claim'); // TODO: Better error handling  
 }  
}, [claimIsError]);  
  
// No changes to `useWriteContract`  
const { writeContract: claim, isPending: claimIsPending } = useWriteContract();  
  
// Other code...  
  
// Update the call to `claim`  
const handleClaimClick = () => {  
 claim(claimData?.request);  
};  
You'll also need to update your handler to use the TypeScript pre-check feature, because the claim function will be briefly undefined.  
const handleClaimClick = () => {  
 claim(claimData!.request);  
};  
Reload the site and observe that the alert is triggered on load if you're signed in with an address that has already claimed tokens. You'll also see that the button is disabled, as though the user had clicked it and a transaction is loading in the wallet.  
Making Adjustments  
The reason for this is a subtle difference in how useWriteContract and useSimulateContract work.  
In the last step-by-step, you saw how viem runs a simulation of the transaction when the write function is called. useSimulateContract eagerly runs this simulation and updates it's variables.  
You'll need to make some modifications for it to work. The claimIsError variable is being triggered when the data for the call is simulated, not when the call has settled. As a result, it immediately generates the error, and triggers the alert without requiring the user to click the button.  
You can solve this a number of ways, including simply not rendering the button if the user has already claimed. You could also modify the code, and combine it with isError, to share this information to the user.  
const {  
 data: claimData,  
 isFetching: claimIsFetching,  
 isError: claimIsError,  
} = useSimulateContract({  
 address: contractData.address as `0x${string}`,  
 abi: contractData.abi,  
 functionName: 'claim',  
});  
  
// Deleted `useEffect` for `claimIsError`  
  
const { writeContract: claim, isPending: claimIsPending } = useWriteContract();  
  
// Other code  
  
return (  
 <div>  
 <p>{claimIsFetching.toString()}</p>  
 <p>{'Token Balance: ' + tokenBalance}</p>  
 <button disabled={claimIsPending || claimIsError} onClick={handleClaimClick}>  
 {claimIsPending ? 'Complete In Wallet' : 'Claim Tokens'}  
 </button>  
 <p>{claimIsError ? 'Unable to claim tokens.' : 'Claim your tokens!'} </p>  
 </div>  
);  
Conclusion  
In this step-by-step, you updated your app to use the useSimulateContract hook to provide a speedier wallet interaction for your users. You've also learned how you can predict and respond to potential errors without the user needing to attempt to send a transaction. You could use this functionality to let them know a username is already taken, a bid amount is not large enough, or an item is no longer available.  
ON THIS PAGE  
Refining the Claim Component  
Conclusion  
Using useSimulateContract  
Making Adjustments  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/docs/writing-to-contracts/useWriteContract  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Get Help on Discord  
Learn to Build Smart Contracts and Onchain Apps  
Introduction to Ethereum  
Development Tools  
Development with Hardhat  
Development With Foundry  
Smart Contract Development  
Token Development  
Hardhat Tools and Testing  
Onchain App Development (Frontend)  
Overview  
Frontend Setup  
Connecting to the Blockchain  
Reading and Displaying Data  
Writing to Contracts  
The `useWriteContract` hook  
The `useSimulateContract` hook  
Exercise Contracts  
The `useWriteContract` hook  
The useWriteContract hook allows you to call your public and external smart contract functions that write to state and create a permanent modification to the data on chain.  
Objectives  
By the end of this guide you should be able to:  
Implement wagmi's useWriteContract hook to send transactions to a smart contract  
Configure the options in useWriteContract  
Display the execution, success, or failure of a function with button state changes, and data display  
Sending a Transaction to the Blockchain  
DANGER  
In this step-by-step, you're going to start with the useWriteContract hook. You probably won't want to use this method in production. In the next step-by-step, we'll show you the [useSimulateContract] hook, how it works with useWriteContract, and how you can use it to create a better user experience.  
Exploring them separately will highlight the functionality provided by the prepare hook.  
CAUTION  
In this module, you'll extend the onchain app you build in the previous module, Reading and Displaying Data.  
You've built an app that can read from your Simple DAO smart contract, but so far, you've used BaseScan to send transactions that call your write functions. You can use the useWriteContract hook in a similar way to call those functions directly from your app.  
Setting up the Component  
Add a new component called TokenInfo to the project, and a state variable for tokenBalance.  
import { useState } from 'react';  
  
export function TokenInfo() {  
 const [tokenBalance, setTokenBalance] = useState(0);  
}  
Reading the Token Balance  
You'll need to know how many tokens the user has to be able to make decisions on what UI controls to display, so start by adding a useReadContract. You don't have a function for this directly in your contract, but your contract inherits from the OpenZeppelin ERC20 contract, which has a function called balanceOf that takes an address and returns the balance for that address.  
You'll need the user's address to use in args, which you can conveniently get from the useAccount hook using the pattern below.  
const { data: balanceData, queryKey: balanceQueryKey } =  
 useReadContract({  
 address: contractData.address as `0x${string}`,  
 abi: contractData.abi,  
 functionName: "balanceOf",  
 args: [useAccount().address],  
 });  
  
useEffect(() => {  
 if (balanceData) {  
 setTokenBalance(balanceData as number);  
 }  
}, [balanceData]);  
  
useEffect(() => {  
 queryClient.invalidateQueries({ queryKey: balanceQueryKey });  
}, [blockNumber, queryClient]);  
CAUTION  
Remember, this is an expensive method to watch for data to change on the blockchain. In this case, a more production-suitable solution might be to call balanceOf after the user has done something that might change the balance.  
Set the return for your component to display this balance to the user:  
return (  
 <div>  
 <p>{'Token Balance: ' + tokenBalance}</p>  
 </div>  
);  
Then, add the component to your app in index.tsx.  
return (  
 <div className={styles.container}>  
 <main className={styles.main}>  
 <ConnectButton />  
 <ConnectionWindow />  
 <TokenInfo />  
 <IssueList />  
 </main>  
);  
Run the app and make sure you see the expected balance displayed on the page.  
Setting up useWriteContract  
The useWriteContract hook is configured similarly to the useReadContract hook, with one important difference. You'll need to decompose the write property from the function call. This is a function that you can use to call your smart contract function whenever you'd like!  
const { writeContract: claim, isPending: claimIsPending } = useWriteContract();  
Add an event handler function and a button. As with the useReadContract hook, you can use isPending and other state helpers to adjust your UI. The name of this one is a little misleading. isPending will be true starting from the moment the transaction gets sent to the user's wallet.  
You can use this to nudge them to look to their wallet to complete the transaction. Additionally, add a useEffect to watch for an error state.  
const handleClaimClick = () => {  
 claim({  
 abi: contractData.abi,  
 address: contractData.address as `0x${string}`,  
 functionName: 'claim',  
 });  
};  
  
return (  
 <div>  
 <p>{'Token Balance: ' + tokenBalance}</p>  
 <button disabled={claimIsPending} onClick={handleClaimClick}>  
 {claimIsPending ? 'Complete In Wallet' : 'Claim Tokens'}  
 </button>  
 </div>  
);  
Try it out. Notice that the button text briefly changes without the wallet window popping up if you click the Claim Tokens button while connected with a wallet that already owns the tokens. The reason this happens is that viem, which underlies wagmi, runs a simulation of the transaction to estimate gas costs. If that simulation fails, it triggers the fail mechanism immediately, rather than allowing the app to send a bad transaction to the blockchain and cost the user gas for a call doomed to fail. You will fix this in the next tutorial.  
In the meantime, you'll need to change to a new wallet or redeploy your contract a couple of times to complete your testing. Do that, and try out the call on a wallet that hasn't claimed any tokens. Notice that the button is disabled and the text now prompts the user to look to their wallet to approve the transaction.  
Conclusion  
In this step-by-step, you've learned how to use the useWriteContract hook to call your smart contract functions on demand. You've also tested methods to manage the UI/UX experience for your users, based on the state of the transaction, as well as its success or failure.  
ON THIS PAGE  
Sending a Transaction to the Blockchain  
Conclusion  
Setting up the Component  
Reading the Token Balance  
Setting up useWriteContract  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/base-learn/hardhat-tools-and-testing/overview  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Overview of Hardhat Tools and Testing  
This series of guides shows you how to use a number of Hardhat plugins that will help you more effectively build and test your smart contracts.  
Learn how to keep your contracts under the 24 kiB limit, improve gas costs for your users, make sure your unit tests fully cover your code, and practice debugging.  
Objectives  
By the end of these guides, you should be able to:  
Profiling Size  
Use Hardhat Contract Sizer plugin to profile contract size  
Describe common strategies for managing the contract size limit  
Describe the impact that inheritance has on the byte code size limit  
Describe the impact that external contracts have on the byte code size limit  
Describe the impact of using libraries has on the byte code size limit  
Describe the impact of using the Solidity optimizer  
Profiling Gas  
Use the Hardhat Gas Reporter plugin to profile gas usage  
Describe common strategies for improving the gas usage of a contract  
Debugging  
Use console.log to write debugging logs  
List common errors and their resolutions  
Determine if an error is a contract error or an error in the test  
Test Coverage  
Use the Solidity Coverage plugin to analyze the coverage of your test suite  
Increase the coverage of your test suite  
Prerequisites  
1. Basic understanding of writing smart contracts  
These guides assume that you're reasonably comfortable writing basic smart contracts. If you're just getting started, jump over to our Base Learn guides and start learning!  
2. Familiarity with Hardhat  
We also assume that you've got Hardhat up and running, and can write unit tests for your smart contracts. If you're not there yet, but already know Solidity, you can setup Hardhat here.  
ON THIS PAGE  
Prerequisites  
1. Basic understanding of writing smart contracts  
2. Familiarity with Hardhat  
Debugging  
Test Coverage  
Prerequisites  
1. Basic understanding of writing smart contracts  
2. Familiarity with Hardhat  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/arbitration  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
APPENDIX 1: SEQUENCER, TESTNET, AND BASENAMES INTERFACE TERMS ARBITRATION AGREEMENT  
1. Applicability of Arbitration Agreement  
Subject to the terms of this Sequencer, Testnet, and Basenames Interface Terms Arbitration Agreement (as amended, restated, supplemented or otherwise modified and in effect from time to time, the “Arbitration Agreement”), you and Coinbase agree that any dispute, claim, or disagreement arising out of or relating in any way to your access to or use of Base, the Site, a Coinbase Sequencer, the Services, or the Base Terms (the “Base ToS”) and prior versions of the Base ToS, including claims and disputes that arose between us before the effective date of the Base ToS (each, a “Dispute”) will be resolved by binding arbitration, rather than in court, except that: (1) you and Coinbase may assert claims or seek relief in small claims court if such claims qualify and remain in small claims court; and (2) you and Coinbase must resolve in court any Dispute about infringement, misuse, or misappropriation of intellectual property rights (such as trademarks, trade dress, domain names, trade secrets, copyrights, and patents). For purposes of this Arbitration Agreement, “Dispute” will also include disputes that arose or involve facts occurring before the existence of this or any prior versions of the Base ToS as well as claims that may arise after the termination of the Base ToS. Any capitalized term used herein without definition shall have the meaning assigned thereto in the Base ToS.  
2. Waiver of Jury Trial  
YOU AND COINBASE HEREBY WAIVE ANY CONSTITUTIONAL AND STATUTORY RIGHTS TO SUE IN COURT AND HAVE A TRIAL IN FRONT OF A JUDGE OR A JURY. You and Coinbase are instead electing that all Disputes shall be resolved by arbitration under this Arbitration Agreement, except as specified in the section entitled “Applicability of Arbitration Agreement” above. There is no judge or jury in arbitration, and court review of an arbitration award is subject to very limited review.  
3. Waiver of Class and Other Non-Individualized Relief  
YOU AND COINBASE AGREE THAT, EXCEPT AS SPECIFIED IN SECTION 8, EACH OF US MAY BRING CLAIMS AGAINST THE OTHER ONLY ON AN INDIVIDUAL BASIS AND NOT ON A CLASS, REPRESENTATIVE, OR COLLECTIVE BASIS, AND THE PARTIES HEREBY WAIVE ALL RIGHTS TO HAVE ANY DISPUTE BE BROUGHT, HEARD, ADMINISTERED, RESOLVED, OR ARBITRATED ON A CLASS, COLLECTIVE, REPRESENTATIVE, OR MASS ACTION BASIS. ONLY INDIVIDUAL RELIEF IS AVAILABLE, AND DISPUTES OF MORE THAN ONE CUSTOMER OR USER CANNOT BE ARBITRATED OR CONSOLIDATED WITH THOSE OF ANY OTHER CUSTOMER OR USER. Subject to this Arbitration Agreement, the arbitrator may award declaratory or injunctive relief only in favor of the individual party seeking relief and only to the extent necessary to provide relief warranted by the party's individual claim. Nothing in this paragraph is intended to, nor shall it, affect the terms and conditions under Section 8 entitled “Batch Arbitration.” Notwithstanding anything to the contrary in this Arbitration Agreement, if a court decides by means of a final decision, not subject to any further appeal or recourse, that the limitations of this section, “Waiver of Class and Other Non-Individualized Relief,” are invalid or unenforceable as to a particular claim or request for relief (such as a request for public injunctive relief), you and Coinbase agree that that particular claim or request for relief (and only that particular claim or request for relief) shall be severed from the arbitration and may be litigated in the state or federal courts located in the State of California. All other Disputes that are not severed shall be litigated in small claims court or arbitrated. This section does not prevent you or Coinbase from participating in a class-wide settlement of claims.  
4. Rules and Forum  
The Base ToS evidences a transaction involving interstate commerce; and notwithstanding any other provision herein with respect to the applicable substantive law, the Federal Arbitration Act, 9 U.S.C. § 1 et seq., will govern the interpretation and enforcement of this Arbitration Agreement and any arbitration proceedings. The arbitration will be administered by the American Arbitration Association (“AAA”), in accordance with the Consumer Arbitration Rules (the "AAA Rules") then in effect, except as modified by this Arbitration Agreement. The AAA Rules are currently available at https://www.adr.org/sites/default/files/Consumer%20Rules.pdf.  
A party who wishes to initiate arbitration must provide the other party with a request for arbitration (the “Request”). If you initiate the arbitration, you must provide Coinbase a copy of your Request by email at [arbitration@coinbase.com] or through Coinbase’s registered agent for service of process. The Request must include: (1) the name, telephone number, mailing address, email address of the party seeking arbitration, and the username and wallet addresses (if any) associated with the applicable Account(s); (2) a statement of the legal claims being asserted and the factual bases of those claims; (3) a description of the remedy sought, including an accurate, good faith calculation of the amount in controversy in United States Dollars; (4) if you are the party making the Request, a statement certifying completion of the Formal Complaint Process as described in the Base ToS; and (5) evidence that the requesting party has paid any necessary filing fees in connection with such arbitration.  
If the party requesting arbitration is represented by counsel, the Request shall also include counsel’s name, telephone number, mailing address, and email address. Such counsel must also sign the Request. By signing the Request, counsel certifies to the best of counsel’s knowledge, information, and belief, formed after an inquiry reasonable under the circumstances, that: (1) the Request is not being presented for any improper purpose, such as to harass, cause unnecessary delay, or needlessly increase the cost of dispute resolution; (2) the claims, defenses and other legal contentions are warranted by existing law or by a nonfrivolous argument for extending, modifying, or reversing existing law or for establishing new law; and (3) the factual and damages contentions have evidentiary support or, if specifically so identified, will likely have evidentiary support after a reasonable opportunity for further investigation or discovery.  
Unless you and Coinbase otherwise agree, or the Batch Arbitration process discussed in Section 8 is triggered, the arbitration will be conducted in the county where you reside. Subject to the AAA Rules, the arbitrator may direct a limited and reasonable exchange of information between the parties, consistent with the expedited nature of the arbitration. If the AAA is not available to arbitrate, the parties will select an alternative arbitral forum. Your responsibility to pay any AAA fees and costs will be solely as set forth in the applicable AAA Rules.  
You and Coinbase agree that all materials and documents exchanged during the arbitration proceedings shall be kept confidential and shall not be shared with anyone except the parties’ attorneys, accountants, or business advisors, and then subject to the condition that they agree to keep all materials and documents exchanged during the arbitration proceedings confidential.  
5. Arbitrator  
The arbitrator will be either a retired judge or an attorney licensed to practice law in the state of California and will be selected by the parties from the AAA's roster of consumer dispute arbitrators. If the parties are unable to agree upon an arbitrator within thirty-five (35) business days of delivery of the Request, then the AAA will appoint the arbitrator in accordance with the AAA Rules, provided that if the Batch Arbitration process under Section 8 is triggered, the AAA will appoint the arbitrator for each batch.  
6. Authority of Arbitrator  
The arbitrator shall have exclusive authority to resolve any Dispute, including, without limitation, disputes arising out of or related to the interpretation or application of the Arbitration Agreement, including the enforceability, revocability, scope, or validity of the Arbitration Agreement or any portion of the Arbitration Agreement, except for the following: (1) all Disputes arising out of or relating to the section entitled “Waiver of Class and Other Non-Individualized Relief,” including any claim that all or part of the section entitled “Waiver of Class and Other Non-Individualized Relief” is unenforceable, illegal, void or voidable, or that such Section entitled “Waiver of Class and Other Non-Individualized Relief” has been breached, shall be decided by a court of competent jurisdiction and not by an arbitrator; (2) except as expressly contemplated in the section entitled “Batch Arbitration,” all Disputes about the payment of arbitration fees shall be decided only by a court of competent jurisdiction and not by an arbitrator; (3) all Disputes about whether either party has satisfied any condition precedent to arbitration shall be decided only by a court of competent jurisdiction and not by an arbitrator; (4) all Disputes about which version of the Arbitration Agreement applies shall be decided only by a court of competent jurisdiction and not by an arbitrator; and (5) all Disputes about whether a Dispute is carved out from arbitration in the Section above entitled “Applicability of Arbitration Agreement” shall be decided only by a court of competent jurisdiction and not by an arbitrator. The arbitration proceeding will not be consolidated with any other matters or joined with any other cases or parties, except as expressly provided in the section entitled “Batch Arbitration.” The arbitrator shall have the authority to grant motions dispositive of all or part of any Dispute. The arbitrator shall issue a written award and statement of decision describing the essential findings and conclusions on which the award is based, including the calculation of any damages awarded. The award of the arbitrator is final and binding upon you and us. This means that, among other things, you and we agree that an arbitral award shall have no preclusive effect in any other proceeding involving other parties. Judgment on the arbitration award may be entered in any court having jurisdiction. In any award of damages, the arbitrator shall abide by the “Limitation of Liability” section of the Terms.  
7. Attorneys’ Fees and Costs  
The parties shall bear their own attorneys’ fees and costs in arbitration unless the arbitrator finds that either the substance of the Dispute or the relief sought in the Request was frivolous or was brought for an improper purpose (as measured by the standards set forth in Federal Rule of Civil Procedure 11(b)). If you or Coinbase need to invoke the authority of a court of competent jurisdiction to compel arbitration, then the party that obtains an order compelling arbitration in such action shall have the right to collect from the other party its reasonable costs, necessary disbursements, and reasonable attorneys' fees incurred in securing an order compelling arbitration. The prevailing party in any court action relating to whether either party has satisfied any condition precedent to arbitration, including the Formal Complaint Process, is entitled to recover their reasonable costs, necessary disbursements, and reasonable attorneys’ fees and costs.  
8. Batch Arbitration  
To increase the efficiency of administration and resolution of arbitrations, you and Coinbase agree that in the event that there are one hundred (100) or more individual Requests of a substantially similar nature filed against Coinbase by or with the assistance of the same law firm, group of law firms, or organizations, within a thirty (30) day period (or as soon as possible thereafter), the AAA shall (1) administer the arbitration demands in batches of 100 Requests per batch (plus, to the extent there are less than 100 Requests left over after the batching described above, a final batch consisting of the remaining Requests); (2) appoint one arbitrator for each batch; and (3) provide for the resolution of each batch as a single consolidated arbitration with one set of filing and administrative fees due per side per batch, one procedural calendar, one hearing (if any) in a place to be determined by the arbitrator, and one final award (“Batch Arbitration”).  
All parties agree that Requests are of a “substantially similar nature” if they arise out of or relate to the same event or factual scenario and raise the same or similar legal issues and seek the same or similar relief. To the extent the parties disagree on the application of the Batch Arbitration process, the disagreeing party shall advise the AAA, and the AAA shall appoint a sole standing arbitrator to determine the applicability of the Batch Arbitration process (“Administrative Arbitrator”). In an effort to expedite resolution of any such dispute by the Administrative Arbitrator, the parties agree the Administrative Arbitrator may set forth such procedures as are necessary to resolve any disputes promptly. The Administrative Arbitrator’s fees shall be paid by Coinbase.  
You and Coinbase agree to cooperate in good faith with the AAA to implement the Batch Arbitration process including the payment of single filing and administrative fees for batches of Requests, as well as any steps to minimize the time and costs of arbitration, which may include: (1) the appointment of a discovery special master to assist the arbitrator in the resolution of discovery disputes; and (2) the adoption of an expedited calendar of the arbitration proceedings.  
This Batch Arbitration provision shall in no way be interpreted as authorizing a class, collective and/or mass arbitration or action of any kind, or arbitration involving joint or consolidated claims under any circumstances, except as expressly set forth in this provision.  
9. Modification  
If we make any updates to the Arbitration Agreement, we will make the updated terms available to you by publishing them on the Site. Your continued use of the Site and/or Services, including the acceptance of products and services offered on the Site following the posting of changes to this Arbitration Agreement constitutes your acceptance of any such changes.  
10. Severability  
If any provision of this Arbitration Agreement shall be determined to be invalid or unenforceable under any rule, law, or regulation of any local, state, or federal government agency, such provision will be changed and interpreted to accomplish the objectives of the provision to the greatest extent possible under any applicable law and the validity or enforceability of any other provision of this Arbitration Agreement shall not be affected.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/base-contracts  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Contract Addresses  
L2 Contract Addresses  
Base Mainnet  
Name Address  
WETH9 0x4200000000000000000000000000000000000006  
L2CrossDomainMessenger 0x4200000000000000000000000000000000000007  
L2StandardBridge 0x4200000000000000000000000000000000000010  
SequencerFeeVault 0x4200000000000000000000000000000000000011  
OptimismMintableERC20Factory 0xF10122D428B4bc8A9d050D06a2037259b4c4B83B  
GasPriceOracle 0x420000000000000000000000000000000000000F  
L1Block 0x4200000000000000000000000000000000000015  
L2ToL1MessagePasser 0x4200000000000000000000000000000000000016  
L2ERC721Bridge 0x4200000000000000000000000000000000000014  
OptimismMintableERC721Factory 0x4200000000000000000000000000000000000017  
ProxyAdmin 0x4200000000000000000000000000000000000018  
BaseFeeVault 0x4200000000000000000000000000000000000019  
L1FeeVault 0x420000000000000000000000000000000000001a  
EAS 0x4200000000000000000000000000000000000021  
EASSchemaRegistry 0x4200000000000000000000000000000000000020  
LegacyERC20ETH 0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000  
Base Testnet (Sepolia)  
Name Address  
WETH9 0x4200000000000000000000000000000000000006  
L2CrossDomainMessenger 0x4200000000000000000000000000000000000007  
L2StandardBridge 0x4200000000000000000000000000000000000010  
SequencerFeeVault 0x4200000000000000000000000000000000000011  
OptimismMintableERC20Factory 0x4200000000000000000000000000000000000012  
GasPriceOracle 0x420000000000000000000000000000000000000F  
L1Block 0x4200000000000000000000000000000000000015  
L2ToL1MessagePasser 0x4200000000000000000000000000000000000016  
L2ERC721Bridge 0x4200000000000000000000000000000000000014  
OptimismMintableERC721Factory 0x4200000000000000000000000000000000000017  
ProxyAdmin 0x4200000000000000000000000000000000000018  
BaseFeeVault 0x4200000000000000000000000000000000000019  
L1FeeVault 0x420000000000000000000000000000000000001a  
EAS 0x4200000000000000000000000000000000000021  
EASSchemaRegistry 0x4200000000000000000000000000000000000020  
LegacyERC20ETH 0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000  
\*L2 contract addresses are the same on both mainnet and testnet.  
L1 Contract Addresses  
Ethereum Mainnet  
Name Address  
AddressManager 0x8EfB6B5c4767B09Dc9AA6Af4eAA89F749522BaE2  
L1CrossDomainMessenger 0x866E82a600A1414e583f7F13623F1aC5d58b0Afa  
L1ERC721Bridge 0x608d94945A64503E642E6370Ec598e519a2C1E53  
L1StandardBridge 0x3154Cf16ccdb4C6d922629664174b904d80F2C35  
L2OutputOracle 0x56315b90c40730925ec5485cf004d835058518A0  
OptimismMintableERC20Factory 0x05cc379EBD9B30BbA19C6fA282AB29218EC61D84  
OptimismPortal 0x49048044D57e1C92A77f79988d21Fa8fAF74E97e  
ProxyAdmin 0x0475cBCAebd9CE8AfA5025828d5b98DFb67E059E  
SystemConfig 0x73a79Fab69143498Ed3712e519A88a918e1f4072  
SystemDictator 0x1fE3fdd1F0193Dd657C0a9AAC37314D6B479E557  
Unneeded contract addresses  
Certain contracts are mandatory according to the OP Stack SDK, despite not being utilized. For such contracts, you can simply assign the zero address:  
StateCommitmentChain  
CanonicalTransactionChain  
BondManager  
Ethereum Testnet (Sepolia)  
Name Address  
AddressManager 0x709c2B8ef4A9feFc629A8a2C1AF424Dc5BD6ad1B  
AnchorStateRegistryProxy 0x4C8BA32A5DAC2A720bb35CeDB51D6B067D104205  
DelayedWETHProxy 0x7698b262B7a534912c8366dD8a531672deEC634e  
DisputeGameFactoryProxy 0xd6E6dBf4F7EA0ac412fD8b65ED297e64BB7a06E1  
FaultDisputeGame 0x48F9F3190b7B5231cBf2aD1A1315AF7f6A554020  
L1CrossDomainMessenger 0xC34855F4De64F1840e5686e64278da901e261f20  
L1ERC721Bridge 0x21eFD066e581FA55Ef105170Cc04d74386a09190  
L1StandardBridge 0xfd0Bf71F60660E2f608ed56e1659C450eB113120  
L2OutputOracle 0x84457ca9D0163FbC4bbfe4Dfbb20ba46e48DF254  
MIPS 0xFF760A87E41144b336E29b6D4582427dEBdB6dee  
OptimismMintableERC20Factory 0xb1efB9650aD6d0CC1ed3Ac4a0B7f1D5732696D37  
OptimismPortal 0x49f53e41452C74589E85cA1677426Ba426459e85  
PermissionedDisputeGame 0x54966d5A42a812D0dAaDe1FA2321FF8b102d1ee1  
PreimageOracle 0x627F825CBd48c4102d36f287be71f4234426b9e4  
ProxyAdmin 0x0389E59Aa0a41E4A413Ae70f0008e76CAA34b1F3  
SystemConfig 0xf272670eb55e895584501d564AfEB048bEd26194  
Base Admin Addresses  
Base Mainnet  
Admin Role Address Type of Key  
Batch Sender 0x5050f69a9786f081509234f1a7f4684b5e5b76c9 EOA managed by Coinbase Technologies  
Batch Inbox 0xff00000000000000000000000000000000008453 EOA (with no known private key)  
Output Proposer 0x642229f238fb9de03374be34b0ed8d9de80752c5 EOA managed by Coinbase Technologies  
Proxy Admin Owner (L1) 0x7bB41C3008B3f03FE483B28b8DB90e19Cf07595c 2-of-2 Nested Gnosis Safe (signers below)  
L1 Nested Safe Signer (Coinbase) 0x9855054731540A48b28990B63DcF4f33d8AE46A1 Gnosis Safe  
L1 Nested Safe Signer (Optimism) 0x9BA6e03D8B90dE867373Db8cF1A58d2F7F006b3A Gnosis Safe  
Proxy Admin Owner (L2) 0x2304cb33d95999dc29f4cef1e35065e670a70050 2-of-2 Nested Gnosis Safe (signers below)  
L2 Nested Safe Signer (Coinbase) 0xd94e416cf2c7167608b2515b7e4102b41efff94f Gnosis Safe  
L2 Nested Safe Signer (Optimism) 0x28EDB11394eb271212ED66c08f2b7893C04C5D65 Gnosis Safe  
Challenger 0x6f8c5ba3f59ea3e76300e3becdc231d656017824 1-of-2 Smart contract  
System config owner 0x14536667Cd30e52C0b458BaACcB9faDA7046E056 Gnosis Safe  
Guardian 0x14536667Cd30e52C0b458BaACcB9faDA7046E056 Gnosis Safe  
Base Testnet (Sepolia)  
Admin Role Address Type of Key  
Batch Sender 0x6CDEbe940BC0F26850285cacA097C11c33103E47 EOA managed by Coinbase Technologies  
Batch Inbox 0xff00000000000000000000000000000000084532 EOA (with no known private key)  
Output Proposer 0x20044a0d104E9e788A0C984A2B7eAe615afD046b EOA managed by Coinbase Technologies  
Proxy Admin Owner (L1) 0x0fe884546476dDd290eC46318785046ef68a0BA9 Gnosis Safe  
Proxy Admin Owner (L2) 0x20f984546476ddd290ec46318785046ef68a1cba Gnosis Safe  
Challenger 0xDa3037Ff70Ac92CD867c683BD807e5A484857405 EOA managed by Coinbase Technologies  
System config owner 0x0fe884546476dDd290eC46318785046ef68a0BA9 Gnosis Safe  
Guardian 0xA9FF930151130fd19DA1F03E5077AFB7C78F8503 EOA managed by Coinbase Technologies  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/basenames-tutorial-using-wagmi  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Add Basenames to your onchain app  
Basenames is now live! But what exactly is it? Basenames allows users to register human-readable names for their addresses and serves as a foundational building block for onchain identity. Think of it as your favorite social media handle, but even bigger. Your Basename is multichain by default and yours forever—no platform can take it away from you (just make sure to pay your fee).  
Integrating Basenames into your onchain app enhances the user experience by masking complex wallet addresses. Just as domains simplify IP addresses, Basenames do the same for wallet addresses.  
This tutorial shows you how to display Basenames on behalf of your users. We'll walk through setting up the necessary files and configurations to interact with the Basenames ENS resolver directly. Let's begin!  
Objectives  
By the end of this tutorial, you should be able to:  
Understand how onchain identity works on the Base network  
Enable users to use their onchain identity in your app  
Pull metadata from your users' Basename profile  
Steps  
First, create a directory to store the ABI (Application Binary Interface) for the Basenames ENS resolver. The ABI will allow your project to interact with the smart contract that handles Basenames.  
In your project folder, run the following commands:  
mkdir abis  
cd abis  
touch L2ResolverAbi.ts  
This will create a new directory named abis and a file named L2ResolverAbi.ts within it.  
Next, add the following placeholder code to the L2ResolverAbi.ts file:  
src/abis/L2ResolverAbi.ts  
export default [  
 // ABI information goes here  
] as const;  
TIP  
You will need to replace the placeholder comment with the actual ABI. Here is the link to the full L2ResolverAbi.  
To interact with the Base blockchain, you will need to update the wagmi configuration. This will allow your project to connect to the Base network and use its features.  
Update your wagmi.ts file as follows:  
'use client';  
  
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';  
import { ReactNode } from 'react';  
import { http, createConfig, WagmiProvider } from 'wagmi';  
import { base } from 'wagmi/chains';  
  
export const config = createConfig({  
 chains: [base],  
 transports: {  
 [base.id]: http(),  
 },  
 ssr: true,  
});  
  
const queryClient = new QueryClient();  
  
export default function EthereumProviders({  
 children,  
}: {  
 children: ReactNode;  
}) {  
 return (  
 <WagmiProvider config={config}>  
 <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>  
 </WagmiProvider>  
 );  
}  
This code sets up your application to use the Base network, enabling the project to interact with the blockchain.  
Next, we'll create a new directory to house the functions that will resolve and interact with Basenames. These functions will be responsible for fetching Basename information from the blockchain.  
In your project folder, create the apis directory and add a basenames.tsx file:  
WHAT'S HAPPENING IN THE CODE?  
convertReverseNodeToBytes(): This function is creating the reverse node so we can look up a name given an address. Each address gets its own reverse record in our registry that's created in a deterministic way.  
You can see the implementation of convertReverseNodeToBytes() in the OnchainKit repo  
BasenameTextRecordKeys: Metadata (e.g., github, twitter, etc.) are stored as text records so we can look them up based on the enum key.  
src/apis/basenames.tsx  
import {  
 Address,  
 ContractFunctionParameters,  
 createPublicClient,  
 encodePacked,  
 http,  
 keccak256,  
 namehash,  
} from 'viem';  
import { base, mainnet } from 'viem/chains';  
import L2ResolverAbi from '@/abis/L2ResolverAbi';  
  
// Function to resolve a Basename  
export async function getBasename(address: Address) {  
 try {  
 const addressReverseNode = convertReverseNodeToBytes(address, base.id);  
 const basename = await baseClient.readContract({  
 abi: L2ResolverAbi,  
 address: BASENAME\_L2\_RESOLVER\_ADDRESS,  
 functionName: 'name',  
 args: [addressReverseNode],  
 });  
 if (basename) {  
 return basename as BaseName;  
 }  
 } catch (error) {  
 // Handle the error accordingly  
 console.error('Error resolving Basename:', error);  
 }  
}  
This code provides the foundation for resolving Basenames using the Base network.  
TIP  
You can find the complete implementation in the full basenames.tsx file.  
Now that the necessary functions are in place, you can implement the Basenames functionality in your app. For this example, we'll modify the page.tsx file to display Basename information on the server and client side.  
Here's how to set it up:  
src/app/page.tsx  
import {  
 BasenameTextRecordKeys,  
 getBasename,  
 getBasenameAvatar,  
 getBasenameTextRecord,  
} from '@/apis/basenames';  
import BasenameDetails from '@/components/BasenameDetails';  
import EthereumProviders from '@/contexts/EthereumProviders';  
import { useAccount } from 'wagmi';  
  
// shrek.base.eth  
  
const address = '0x8c8F1a1e1bFdb15E7ed562efc84e5A588E68aD73'; // const account = useAccount(); \n address = account?.address;  
  
async function fetchData() {  
 const basename = await getBasename(address);  
  
 if (basename === undefined) throw Error('failed to resolve address to name');  
  
 const avatar = await getBasenameAvatar(basename);  
  
 const description = await getBasenameTextRecord(  
 basename,  
 BasenameTextRecordKeys.Description  
 );  
  
 const twitter = await getBasenameTextRecord(  
 basename,  
 BasenameTextRecordKeys.Twitter  
 );  
  
 return {  
 basename,  
 avatar,  
 description,  
 twitter,  
 };  
}  
  
export default async function Home() {  
 const data = await fetchData();  
  
 return (  
 <EthereumProviders>  
 <main className='flex min-h-screen flex-col gap-12 p-24'>  
 <div className='mb-12'>  
 <h1 className='text-xl mb-4'>Server-side rendered:</h1>  
 <ul className='flex flex-col gap-4'>  
 <li className='flex flex-col gap-2'>  
 <span>Address</span>  
 <strong>{address}</strong>  
 </li>  
 <li className='flex flex-col gap-2'>  
 <span>Basename</span>  
 <strong>{data.basename}</strong>  
 </li>  
 <li className='flex flex-col gap-2'>  
 <span>Avatar</span>  
 <strong>  
 <img  
 src={data.avatar}  
 alt={data.basename}  
 width={100}  
 height={100}  
 />  
 </strong>  
 </li>  
 <li className='flex flex-col gap-2'>  
 <span>Description</span>  
 <strong>{data.description}</strong>  
 </li>  
 <li className='flex flex-col gap-2'>  
 <span>Twitter</span>  
 <strong>{data.twitter}</strong>  
 </li>  
 </ul>  
 </div>  
 <div>  
 <h1 className='text-xl mb-4'>Client-side rendered:</h1>  
 <BasenameDetails address={address} />  
 </div>  
 </main>  
 </EthereumProviders>  
 );  
}  
In this example, the Home component fetches Basename data and displays it in both server-side and client-side rendered sections. This allows your app to provide a seamless user experience, showing Basename details like the avatar, description, and associated Twitter handle.  
Conclusion  
Congratulations! You've successfully integrated Basenames into your project. By setting up the necessary ABI, configuring your wagmi project, and implementing custom functions to resolve and display Basenames, you've enhanced your app's user experience by making wallet addresses more user-friendly. Your users can now enjoy a personalized, recognizable onchain identity across the Base network. Keep exploring and building to unlock even more possibilities with Basenames!  
Tags:account abstraction  
ON THIS PAGE  
Steps  
Conclusion  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/basenames-tutorial-with-onchainkit  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Add Basenames to your wagmi/viem App using OnchainKit  
Basenames is now live! But what exactly is it? Basenames allows users to register human-readable names for their addresses and serves as a foundational building block for onchain identity. Think of it as your favorite social media handle, but even bigger. Your Basename is multichain by default and yours forever—no platform can take it away from you (just make sure to pay your fee).  
Integrating Basenames into your onchain app enhances the user experience by masking complex wallet addresses. Just as domains simplify IP addresses, Basenames do the same for wallet addresses.  
OnchainKit is a React component library designed to make building Onchain applications easier. In this tutorial, we'll use the <Identity/> component to resolve Basenames.  
This demo uses Coinbase Smart Wallet and Coinbase Wallet, but Basenames is supported across many other wallets.  
Objectives  
By the end of this tutorial, you should be able to:  
Understand how onchain identity works on the Base network  
Enable users to use their onchain identity in your app using OnchainKit  
If you're starting from scratch, you'll need to create a new wagmi project. If you already have an existing wagmi project, you can skip ahead to the section on installing OnchainKit.  
To create a new wagmi project using TypeScript and install the required dependencies, run the following command:  
bun create wagmi  
Next, you'll need to install OnchainKit. Run the following command:  
bun add @coinbase/onchainkit  
After adding OnchainKit, install all dependencies and start your development server with:  
bun install && bun run dev  
This command will install the necessary dependencies and start a development server.  
To follow along with the tutorial effectively, open your web browser and your IDE side by side. This setup will allow you to code and see the changes in real time.  
Update Wagmi config  
In this section, we will configure your wagmi project to support the Base blockchain by importing the necessary modules.  
Start by importing the base and baseSepolia chains into your wagmi config. Navigate to src/wagmi.ts and update the file as follows:  
wagmi.ts  
import { http, cookieStorage, createConfig, createStorage } from 'wagmi';  
import { base, baseSepolia } from 'wagmi/chains';  
import { coinbaseWallet, injected } from 'wagmi/connectors';  
  
export function getConfig() {  
 return createConfig({  
 chains: [base, baseSepolia],  
 connectors: [  
 injected(),  
 coinbaseWallet({  
 appName: 'Create Wagmi',  
 preference: 'smartWalletOnly',  
 }),  
 ],  
 storage: createStorage({  
 storage: cookieStorage,  
 }),  
 ssr: true,  
 transports: {  
 [base.id]: http(),  
 [baseSepolia.id]: http(),  
 },  
 });  
}  
  
declare module 'wagmi' {  
 interface Register {  
 config: ReturnType<typeof getConfig>;  
 }  
}  
This configuration sets up the wagmi project to connect to the Base and BaseSepolia networks, utilizing Coinbase Wallet and other connectors.  
Now we’ll create a component to display the Basenames associated with an address.  
USE BASE AS YOUR CHAIN  
Ensure Chain is Set to Base Be sure to set the chain={base} parameter; otherwise, it will default to ENS (Ethereum Name Service).  
src/components/basename.tsx  
'use client';  
import React from 'react';  
('use client');  
import React from 'react';  
import { Avatar, Identity, Name, Address } from '@coinbase/onchainkit/identity';  
import { base } from 'viem/chains';  
  
interface DisplayBasenameProps {  
 address: `0x${string}` | undefined;  
}  
  
export function Basenames({ address }: DisplayBasenameProps) {  
 return (  
 <Identity  
 address={address}  
 chain={base}  
 schemaId="0xf8b05c79f090979bf4a80270aba232dff11a10d9ca55c4f88de95317970f0de9"  
 >  
 <Avatar address={address} chain={base} />  
 <Name address={address} chain={base} />  
 <Address />  
 </Identity>  
 );  
}  
This component uses OnchainKit to fetch and display the Basename, Avatar, and Address associated with the provided address.  
STYLE THE AVATAR  
Remember to style the Avatar component to fit your application's design.  
Next, we’ll integrate the newly created Basenames component into your application. For instance, if you have a Header component with a Login button that allows users to connect their wallet, you can display the user's Basename instead of their wallet address after they connect.  
Here’s an example of how to modify your header component to include the Basenames component:  
src/app/page.tsx  
'use client';  
import Footer from 'src/components/Footer';  
import { ONCHAINKIT\_LINK } from 'src/links';  
import { useAccount } from 'wagmi';  
import LoginButton from '../components/LoginButton';  
import { Basenames } from '../components/basename';  
  
export default async function Page() {  
 const { address } = useAccount();  
 const account = useAccount();  
  
 return (  
 <div className="flex h-full w-96 max-w-full flex-col px-1 md:w-[1008px]">  
 <section className="mb-6 mt-6 flex w-full flex-col md:flex-row">  
 <div className="flex w-full flex-row items-center justify-between gap-2 md:gap-0">  
 <a href={SUPERCOOL\_APP\_LINK} title="Supercool App" target="\_blank" rel="noreferrer">  
 <h1 className="text-xl font-normal not-italic tracking-[-1.2px] text-indigo-600">  
 Supercool App  
 </h1>  
 </a>  
 <div className="flex items-center gap-3">  
 {!address && <LoginButton />} {/\*Connect wallet button\*/}  
 {account.status === 'connected' && (  
 <div>  
 <Basenames address={account.addresses?.[0]} />  
 </div>  
 )}  
 {/\* <h1>{result}</h1> \*/}  
 </div>  
 </div>  
 </section>  
 <section className="templateSection flex w-full flex-col items-center justify-center gap-4 rounded-xl bg-gray-100 px-2 py-4 md:grow"></section>  
 <Footer />  
 </div>  
 );  
}  
In this example, once the user connects their wallet, the Basenames component is conditionally rendered to display their Basename instead of the raw wallet address.  
Conclusion  
That’s it! You’ve successfully integrated Basenames into your wagmi/viem app using OnchainKit. Test your application by connecting a wallet and observing how the Basename is displayed instead of the wallet address.  
Tags:account abstraction  
ON THIS PAGE  
Conclusion  
Update Wagmi config  
Conclusion  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/contracts  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Contracts  
This page lists contract addresses for onchain apps that we have deployed.  
Base Mainnet  
Multicall3  
Contract Address  
Multicall3 0xcA11bde05977b3631167028862bE2a173976CA11  
Uniswap v3  
Contract Address  
Permit2 0x000000000022D473030F116dDEE9F6B43aC78BA3  
universal router 0x198EF79F1F515F02dFE9e3115eD9fC07183f02fC  
v3CoreFactory 0x33128a8fC17869897dcE68Ed026d694621f6FDfD  
multicall 0x091e99cb1C49331a94dD62755D168E941AbD0693  
proxyAdmin 0x3334d83e224aF5ef9C2E7DDA7c7C98Efd9621fA9  
tickLens 0x0CdeE061c75D43c82520eD998C23ac2991c9ac6d  
nftDescriptor 0xF9d1077fd35670d4ACbD27af82652a8d84577d9F  
nonfungibleTokenPositionDescriptor 0x4f225937EDc33EFD6109c4ceF7b560B2D6401009  
descriptorProxy 0x4615C383F85D0a2BbED973d83ccecf5CB7121463  
nonfungibleTokenPositionManager 0x03a520b32C04BF3bEEf7BEb72E919cf822Ed34f1  
v3Migrator 0x23cF10b1ee3AdfCA73B0eF17C07F7577e7ACd2d7  
v3Staker 0x42bE4D6527829FeFA1493e1fb9F3676d2425C3C1  
quoterV2 0x3d4e44Eb1374240CE5F1B871ab261CD16335B76a  
swapRouter 0x2626664c2603336E57B271c5C0b26F421741e481  
Uniswap v2  
Contract Address  
Factory 0x8909Dc15e40173Ff4699343b6eB8132c65e18eC6  
Router 0x4752ba5dbc23f44d87826276bf6fd6b1c372ad24  
Base Testnet (Sepolia)  
Multicall3  
Contract Address  
Multicall3 0xcA11bde05977b3631167028862bE2a173976CA11  
Uniswap v3  
Contract Address  
Permit2 0x000000000022d473030f116ddee9f6b43ac78ba3  
universal router 0x050E797f3625EC8785265e1d9BDd4799b97528A1  
v3CoreFactory 0x4752ba5DBc23f44D87826276BF6Fd6b1C372aD24  
multicall 0xd867e273eAbD6c853fCd0Ca0bFB6a3aE6491d2C1  
proxyAdmin 0xD7303474Baca835743B54D73799688990f24a79D  
tickLens 0xedf6066a2b290C185783862C7F4776A2C8077AD1  
nftDescriptor 0x4e0caFF1Df1cCd7CF782FDdeD77f020699B57f1a  
nonfungibleTokenPositionDescriptor 0xd7c6e867591608D32Fe476d0DbDc95d0cf584c8F  
nonfungibleTokenPositionManager 0x27F971cb582BF9E50F397e4d29a5C7A34f11faA2  
v3Migrator 0xCbf8b7f80800bd4888Fbc7bf1713B80FE4E23E10  
v3Staker 0x62725F55f50bdE240aCa3e740D47298CAc8d57D5  
quoterV2 0xC5290058841028F1614F3A6F0F5816cAd0df5E27  
swapRouter 0x94cC0AaC535CCDB3C01d6787D6413C739ae12bc4  
Testnet interfaces  
INFO  
Two community projects, BaseX and DapDap, provide testnet interfaces for Uniswap contracts if you prefer to interact in the browser instead of with the contracts directly.  
Uniswap v2  
Contract Address  
Factory 0x7Ae58f10f7849cA6F5fB71b7f45CB416c9204b1e  
Router 0x1689E7B1F10000AE47eBfE339a4f69dECd19F602  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/cookie-policy  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Base Cookie Policy  
Last updated: January 17, 2024  
This Cookie Policy explains how Base (referred to here as "we", "us" and "our") uses cookies and similar technologies when you visit base.org (the "Site") and/or when you explore and use Base through the Base protocol or any other applications, tools, and features we operate (collectively referred to as the "Services"). It explains what these technologies are and why we use them, as well as your rights to control our use of them.  
In some cases, we may use cookies and similar technologies to collect personal information, or information that becomes personal information if we combine it with other information. In such cases, the Base Privacy Policy will apply in addition to this Cookie Policy.  
1. What Are Cookies?  
Browser cookies are text files with small pieces of data downloaded onto your computer or mobile device. Browser cookies and other similar technologies enable websites and apps to store information or facilitate access to information stored on your device to enable certain features and distinguish you from other visitors. These technologies are used by most website and app providers to let users navigate between pages efficiently, ensure security of the webpage or application, understand how their websites are used, remember user preferences and generally improve the user experience. More information on cookies and their use can be found at aboutcookies.org or allaboutcookies.org. Cookies set by the website operator are called "first party cookies" and Cookies set by parties other than the website operator are called "third party cookies". You should check the third-party's website for more information on how they use cookies.  
2. What Do We Use Cookies For?  
When you access our Site and Services, we, or companies we work with, may place cookies (collectively called "cookies" in this Cookie Policy) and similar technologies (such as web beacons, software development kits ("SDKs"), APIs, tags and local storage) on your computer or other device for the following purposes:  
Strictly Necessary purposes  
Strictly Necessary cookies are essential for our Services to function and therefore cannot be switched off. They are usually only set in response to actions made by you which amount to a request for services, such as setting your privacy preferences, logging in, or filling in forms. These also include cookies we may rely on for security purposes, such as to prevent unauthorised access attempts. You can set your browser to block or alert you about these cookies at any time, but some features of our Services may not work.  
Performance purposes  
We use these cookies to count visits and traffic sources so we can measure and improve the performance of our Services. These cookies help us to know which pages are the most and least popular and see how visitors move around the Site, and to resolve any errors that occur on the Services quickly to provide you with a better experience. For example, our first party base\_device\_id cookie is used to provide analytics on how our Site and Services perform. This cookie lasts for 3 months.  
3. How to Manage Cookies, Similar Technologies and Targeted Online Mobile Advertising  
You have the right to decide whether to accept or reject cookies (except strictly necessary cookies). You can enable or disable categories of cookies by visiting our Cookie Preference Manager. This includes both first party and third party cookies. You can use the browser with which you are viewing this website to enable, disable or delete cookies. To do this, follow the instructions provided by your browser (usually located within the "Help", "Tools" or "Edit" settings). However, please note, if you set your browser to disable cookies, you may not be able to access secure areas of our Services. Also, if you disable cookies, other parts of our Services may not function properly.  
There are also additional tools available to manage third party cookies. You can visit the DAA's opt-out portal available at optout.aboutads.info, the DAA of Canada's opt-out portal available at youradchoices.ca/en/tools, or visit the NAI's opt-out portal available at optout.networkadvertising.org. Residents of the European Union may opt-out of online behavioural advertising served by the European Interactive Digital Advertising Alliance's participating member organizations by visiting youronlinechoices.eu or through your mobile device settings, where available and the DAA's AppChoices mobile application opt-out offering here: youradchoices.com/appchoices.  
If you have any questions about our use of cookies or other technologies, please submit your request to privacy@base.org.  
4. Will This Cookie Policy Be Updated?  
We may update this Cookie Policy from time to time to reflect, for example, changes to the cookies we use or for other operational, legal or regulatory reasons. You can also revisit this page if you wish to keep yourself informed.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/differences  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Differences between Ethereum and Base  
Base is built on the Bedrock release of the OP Stack, which is designed from the ground up to be as close to Ethereum as possible. Because of this, there are very few differences when it comes to building on Base and Ethereum.  
However, there are still some minor discrepancies between the behavior of Base and Ethereum that you should be aware of when building apps on top of Base.  
These minor differences include:  
Opcodes  
Blocks  
Network specifications  
Transaction costs  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/fees  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Fees  
How do network fees on Base work?  
Every Base transaction consists of two costs: an L2 (execution) fee and an L1 (security) fee. The L2 fee is the cost to execute your transaction on the L2, and the L1 fee is the estimated cost to publish the transaction on the L1. Typically the L1 security fee is higher than the L2 execution fee.  
The L1 fee will vary depending on the amount of transactions on the L1. If the timing of your transaction is flexible, you can save costs by submitting transactions during periods of lower gas on the L1 (for example, over the weekend)  
Similarly, the L2 fee can increase and decrease depending on how many transactions are being submitted to the L2. This adjustment mechanism has the same implementation as the L1; you can read more about it here.  
For additional details about fee calculation on Base, please refer to the op-stack developer documentation.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/network-information  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Network Information  
Base Mainnet  
Name Value  
Network Name Base Mainnet  
Description The public mainnet for Base.  
RPC Endpoint https://mainnet.base.org  
Rate limited and not for production systems.  
Chain ID 8453  
Currency Symbol ETH  
Block Explorer https://base.blockscout.com/  
Base Testnet (Sepolia)  
Name Value  
Network Name Base Sepolia  
Description A public testnet for Base.  
RPC Endpoint https://sepolia.base.org  
Rate limited and not for production systems.  
Chain ID 84532  
Currency Symbol ETH  
Block Explorer https://sepolia-explorer.base.org  
INFO  
L1 & L2 protocol and network-related smart contract deployments can be found on the Base Contracts page.  
INFO  
For production systems, we recommend using a node from one of our node partners, or running your own Base node.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/preparing-for-fault-proofs-on-base-sepolia  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Preparing for fault proofs on Base Sepolia (Testnet)  
Fault proofs are a critical implementation in an L2’s path towards decentralization. They enable a more decentralized approach to validating L2 state and pave the way towards more community participation.  
They improve decentralization with two important capabilities:  
Permissionless output proposals: In an L2 without fault proofs, only the centralized proposer can create and submit output roots about the state of the L2. Now with fault proofs, anyone can make claims about Base's current state instead of relying on a central party.  
Permissionless challenges to output proposals: If someone makes a faulty or fraudulent claim, anyone can challenge it.  
These changes allow anyone to withdraw funds from Base to L1 without having to rely on centralized actors.  
Preparing for fault proofs  
Fault proofs are expected to go live for Base Sepolia (Testnet) in mid-July.  
What’s changing for testnet withdrawals:  
Withdrawals will involve proving and finalizing based on the fault proof system.  
Withdrawals will no longer be instantaneous: they will take at least seven days to finalize, but can take longer depending on the outcome of the corresponding dispute game used.  
In addition, the 'DisputeGameFactory' will replace the 'L2OutputOracle' as the new contract for proposing output root statements. This change is part of the broader shift towards the fault proofing system, which is expected to enhance the security and reliability of the platform.  
If you are in the process of withdrawing your testnet funds from L2 to L1:  
Withdrawals before the upgrade in mid-July will be processed instantaneously.  
Withdrawals during or after the fault proofs upgrade for Base Sepolia will take at least seven days to complete.  
If your withdrawal of testnet funds from Base Sepolia to Ethereum Sepolia coincides with the upgrade proceeding in mid-July, you will be required to resubmit your withdrawal.  
If your team is operating a bridge on Base Sepolia:  
Please provide your users with a notice on your UI to inform them that fault proofs will be enabled for Base Sepolia in mid-July.  
Please ensure it’s clear to your users that testnet withdrawals will now take at least seven days.  
Assess and update your bridging logic, and make sure the new L1 contracts are being used.  
Fault proof contract upgrades will be completed atomically, meaning all affected L1 contracts will be upgraded in a single transaction. No action will be required from node operators.  
Please note, bridge.base.org now redirects to Superbridge and other bridges (collectively, "Superchain bridges"). Superchain bridges are available to initiate and complete deposits and withdrawals to and from Base. Please refer to our docs for more information on bridging.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/privacy-policy  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Base Global Privacy Policy  
Last updated: July 12, 2023  
At Base (referred to here as “we”, “us” or “our”), we respect and protect the privacy of those users and developers (“you” and “your” or “Users” and “Developers”, as relevant) who explore and use Base (“Base”) through the Base protocol or any other applications, tools, and features we operate (collectively, the “Services”).  
This Privacy Policy describes how we collect, use, and disclose personal information when you use our Services, which include the services offered on our website https://base.org ( “Site”). This Privacy Policy does not apply to any processing which Base carries out as a processor on behalf of those Users and Developers who explore and use Base. Please note that we do not control websites, applications, or services operated by third parties, and we are not responsible for their actions. We encourage you to review the privacy policies of the other websites, decentralised applications, and services you use to access or interact with our Services.  
1. WHAT INFORMATION WE COLLECT   
We collect the following personal information when providing the Services:  
Information you provide  
Your public wallet address (“Wallet Address”)  
Publicly available blockchain data (“Blockchain Data”)  
Where you agree to engage in our surveys or sign up to receive marketing communications about Base products and offerings, we will ask for the following “Basic User Information”  
Name  
Email  
Social media handles  
Business name  
Information Collected Automatically  
App, Browser and Device Information:  
Information about the device, operating system, and browser you’re using~~ ~~  
Other device characteristics or identifiers (e.g. plugins, the network you connect to)  
IP address/derived location information  
Information we obtain from Affiliates and third parties  
Information from Coinbase Companies (“Affiliates”): We may obtain information about you, such as Basic User Information from our Affiliates (for instance, if you use Base with your Coinbase-hosted wallet) as part of facilitating, supporting, or providing our Services.  
Blockchain Data: We may analyze public blockchain data, including timestamps of transactions or events, transaction IDs, digital signatures, transaction amounts and wallet addresses  
Information from Analytics Providers: We receive information about your website usage and interactions from third party analytics providers. This includes browser fingerprint, device information,and IP address.  
Error Tracking Data: We utilize information from third party service providers to provide automated error monitoring, reporting, alerting and diagnostic capture for Service and Site errors to allow User or Developers to build more effectively on the Base platform.  
2. HOW WE USE YOUR INFORMATION   
We may use your personal information for the following purposes or as otherwise described at the time of collection. If you reside outside the United Kingdom or European Economic Area (“EEA”), the legal bases on which we rely in your country may differ from those listed below.  
Purpose Information Used Our Legal Basis  
To provide you with the Base Services We use certain information that is necessary to conclude and perform our Terms of Service or other relevant contract(s) with you. Wallet AddressBlockchain Data Contractual Necessity  
To promote the safety, security and integrity of our Services Basic User InformationInformation from Analytics Providers Contractual Necessity  
To allow Users or Developers to build more effectively on the Base platform Error Tracking Data Legitimate Interests  
To send you Base Forms for marketing and product development Basic User Information Legitimate Interests   
3. HOW AND WHY WE SHARE YOUR INFORMATION   
We share certain information about you with service providers, partners and other third parties in order to help us provide our Services. Here’s how:  
Affiliates. Basic User Information that we process and collect may be transferred between Affiliates, Services, and employees affiliated with us as a normal part of conducting business and offering our Services to you.  
Linked Third Party Websites or Services. When you use third-party services (like when you connect your self-custodial wallet to decentralized applications on the Base network) or websites that are linked through our Services, the providers of those services or products may receive information about you (like your wallet address) from Base, you, or others. Please note that when you use third-party services or connect to third-party websites which are not governed by this Privacy Policy, their own terms and privacy policies will govern your use of those services and products.  
Professional advisors, industry partners, authorities and regulators. We share your information described in Section 1. What Information We Collect with our advisors, regulators, tax authorities, law enforcement, government agencies, and industry partners when needed to:  
respond pursuant to applicable law or regulations, court orders, legal process or government requests;  
detect, investigate, prevent, or address fraud and other illegal activity or security and technical issues; and  
protect the rights, property, and safety of our Users, Developers, Affiliates, or others, including to prevent death or imminent bodily harm.  
Vendors and Third-Party Service Providers. When we share information with third-party service providers to help us provide our Services, we require them to use your information on our behalf in accordance with our instructions and terms and only process as necessary for the purpose of the contract.  
4. HOW LONG WE RETAIN YOUR PERSONAL INFORMATION  
We retain your information as needed to provide our Services, comply with legal obligations or protect our or others’ interests. While retention requirements vary by country, we maintain internal retention policies on the basis of how information needs to be used. This includes considerations such as when the information was collected or created, whether it is necessary in order to continue offering you our Services or to protect the safety, security and integrity of our Services, and whether we are required to hold the information to comply with our legal obligations.  
5. CHILDREN’S PERSONAL INFORMATION  
The Services are not directed to persons under the age of 18, and we do not knowingly request or collect any information about persons under the age of 18. If you are under the age of 18, please do not provide any personal information through the Site or Services. If a User submitting personal information is suspected of being younger than 18 years of age, we will take steps to delete the individual’s information as soon as possible.  
6. INTERNATIONAL TRANSFERS  
To facilitate our global operations, we and our third-party partners and service providers may transfer and store throughout the world, including in the United States.  
If you reside in the EEA, Switzerland, or the United Kingdom, we rely upon a variety of legal mechanisms to facilitate these transfers of your personal information (collectively, “European Personal Data”). \*\*\*\*  
We rely primarily on the European Commission’s Standard Contractual Clauses to facilitate the international and onward transfer of European Personal Data to third countries, including from our EU operating entities to Coinbase, Inc. in the United States, who provides the primary infrastructure for the Services.  
We also rely on adequacy decisions from the European Commission where available and exemptions provided for under data protection law (e.g. Article 49 GDPR).  
7. HOW TO CONTACT US WITH QUESTIONS  
If you have questions or concerns regarding this Privacy Policy, or if you have a complaint, please contact us at privacy@base.org.  
8. CHANGES TO THIS PRIVACY POLICY  
We’re constantly trying to improve our Services, so we may need to change this Privacy Policy from time to time as well. We post any changes we make to our Privacy Policy on this page and, where appropriate, we will provide you with reasonable notice of any material changes before they take effect or as otherwise required by law. The date the Privacy Policy was last updated is identified at the top of this page.  
9. OUR RELATIONSHIP WITH YOU  
Coinbase Technologies, Inc., 251 Little Falls Drive, City of Wilmington, County of New Castle, Delaware 19808, acts as controller of your personal data.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/quickstart  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Quick Start: Deploy on Base  
Welcome to the Base deployment quickstart guide! This comprehensive walkthrough will help you set up your environment and deploy smart contracts on Base. Whether you're a seasoned developer or just starting out, this guide has got you covered.  
What You'll Achieve  
By the end of this quickstart, you'll be able to:  
Set up your development environment to deploy on Base  
Deploy your smart contracts to Base  
Connect your frontend to your smart contracts  
WHY BASE?  
Base is a fast, low-cost, builder-friendly Ethereum L2 built to bring the next billion users onchain. By following this guide, you'll join a vibrant ecosystem of developers, creators, and innovators who are building a global onchain economy.  
Set Up Your Development Environment  
Create a new project directory  
mkdir my-base-project && cd my-base-project  
Install Foundry, a powerful framework for smart contract development  
curl -L https://foundry.paradigm.xyz | bash  
foundryup  
This installs Foundry and updates it to the latest version.  
Initialize a new Solidity project  
forge init  
Your Foundry project is now ready. You'll find an example contract in the src directory, which you can replace with your own contracts. For the purposes of this guide, we'll use the Counter contract provided in /src/Counter.sol  
TIP  
Foundry provides a suite of tools for Ethereum application development, including Forge (for testing), Cast (for interacting with the chain), and Anvil (for setting up a local node). You can learn more about Foundry here.  
Configure Foundry with Base  
To deploy your smart contracts to Base, you need two key components:  
A node connection to interact with the Base network  
A funded private key to deploy the contract  
Lets set up both of these:  
1. Set up your node connection  
Create a .env file in your project's root directory  
Add the Base network RPC URL to your .env file  
BASE\_RPC\_URL="https://mainnet.base.org"  
BASE\_SEPOLIA\_RPC\_URL="https://sepolia.base.org"  
Load your environment variables  
source .env  
TIP  
Base Sepolia is the test network for Base, which we will use for the rest of this guide. You can obtain free Base Sepolia ETH from one of the faucets listed here.  
2. Secure your private key  
Store your private key in Foundry's secure keystore  
cast wallet import deployer --interactive  
When prompted enter your private key and a password.  
Your private key is stored in ~/.foundry/keystores which is not tracked by git.  
DANGER  
Never share or commit your private key. Always keep it secure and handle with care.  
Deploy Your Contracts  
Now that your environment is set up, let's deploy your contracts to Base Sepolia.  
Use the following command to compile and deploy your contract  
forge create ./src/Counter.sol:Counter --rpc-url $BASE\_SEPOLIA\_RPC\_URL --account deployer  
Note the format of the contract being deployed is <contract-path>:<contract-name>.  
After successful deployment, the transaction hash will be printed to the console output  
Copy the deployed contract address and add it to your .env file  
COUNTER\_CONTRACT\_ADDRESS="0x..."  
Load the new environment variable  
source .env  
Verify Your Deployment  
To ensure your contract was deployed successfully:  
Check the transaction on Sepolia Basescan.  
Use the cast command to interact with your deployed contract from the command line  
cast call $COUNTER\_CONTRACT\_ADDRESS "number()(uint256)" --rpc-url $BASE\_SEPOLIA\_RPC\_URL  
This will return the initial value of the Counter contract's number storage variable, which will be 0.  
Congratulations! You've deployed your smart contracts to Base Sepolia!  
Next Steps  
Use Onchainkit to connect your frontend to your contracts! Onchainkit is a library of ready-to-use React components and Typescript utilities.  
Learn more about interacting with your contracts in the command line using Foundry from our Foundry tutorial.  
  
  
  
  
ON THIS PAGE  
Set Up Your Development Environment  
Configure Foundry with Base  
Deploy Your Contracts  
Verify Your Deployment  
2. Secure your private key  
Next Steps  
Verify Your Deployment  
Next Steps  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/security  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Security  
Bug bounty program  
In line with our strategy of being the safest way for users to access crypto:  
Coinbase will be extending our best-in-industry million-dollar HackerOne bug bounty program to cover the Base network, the Base bridge contracts, and Base infrastructure.  
Coinbase will be working in tandem with OP Labs to harden the security guarantees of Bedrock and accelerate the timeline for decentralized fault-proofs on the OP Stack.  
Coinbase's bug bounty program will run alongside Optimism's existing Immunefi Bedrock bounty program to support the open source Bedrock OP Stack framework.  
Reporting vulnerabilities  
All potential vulnerability reports can be submitted via the HackerOne platform.  
The HackerOne platform allows us to have a centralized and single reporting source for us to deliver optimized SLA's and results. All reports submitted to the platform are triaged around the clock by our team of Coinbase engineers with domain knowledge, assuring the best quality of review.  
For more information on reporting vulnerabilities and our HackerOne bug bounty program, view our security program policies.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/terms-of-service  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Sequencer, Testnet, Basenames Interface Terms  
Last Updated: August 5, 2024  
We’re excited you’re interested in Base, a layer-two optimistic rollup on the Ethereum public blockchain. While we do not control Base, these Terms of Service (“Terms”) constitute a legally binding contract made between you and Coinbase Technologies, Inc. (“Coinbase,” “we,” or “us”) that governs your access to and use of the Coinbase Sequencer, Base Testnet, and Basenames Interface, each of which is defined below (collectively, the “Services”). By using the Services in any way, you agree to be bound by these Terms. If you do not accept the terms and conditions of these Terms, you are not permitted to access or otherwise use the Services.  
BEFORE WE INCLUDE ANY OTHER DETAILS, WE WANT TO GIVE YOU NOTICE OF SOMETHING UP FRONT: BY AGREEING TO THESE TERMS, YOU AND WE AGREE TO RESOLVE ANY DISPUTES WE MAY HAVE WITH EACH OTHER VIA BINDING ARBITRATION OR IN SMALL CLAIMS COURT (INSTEAD OF A COURT OF GENERAL JURISDICTION), AND YOU AGREE TO DO SO AS AN INDIVIDUAL (INSTEAD OF, FOR EXAMPLE, AS A REPRESENTATIVE OR MEMBER OF A CLASS IN A CLASS ACTION). TO THE EXTENT THAT THE LAW ALLOWS, YOU ALSO WAIVE YOUR RIGHT TO A TRIAL BY JURY. FOR MORE INFORMATION, SEE OUR ARBITRATION AGREEMENT “DISPUTE RESOLUTION, ARBITRATION AGREEMENT, CLASS ACTION WAIVER, AND JURY TRIAL WAIVER.”  
1. Base and Bridging Smart Contracts  
The Base protocol (“Base”) is an open source, optimistic rollup protocol that operates with the Ethereum blockchain. The Base protocol includes protocol smart contracts that allow you to “bridge” (i.e., lock assets on one blockchain protocol and replicate them on another protocol) digital assets between Ethereum and/or Base (“Bridging Smart Contracts”). Neither Base nor the Bridging Smart Contracts are part of the Services. They are both operated through the use of certain open source software such as the OP Stack, an open sourced codebase approved by a decentralized, representative body of Optimism governance (the “Optimism Collective”), and a set of smart contracts that once deployed to the Base protocol are not controlled by Coinbase (even if Coinbase contributed to their initial development). Coinbase does not control what third parties may build on Base, the activity of such parties, any user transacting on Base, or any data stored on Base itself, and Coinbase does not take possession, custody, or control over any virtual currency or other digital asset on Base or the Bridging Smart Contracts, unless expressly stated in a written contract signed by Coinbase. You acknowledge and agree that Coinbase makes no representations or warranties with respect to Base or the Bridging Smart Contracts, and that, if you use Base or the Bridging Smart Contracts, you do so at your own risk.  
2. Basenames  
Basenames is an open source blockchain-based naming protocol that maintains a registry of all domains and subdomains on Base through a series of smart contracts deployed on Base. Basenames is not part of the Services. Users may, through interacting with the Basenames, search such registry, register domains and subdomains and manage their registered names. The Basenames interface located at https://base.org/names (the “Basenames Interface”) is one, but not the exclusive, means of accessing Basenames. You are responsible for conducting your own diligence on other interfaces enabling you to access Basenames to understand the fees and risks that they present. You understand that anyone can register and own a domain name (and its subdomains) that is not already registered on the registry maintained by Basenames. You further understand that names registered on the registry maintained by Basenames may expire and you are responsible for monitoring and renewing the registration of such names. You acknowledge that Coinbase is not able to forcibly remove, prevent or otherwise interfere with the ability of any person to register a domain name on the registry operated by Basenames and you hereby acknowledge that Coinbase will not be liable for any claims or damages whatsoever associated with your use, inability to use any domain names subject of registration, or to be registered, on the registry maintained by Basenames. You agree that Basenames is purely non-custodial, meaning you are solely responsible for the custody of the cryptographic private keys to the digital asset wallets you hold and use to access Basenames.  
3. Who May Use the Services  
You may only use the Services if you are legally capable of forming a binding contract with Coinbase in your respective jurisdiction which may require your parents consent if you’re not the legal age of majority (which in many jurisdictions is 18), and not barred from using the Services under the laws of any applicable jurisdiction, for example, that you do not appear on the U.S. Treasury Department’s list of Specially Designated Nationals and are not located or organized in a U.S. sanctioned jurisdiction. If you are using the Services on behalf of an entity or other organization, you agree to these Terms for that entity or organization and represent to Coinbase that you have the authority to bind that entity or organization to these Terms.  
4. Rights We Grant You  
As between you and us, Coinbase is the owner of the Services, including all related intellectual property rights and proprietary content, information, material, software, images, text, graphics, illustrations, logos, trademarks (including the Base logo, the Base name, the Coinbase logo, the Coinbase name, and any other Coinbase or Base marks), service marks, copyrights, photographs, audio, video, music, and the “look and feel” of the Services. We hereby permit you to use and access the Services, provided that you comply with these Terms. If any software, content or other materials owned or controlled by us are distributed to you as part of your use of the Services, we hereby grant you a non-sublicensable, non-transferable, and non-exclusive right and license to execute, access and display such software, content and materials provided to you as part of the Services, in each case for the sole purpose of enabling you to use the Services as permitted by these Terms. To use any parts of the contents of the Services other than for personal and non-commercial use, you must seek permission from Coinbase in writing. Coinbase reserves the right to refuse permission without providing any reasons.  
5. Accessing the Services  
To access the Services, Base, or the Bridging Smart Contracts you must connect a compatible cryptocurrency wallet software (“Wallet”). Your relationship with any given Wallet provider is governed by the applicable terms of that Wallet provider, not these Terms. You are responsible for maintaining the confidentiality of any private key controlled by your Wallet and are fully responsible for any and all messages or conduct signed with your private key. We accept no responsibility or liability to you in connection with your use of a Wallet, and make no representations and warranties regarding how the Services, Base, or the Bridging Smart Contracts will operate or be compatible with any specific Wallet. We reserve the right, in our sole discretion, to prohibit certain Wallet addresses from being able to use or engage in transactions via the Coinbase Sequencer or from using other aspects of the Services.  
As between you and Coinbase, you retain ownership and all intellectual property rights to the content and materials you submit to the Services. But, you grant us a limited, non-exclusive, worldwide, royalty free license to use your content solely for the purpose of operating the Services (i.e., the Sequencer and Base Testnet) for so long as we operate the Services. To avoid any doubt, this license does not allow us to use your intellectual property beyond operating the Services (e.g., in advertisements).  
6. The Services  
Coinbase offers the following Services that enable you to access and interact with Base and the Bridging Smart Contracts:  
The Sequencer: The Coinbase Sequencer is a node operated by Coinbase that receives, records, and reports transactions on Base. While The Coinbase Sequencer is, initially, the only sequencer node supporting transactions on Base, additional nodes may be provided by third parties in the future and there are other mechanisms for submitting transactions through Ethereum. The Coinbase Sequencer does not store, take custody of, control, send, or receive your virtual currency, except for receiving applicable gas fees. It also does not have the ability to modify, reverse, or otherwise alter any submitted transactions, and will not have access to your private key or the ability to control value on your behalf. We reserve the right to charge and modify the fees in connection with your use of the Coinbase Sequencer. These fees may also be subject to taxes under applicable law.  
Base Testnet: The Base Testnet is a test environment that allows you to build applications integrated with Base. You are permitted to access and use the Base Testnet only to test and improve the experience, security, and design of Base or applications built on Base, subject to these Terms. Base Testnet Tokens will not be converted into any future rewards offered by Coinbase. Coinbase may change, discontinue, or terminate, temporarily or permanently, all or any part of the Base Testnet, at any time and without notice.  
Basenames Interface: The Basenames Interface is a web application and graphical user display operated by Coinbase and located at base.org/names. It enables you to interact with Basenames by creating blockchain messages that you can sign and broadcast to Base using your Wallet. The Basenames Interface will not have access to your private key at any point.  
7. Acceptable Use  
You agree that you will not use the Services in any manner or for any purpose other than as expressly permitted by these Terms. That means, among other things, you will not use the Services to do or encourage any of the following:  
Infringe or violate the intellectual property rights or any other rights of anyone else (including Coinbase) or attempt to decompile, disassemble, or reverse engineer the Services;  
Violate any applicable law or regulation, including without limitation, any applicable anti-money laundering laws, anti-terrorism laws, export control laws, end user restrictions, privacy laws or economic sanctions laws/regulations, including those administered by the U.S. Department of Treasury’s Office of Foreign Assets Control;  
Use the Services in a way that is dangerous, harmful, fraudulent, misleading, deceptive, threatening, harassing, defamatory, obscene, or otherwise objectionable;  
Violate, compromise, or interfere with the security, integrity, or availability of any computer, network, or technology associated with the Services, including using the Services in a manner that constitutes excessive or abusive usage, attempts to disrupt, attack, or interfere with other users, or otherwise impacts the stability of the Services.  
Use any Coinbase brands, logos, or trademarks (or any brands, logos, or trademarks that are confusingly similar) without our express prior written approval, which we may withhold at our discretion for any reason.  
8. Release and Assumption of Risk  
‍By using the Services, Base, or the Bridging Smart Contracts, you represent that you understand there are risks inherent in using cryptographic and public blockchain-based systems, including, but not limited, to the Services and digital assets such as bitcoin (BTC) and ether (ETH). You expressly agree that you assume all risks in connection with your access and use of Base, the Bridging Smart Contracts, Basenames, and the separate Services offered by Coinbase. That means, among other things, you understand and acknowledge that:  
The Base, the Bridging Smart Contracts, Basenames, and the separate Services may be subject to cyberattacks and exploits, which could result in the irrevocable loss or reduction in value of your digital assets or in additional copies of your digital assets being created or bridged without your consent.  
Base is subject to periodic upgrades by the Optimism Collective. The Optimism Collective may approve a protocol upgrade that, if implemented, may significantly impacts Base, and may introduce other risks, bugs, malfunctions, cyberattack vectors, or other changes to Base that could disrupt the operation of Base, the Bridging Smart Contracts, Basenames, or the Services or otherwise cause you damage or loss.  
If you lose your Wallet seed phrase, private keys, or password, you might permanently be unable to access your digital assets. You bear sole responsibility for safeguarding and ensuring the security of your Wallet.  
You further expressly waive and release Coinbase, its parents, affiliates, related companies, their officers, directors, members, employees, consultants, representatives. agents, partners, licensors, and each of their respective successors and assigns (collectively, the “Coinbase Entities”) from any and all liability, claims, causes of action, or damages arising from or in any way related to your use of the Services, and your interaction with Base, the Bridging Smart Contracts, or Basenames. Also, to the extent applicable, you shall and hereby do waive the benefits and protections of California Civil Code § 1542, which provides: “[a] general release does not extend to claims that the creditor or releasing party does not know or suspect to exist in his or her favor at the time of executing the release and that, if known by him or her, would have materially affected his or her settlement with the debtor or released party.”  
9. Interactions with Other Users  
You are responsible for your interactions with other users on or through the Services. While we reserve the right to monitor interactions between users, we are not obligated to do so, and we cannot be held liable for your interactions with other users, or for any user’s actions or inactions. If you have a dispute with one or more users, you release us (and our affiliates and subsidiaries, and our and their respective officers, directors, employees and agents) from claims, demands and damages (actual and consequential) of every kind and nature, known and unknown, arising out of or in any way connected with such disputes. In entering into this release you expressly waive any protections (whether statutory or otherwise) that would otherwise limit the coverage of this release to include only those claims which you may know or suspect to exist in your favor at the time of agreeing to this release.  
10. Feedback  
Any questions, comments, suggestions, ideas, feedback, reviews, or other information about the Services, provided by you to Coinbase, are non-confidential and Coinbase will be entitled to the unrestricted use and dissemination of these submissions for any purpose, commercial or otherwise, without acknowledgment, attribution, or compensation to you.  
11. Privacy  
For more information regarding our collection, use, and disclosure of personal data and certain other data, please see our Privacy Policy. The processing of personal data by Coinbase as a processor will be subject to any data processing agreement that you enter into with Coinbase.  
12. Third-Party Services  
The Services may provide access to services, sites, technology, applications and resources that are provided or otherwise made available by third parties (“Third-Party Services”). Your access and use of Third-Party Services may also be subject to additional terms and conditions, privacy policies, or other agreements with such third parties. Coinbase has no control over and is not responsible for such Third-Party Services, including for the accuracy, availability, reliability, or completeness of information or content shared by or available through Third-Party Services, or on the privacy practices of Third-Party Services. We encourage you to review the privacy policies of Third-Party Services prior to using such services. You, and not Coinbase, will be responsible for any and all costs and charges associated with your use of any Third-Party Services. The integration or inclusion of such Third-Party Services does not imply an endorsement or recommendation. Any dealings you have with third parties while using the Services — including if a Third-Party Service may have infringed your intellectual property rights — are between you and the third party. Coinbase will not be responsible or liable, directly or indirectly, for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any Third-Party Services.  
13. Additional Services  
We or our affiliates may offer additional services that interact with Base, which may require you to agree to additional terms. If, while using an additional service, there is a conflict between these Terms and the additional terms covering that service, the additional terms will prevail.  
14. Indemnification  
To the fullest extent permitted by applicable laws, you will indemnify and hold the Coinbase Entities harmless from and against any claims, disputes, demands, liabilities, damages, losses, and costs and expenses, including, without limitation, reasonable legal and accounting fees arising out of or in any way connected with (a) your access to or use of the Services, (b) your violation of these Terms, or (c) your negligence or willful misconduct. If you are obligated to indemnify any Coinbase Entity hereunder, then you agree that Coinbase (or, at its discretion, the applicable Coinbase Entity) will have the right, in its sole discretion, to control any action or proceeding and to determine whether Coinbase wishes to settle, and if so, on what terms, and you agree to fully cooperate with Coinbase in the defense or settlement of such claim.  
15. Warranty Disclaimers  
TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, BASE, THE BRIDGING SMART CONTRACTS, BASENAMES, AND THE SERVICES ARE PROVIDED ON AN “AS IS” AND “AS AVAILABLE” BASIS WITHOUT ANY REPRESENTATION OR WARRANTY, WHETHER EXPRESS, IMPLIED OR STATUTORY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, COINBASE SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND/OR NON-INFRINGEMENT. THE COINBASE ENTITIES DO NOT MAKE ANY REPRESENTATIONS OR WARRANTIES THAT (I) ACCESS TO THE SERVICES, BASE, THE BRIDGING SMART CONTRACTS, OR BASENAMES WILL BE CONTINUOUS, UNINTERRUPTED, OR TIMELY; (II) THE SERVICES, BASE, THE BRIDGING SMART CONTRACTS, OR BASENAMES WILL BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM OR OTHER SERVICES, INCLUDING ANY WALLETS; (III) THE SERVICES, BASE, THE BRIDGING SMART CONTRACTS, OR BASENAMES WILL BE SECURE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE; (IV) THE SERVICES, BASE, THE BRIDGING SMART CONTRACTS, OR BASENAMES WILL PREVENT ANY UNAUTHORIZED ACCESS TO, ALTERATION OF, OR THE DELETION, DESTRUCTION, DAMAGE, LOSS OR FAILURE TO STORE ANY OF YOUR CONTENT OR OTHER DATA; OR (V) THAT THE SERVICES, BASE, THE BRIDGING SMART CONTRACTS, OR BASENAMES WILL PROTECT YOUR ASSETS FROM THEFT, HACKING, CYBER ATTACK, OR OTHER FORM OF LOSS OR DEVALUATION CAUSED BY THIRD-PARTY CONDUCT.  
16. Limitation of Liability  
TO THE MAXIMUM EXTENT PERMITTED BY LAW, NEITHER THE COINBASE ENTITIES NOR ITS SERVICE PROVIDERS INVOLVED IN CREATING, PRODUCING, OR DELIVERING THE SERVICES WILL BE LIABLE FOR ANY INCIDENTAL, SPECIAL, EXEMPLARY OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOST PROFITS, LOST REVENUES, LOST SAVINGS, LOST BUSINESS OPPORTUNITY, LOSS OF DATA OR GOODWILL, SERVICE INTERRUPTION, COMPUTER DAMAGE OR SYSTEM FAILURE, INTELLECTUAL PROPERTY INFRINGEMENT, OR THE COST OF SUBSTITUTE SERVICES OF ANY KIND ARISING OUT OF OR IN CONNECTION WITH THESE TERMS OR FROM THE USE OF OR INABILITY TO USE THE SERVICES, BASE, THE BRIDGING SMART CONTRACTS, OR BASENAMES, WHETHER BASED ON WARRANTY, CONTRACT, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR ANY OTHER LEGAL THEORY, AND WHETHER OR NOT THE COINBASE ENTITIES OR ITS SERVICE PROVIDERS HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGE, EVEN IF A LIMITED REMEDY SET FORTH HEREIN IS FOUND TO HAVE FAILED OF ITS ESSENTIAL PURPOSE.  
TO THE MAXIMUM EXTENT PERMITTED BY LAW, IN NO EVENT WILL THE COINBASE ENTITIES’ TOTAL LIABILITY ARISING OUT OF OR IN CONNECTION WITH THESE TERMS OR FROM THE USE OF OR INABILITY TO USE THE SERVICES, BASE, THE BRIDGING SMART CONTRACTS, OR BASENAMES EXCEED THE AMOUNTS YOU HAVE PAID OR ARE PAYABLE BY YOU TO THE COINBASE ENTITIES FOR USE OF THE SERVICES OR ONE HUNDRED DOLLARS ($100), WHICHEVER IS HIGHER.  
THE EXCLUSIONS AND LIMITATIONS OF DAMAGES SET FORTH ABOVE ARE FUNDAMENTAL ELEMENTS OF THE BASIS OF THE BARGAIN BETWEEN COINBASE AND YOU.  
IF ANY PORTION OF THESE SECTIONS IS HELD TO BE INVALID UNDER THE LAWS OF YOUR STATE OF RESIDENCE, THE INVALIDITY OF SUCH PORTION WILL NOT AFFECT THE VALIDITY OF THE REMAINING PORTIONS OF THE APPLICABLE SECTIONS. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL OR CERTAIN OTHER DAMAGES, SO THE ABOVE LIMITATIONS AND EXCLUSIONS MAY NOT APPLY TO YOU.  
17. Changes to Terms  
We reserve the right, in our sole discretion, to change these Terms at any time and your continued use of the Services after the date any such changes become effective constitutes your acceptance of the new Terms. You should periodically visit this page to review the current Terms so you are aware of any revisions. If you do not agree to abide by these or any future Terms, you are not permitted to access, browse, or use (or continue to access, browse, or use) the Services.  
18. Notice  
Any notices or other communications provided by us under these Terms, including those regarding modifications to these Terms, will be posted online, in the Services, or through other electronic communication. You agree and consent to receive electronically all communications, agreements, documents, notices and disclosures that we provide in connection with your use of the Services.  
19. Entire Agreement.  
These Terms and any other documents incorporated by reference comprise the entire understanding and agreement between you and Coinbase as to the subject matter hereof, and supersedes any and all prior discussions, agreements and understandings of any kind (including without limitation any prior versions of these Terms), between you and Coinbase. Section headings in these Terms are for convenience only and shall not govern the meaning or interpretation of any provision of these Terms.  
20. Assignment  
We reserve the right to assign our rights without restriction, including without limitation to any Coinbase affiliates or subsidiaries, or to any successor in interest of any business associated with the Services. In the event that Coinbase is acquired by or merged with a third party entity, we reserve the right, in any of these circumstances, to transfer or assign the information we have collected from you as part of such merger, acquisition, sale, or other change of control. You may not assign any rights and/or licenses granted under these Terms. Any attempted transfer or assignment by you in violation hereof shall be null and void. Subject to the foregoing, these Terms will bind and inure to the benefit of the parties, their successors and permitted assigns.  
21. Severability  
If any provision of these Terms is determined to be invalid or unenforceable under any rule, law, or regulation of any local, state, or federal government agency, such provision will be changed and interpreted to accomplish the objectives of the provision to the greatest extent possible under any applicable law and the validity or enforceability of any other provision of these Terms shall not be affected.  
22. Termination; Survival  
We may suspend or terminate your access to and use of the Services at our sole discretion, at any time and without notice to you. Upon any termination, discontinuation or cancellation of the Services, Sections 7 through 27 of the Terms will survive.  
23. Governing Law  
You agree that the laws of the State of California, without regard to principles of conflict of laws, will govern these Terms and any Dispute, except to the extent governed by federal law.  
24. Force Majeure  
We shall not be liable for delays, failure in performance or interruption of service which result directly or indirectly from any cause or condition beyond our reasonable control, including but not limited to, significant market volatility, act of God, act of civil or military authorities, act of terrorists, civil disturbance, war, strike or other labor dispute, fire, interruption in telecommunications or Internet services or network provider services, failure of equipment and/or software, pandemic, other catastrophe or any other occurrence which is beyond our reasonable control and shall not affect the validity and enforceability of any remaining provisions.  
25. Non-Waiver of Rights  
These Terms shall not be construed to waive rights that cannot be waived under applicable laws, including applicable state money transmission laws in the state where you are located. In addition, our failure to insist upon or enforce strict performance by you of any provision of these Terms or to exercise any right under these Terms will not be construed as a waiver or relinquishment to any extent of our right to assert or rely upon any such provision or right in that or any other instance.  
26. Relationship of the Parties  
Coinbase is an independent contractor for all purposes. Nothing in these Terms is intended to or shall operate to create a partnership or joint venture between you and Coinbase, or authorize you to act as agent of Coinbase. These Terms are not intended to, and do not, create or impose any fiduciary duties on us. To the fullest extent permitted by law, you acknowledge and agree that we owe no fiduciary duties or liabilities to you or any other party, and that to the extent any such duties or liabilities may exist at law or in equity, those duties and liabilities are hereby irrevocably disclaimed, waived, and foregone. You further agree that the only duties and obligations that we owe you are those set out expressly in these Terms.  
26. Dispute Resolution, Arbitration Agreement, Class Action Waiver, And Jury Trial Waiver  
If you have a dispute with us, you agree to first contact Coinbase Support via our Customer Support page (https://help.coinbase.com). If Coinbase Support is unable to resolve your dispute, you agree to follow our Formal Complaint Process. You begin this process by submitting our complaint form. If you would prefer to send a written complaint via mail, please include as much information as possible in describing your complaint, including your support ticket number, how you would like us to resolve the complaint, and any other relevant information to us at 82 Nassau St #61234, New York, NY 10038. The Formal Complaint Process is completed when Coinbase responds to your complaint or 45 business days after the date we receive your complaint, whichever occurs first. You agree to complete the Formal Complaint Process before filing an arbitration demand or action in small claims court.  
Disputes with Users Who Reside in the United States or Canada  
If you reside in the United States or Canada, and if you have a dispute with us or if we have a dispute with you, the dispute shall be resolved through binding arbitration or in small claims court pursuant to the Arbitration Agreement in Appendix 1 below.  
As an illustration only, the following is a summary of some of the terms of the Arbitration Agreement:  
Disputes will be resolved individually (in other words, you are waiving your right to proceed against Coinbase in a class action). However, if you or we bring a coordinated group of arbitration demands with other claimants, you and we agree that the American Arbitration Association (AAA) must batch your or our arbitration demand with up to 100 other claimants to increase the efficiency and resolution of such claims.  
Certain disputes must be decided before a court, including (1) any claim that the class action waiver is unenforceable, (2) any dispute about the payment of arbitration fees, (3) any dispute about whether you have completed the prerequisites to arbitration (such as exhausting the support and Formal Complaint processes), (4) any dispute about which version of the Arbitration Agreement applies, and (5) any dispute about whether a dispute is subject to the Arbitration Agreement in the first instance.  
In the event that a dispute is filed with a court that does not fall into one of the above five categories, either you or Coinbase may move to compel the court to order arbitration. If the court issues an order compelling arbitration, the prevailing party on the motion to compel may recover its reasonable attorneys’ fees and costs.  
Disputes with Users Who Reside Outside the United States and Canada  
If you do not reside in the United States or Canada, the Arbitration Agreement in Appendix 1 does not apply to you and you may resolve any claim you have with us relating to, arising out of, or in any way in connection with our Terms, us, or our Services in a court of competent jurisdiction.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tokens/list  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
The Base Token List  
This page is intended for token issuers who already have an ERC-20 contract deployed on Ethereum and would like to submit their token for bridging between Ethereum and Base. Base uses the Superchain token list as a reference for tokens that have been deployed on Base.  
Disclaimer: Base does not endorse any of the tokens that are listed in the Github repository and has conducted only preliminary checks, which include automated checks listed here.  
Adding your token to the list  
The steps below explain how to get your token on the Base Token List.  
Step 1: Deploy your token on Base  
Select your preferred bridging framework and use it to deploy an ERC-20 for your token on Base. We recommend you use the framework provided by Base's standard bridge contracts, and furthermore deploy your token using the OptimismMintableERC20Factory. Deploying your token on Base in this manner provides us with guarantees that will smooth the approval process. If you choose a different bridging framework, its interface must be compatible with that of the standard bridge, otherwise it may be difficult for us to support.  
Step 2: Submit details for your token  
Follow the instructions in the GitHub repository and submit a PR containing the required details for your token. You must specify in your token's data.json file a section for ‘base-sepolia' and/or ‘base’. The change you need to submit is particularly simple if your token has already been added to the Superchain token list. For example, this PR shows the change required for cbETH, which was already on Optimism's token list and relies on the Base standard bridge.  
Step 3: Await final approval  
Reviews are regularly conducted by the Base team and you should receive a reply within 24-72 hours (depending on if the PR is opened on a week day, weekend or holiday).  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tokens/wallet  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
How to ensure ERC-20 tokens are immediately swappable in Coinbase Wallet  
This page is intended for developers that will or have recently deployed ERC-20 token contracts on Base Mainnet and would like their token details to display as quickly as possible on Coinbase Wallet.  
Coinbase Wallet makes any ERC-20 token instantly available for swapping seconds from when the contract is deployed.  
Follow the instructions below to ensure your token logo, asset name, and other metadata also appear on Coinbase Wallet.  
DISCLAIMER  
Base does not endorse any specific token that is deployed on mainnet and made available for swapping.  
Adding your token to the list  
The steps below explain how to have your token display quickly on Coinbase Wallet. These instructions work not only for Base, but for any EVM chain supported by Coinbase Wallet (Optimism, Arbitrum, Polygon, Avalanche, Fantom, BNB).  
Step 1: Deploy your ERC-20 Token on Base Mainnet  
Write and deploy a compliant ERC-20 token smart contract. Test it and then deploy on Base Mainnet.  
Once your ERC-20 contract is deployed, your asset is swappable instantly on Coinbase Wallet in the swap flow. Users can search by contract address or asset name. See below for information on how to show price charts and other metadata.  
Step 2: Prepare your metadata and asset images  
Prepare a high-resolution image of your token's logo. Ensure it is clear, identifiable, and representative of your token.  
Step 3: List your cryptocurrency on a listing aggregator  
Note: At this time, being listed and verified on CoinMarketCap is the best way to ensure your token’s name, image, price chart all show up on Coinbase Wallet.  
You can pay to be listed AND verified on CoinMarketCap following these instructions.  
You can list for free on CoinGecko following these instructions.  
Once CoinGecko lists your token OR CoinMarketCap lists it as verified, your asset's image logo and other metadata will flow into Coinbase Wallet and can be seen by users. It can take 24-48 hours for metadata changes to update.  
Why does my token display in the “Newer tokens” section?  
Tokens that are newly launched and have not had significant trading volume appear in this section. Once your token reaches a market cap of at least $10M on CoinGecko or CoinMarketCap, the newer token label inside Coinbase Wallet is removed.  
Why is there no price chart for my token?  
Your token must be listed and marked as verified on CoinMarketCap for the price chart to display on Coinbase Wallet.  
If the above guidance doesn’t resolve your issue, please submit more information using this Deform.  
Sharing your token  
Custom trading links  
By sharing a unique link to your token’s asset page, your community can more easily interact with your token.  
How to get your custom link:  
Step 1: Grab your custom link for your token by navigating to the asset page on Coinbase Wallet  
Step 2: Click the share button  
DISCLAIMER  
New assets with low liquidity may result in failed swaps or may result in a user receiving less of the destination token due to slippage. An important responsibility of the token creator is to communicate to the community these risks.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/account-abstraction  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Account Abstraction  
Alchemy Account Kit  
Account Kit is a complete solution for account abstraction. Using Account Kit, you can create a smart contract wallet for every user that leverages account abstraction to simplify every step of your app's onboarding experience. It also offers Gas Manager and Bundler APIs for sponsoring gas and batching transactions.  
Biconomy  
Biconomy is an Account Abstraction toolkit that enables you to provide the simplest UX for your dapp or wallet. It offers modular smart accounts, as well as paymasters and bundlers as a service for sponsoring gas and executing transactions at scale.  
Coinbase Account Abstraction Kit  
The Coinbase Developer Platform Account Abstraction Kit is an account abstraction toolkit for building simple onchain user experiences. Account Abstraction Kit provides a paymaster and bundler that allows you to sponsor gas fees and bundle user transactions, improving the user experience of your application.  
Openfort  
Openfort is an infrastructure provider designed to simplify the development of games and gamified experiences across their suite of API endpoints. The platform vertically integrates the AA stack, so game developers can focus on game development without worrying about private key management, the account model or the onchain interactions with paymasters and bundlers. The Openfort platform is compatible with most EVM chains, including Base.  
Pimlico  
Pimlico provides an infrastructure platform that makes building smart accounts simpler. If you are developing, an ERC-4337 smart account, they provide bundlers, verifying paymasters, ERC-20 paymasters, and much more.  
Safe  
Safe provides modular smart account infrastructure and account abstraction stack via their Safe{Core} Account Abstraction SDK, API, and Protocol.  
Stackup  
Stackup provides smart account tooling for building account abstraction within your apps. They offer Paymaster and Bundler APIs for sponsoring gas and sending account abstraction transactions.  
thirdweb  
thirdweb offers the complete toolkit to leverage account abstraction technology to enable seamless user experiences for your users. This includes Account Factory contracts that let your users spin up Smart Accounts, Bundler for UserOps support, and Paymaster to enable gas sponsorships.  
WalletKit  
WalletKit is an all-in-one platform for adding smart, gasless wallets to your app. It has integrated support for ERC 4337 and comes with a paymaster and bundler included, requiring no extra setup.  
WalletKit also offers pre-built components for onboarding users with email and social logins, which can be integrated in under 15 minutes using their React SDK or the wagmi connector. Alternatively, build completely bespoke experiences for your users using WalletKit's Wallets API.  
WalletKit is compatible with most EVM chains, including Base. You can check out the WalletKit documentation here. Start building for free on the Base testnet today.  
ZeroDev  
ZeroDev is an embedded wallet powered by account abstraction. It offers you the ability to create self-custody wallets for your users, sponsor gas, and simplify user flows by batching and automating transactions.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/basenames-faq  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Basenames FAQ  
FAQ  
1. What are Basenames?  
Basenames are a core onchain building block that enable builders to establish their identity on Base by registering human-readable names for their wallet address(es). They are fully onchain, built on the same technology powering ENS names and deployed on Base. These human-readable names can be used when connecting to onchain apps, and sending and receiving on Base and any other EVM chain. Get your Basename at base.org/names.  
2. What are the Basename registration fees?  
Basenames are priced based on name length, and are designed to be globally accessible. Annual registration fees are as follows:  
Letters Annual fee  
3 0.1 ETH  
4 0.01 ETH  
5-9 0.001 ETH  
10+ 0.0001 ETH  
3. How do I get a free or discounted Basename?  
You can get one free Basename (5+ letters) for one year if you meet any of the below criteria:  
Coinbase Verification  
Coinbase One Verification (free renewals with active subscription)  
Summer Pass Level 3 NFT  
Buildathon participant NFT  
base.eth NFT holder  
cb.id username (acquired prior to Fri Aug 9, 2024)  
BNS name owner - free 4+ letter name (basename.app)  
An equivalent-value discount of 0.001 ETH will be applied if registering a shorter name, or registering for more than 1 year, with the exception of the BNS name owner discount (valued at 0.01 ETH per unique address). You will need to pay the standard registration fees if you wish to keep your Basename after your initial discount has been fully applied. Discounts are only applied once, and are limited to one per address. Even if you meet multiple criteria, you will only be eligible for a single discount on one Basename. If you satisfy multiple criteria, we will automatically apply the highest-value discount to your registration.  
We are always looking to add more discounts. If you or your project have ideas for more discounts, please reach out.  
4. Why is there an auction at launch, and how does it work?  
Upon initial launch, there will be a temporary premium placed on all Basenames in the form of a Dutch auction, to ensure a fair and quality distribution of names, and to maximize everyone's chance of getting a name they like without being outcompeted by bots. The premium will start at 100 ETH and decay exponentially over the course of 36 hours. Premiums will be added on to the total registration cost of a Basename. Please note: the premium is intentionally designed to be high so that names can't be instantly bought by bots or traders, and can instead enable fairer access and price discovery for the general public.  
5. Do I have to pay gas to register a Basename?  
If registering with a Smart Wallet, registrations will be gasless, sponsored by Base.  
6. How long can I register a Basename for?  
There is no limit to registration length, but there is a minimum of 1 year.  
7. How can I use my Basename?  
You can use your Basename across apps in the Base ecosystem, starting with base.org, Onchain Registry, and Onchain Summer Pass. You can also use it for sending and receiving on Base and other EVM chains.  
8. Is my profile information published onchain?  
Basenames are fully onchain, and therefore any information you publish is recorded onchain, requires a transaction, and will be broadly composable with the rest of the ecosystem. Please do not publish any information you do not wish to be onchain.  
9. How do I set my Basename as my primary name for my address?  
You can set your Name as your primary name by updating this in Profile Management. If you set your Basename as your primary name, it will be displayed on any wallet or app that has added support for Basenames.  
10. How do I transfer my Basename to another address?  
You can transfer your Basename to another address through Profile Management:  
Transfer token ownership - transfers ownership of the Basename token and associated permissions.  
Transfer management - transfers ability to manage and update profile records.  
Change address resolution - Basename will resolve to a new address.  
Transferring all 3 to the same address will fully transfer ownership of the Basename to that address.  
11. What happens if I forget to renew my Basename?  
If you forget to renew your Name, it will enter a grace period of 90 days, during which you can still renew it. If not renewed during this period, the Basename will become available for others to register.  
12. What happens if a Basename is not renewed during the grace period?  
If a Basename is not renewed after the 90 day grace period, it will be subject to a temporary premium in the form of a Dutch auction. This premium starts at 100ETH and will decay exponentially over the course of 21 days.  
13. Can I link multiple addresses to my Basename?  
Currently, only one address at a time can be linked to a Basename. However, we plan to support multi-address linking in the future.  
14. I am a builder. How do I integrate Basenames to my app?  
If you're a builder looking to integrate Basenames into your app, OnchainKit is the easiest way to get started (tutorial here). If you have ideas for new features or badges that you'd like to integrate with Basenames, we'd love to hear from you.  
15. How do I get a Basename for my app or project?  
You can register a Basename for your app just like any other Basename. If a Basename for your app or project is not available, there is a good chance it was reserved. Please reach out to our team or fill out this form and we will reach out with instructions.  
16. How are Basenames built?  
Basenames are built using the Ethereum Name Service (ENS) protocol, leveraging its decentralized architecture to ensure secure and efficient name resolution.  
17. Do Basenames work on different chains?  
Yes, your Name will work on any chain as long as the app is ENSIP-10 compliant. Note that when sending money or interacting across different chains, you should ensure the receiving platform supports ENS.  
18. How do I use frames with my basename?  
You can pin frames as the ultimate way to make a profile yours. Want someone to mint? Frame it. Want someone to pay you? Frame it. Want to display your onchain identity your own way? Frame it.  
Follow our step-by-step guide to get started. While Basenames supports all frames, we recommend using the Open Frames standard for the best experience.  
Tags:basenames  
ON THIS PAGE  
FAQ  
1. What are Basenames?  
2. What are the Basename registration fees?  
3. How do I get a free or discounted Basename?  
4. Why is there an auction at launch, and how does it work?  
5. Do I have to pay gas to register a Basename?  
6. How long can I register a Basename for?  
7. How can I use my Basename?  
8. Is my profile information published onchain?  
9. How do I set my Basename as my primary name for my address?  
10. How do I transfer my Basename to another address?  
11. What happens if I forget to renew my Basename?  
12. What happens if a Basename is not renewed during the grace period?  
13. Can I link multiple addresses to my Basename?  
14. I am a builder. How do I integrate Basenames to my app?  
15. How do I get a Basename for my app or project?  
16. How are Basenames built?  
17. Do Basenames work on different chains?  
18. How do I use frames with my basename?  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/block-explorers  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Block Explorers  
Arkham  
The Arkham Platform supports Base.  
Arkham is a crypto intelligence platform that systematically analyzes blockchain transactions, showing users the people and companies behind blockchain activity, with a suite of advanced tools for analyzing their activity.  
Blockscout  
A Blockscout explorer is available for Base.  
Blockscout provides tools to help you debug smart contracts and transactions:  
View, verify, and interact with smart contract source code.  
View detailed transaction information  
A testnet explorer for Base Sepolia is also available.  
Etherscan  
An Etherscan block explorer is available for Base.  
Etherscan provides tools to help you view transaction data and debug smart contracts:  
Search by address, transaction hash, batch, or token  
View, verify, and interact with smart contract source code  
View detailed transaction information  
View L1-to-L2 and L2-to-L1 transactions  
A testnet explorer for Base Sepolia is also available.  
DexGuru  
DexGuru provides a familiar UI with data on transactions, blocks, account balances and more. Developers can use it to verify smart contracts and debug transactions with interactive traces and logs visualization.  
L2scan Explorer  
L2scan Explorer is a web-based tool that allows users to analyze Base and other layer 2 networks. It provides a user-friendly interface for viewing transaction history, checking account balances, and tracking the status of network activity.  
OKLink  
OKLink is a multi-chain blockchain explorer that supports Base and provides the following features for developers:  
Search by address, transaction, block, or token  
View, verify, and interact with smart contract source code  
Access a comprehensive and real-time stream of on-chain data, including large transactions and significant fund movements  
Address labels (i.e. project labels, contract labels, risk labels, black address labels, etc.)  
Routescan  
Routescan superchain explorer allows you to search for transactions, addresses, tokens, prices and other activities taking place across all Superchain blockchains, including Base.  
Tenderly Explorer  
With the Tenderly developer explorer you can get unparalleled visibility into your smart contract code. You can easily view detailed transaction information, spot bugs in your code, and optimize gas spend. Supporting Base mainnet and Base Sepolia testnet, Tenderly Explorer helps you track your smart contracts while providing visibility on a granular level.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/bridges  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Superchain Bridges  
Superbridge  
Superbridge enables you to bridge ETH and other supported assets from Ethereum mainnet (L1) directly to Base.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Brid.gg  
Brid.gg is another option that also helps you bridge ETH and supported assets between Ethereum mainnet (L1) and Base.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
INFO  
Coinbase Technologies, Inc., provides links to the above independent service providers for your convenience but assumes no responsibility for their operations. Any interactions with these providers are solely between you and the provider.  
Programmatic Bridging  
See the sample code repository to see how to bridge ETH and ERC-20s from Ethereum to Base.  
CAUTION  
Double check the token address for ERC-20s You can use any ERC-20 that is supported on the network. You can check what assets are on Base and the corresponding contract address via this hub. Ensure there is an address for base, example. Always test with small amounts to ensure the system is working as expected.  
CAUTION  
This implementation only can bridge assets to Base. Do not attempt to alter the code to withdraw the assets.  
  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/cross-chain  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Cross-chain  
Axelar  
Axelar is an interchain platform that connects blockchains to enable universal web3 transactions. By integrating with Axelar, applications built on Base can now easily send messages and assets between the 49+ blockchains connected via Axelar.  
To learn more about Axelar visit our docs. For complete end-to-end examples demonstrating various Axelar use cases please visit the available code examples.  
Supported Networks  
Base Mainnet  
Base Testnet  
Axelarscan  
To view current transactions and live stats about the Axelar network, please visit the Axelarscan block explorer  
Crossmint  
Crossmint allows you to create and deploy NFT Collections and enable cross-chain payments. This enables your users and customers to purchase an NFT from a collection deployed on Base using Ethereum or Solana tokens.  
Check out Crossmint Docs to learn more about NFT Checkout with Crossmint. To power cross-chain payments, click here to get started.  
Supported Networks  
Base Mainnet  
Base Sepolia  
Chainlink CCIP  
Chainlink CCIP is a secure interoperability protocol that allows for securely sending messages, transferring tokens, and initiating actions across different blockchains.  
To get started with integrating Chainlink CCIP in your Base project, visit the Chainlink CCIP documentation.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
LayerZero  
LayerZero is an omnichain interoperability protocol that enables cross-chain messaging. Applications built on Base can use the LayerZero protocol to connect to 35+ supported blockchains seamlessly.  
To get started with integrating LayerZero, visit the LayerZero documentation and provided examples on GitHub.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Wormhole  
Wormhole is a generic messaging protocol that provides secure communication between blockchains.  
By integrating Wormhole, a Base application can access users and liquidity on > 30 chains and > 7 different platforms.  
See this quickstart to get started with integrating Wormhole in your Base project.  
For more information on integrating Wormhole, visit their documentation and the provided GitHub examples.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/data-indexers  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Data Indexers  
Allium  
Allium is an Enterprise Data Platform that serves accurate, fast, and simple blockchain data. Currently serving 15 blockchains and over 100+ schemas, Allium offers near real-time Base data for infrastructure needs and enriched Base data (NFT, DEX, Decoded, Wallet360) for research and analytics.  
Allium supports data delivery to multiple destinations, including Snowflake, Bigquery, Databricks, and AWS S3.  
Documentation:  
Real-time  
Batch-enriched  
To get started, contact Allium here.  
Arkham  
Arkham is a crypto intelligence platform that systematically analyzes blockchain transactions, showing users the people and companies behind blockchain activity, with a suite of advanced tools for analyzing their activity.  
References:  
Platform guide  
Whitepaper  
Codex  
Demos  
Covalent  
Covalent is a hosted blockchain data solution providing access to historical and current on-chain data for 100+ supported blockchains, including Base.  
Covalent maintains a full archival copy of every supported blockchain, meaning every balance, transaction, log event, and NFT asset data is available from the genesis block. This data is available via:  
Unified API - Incorporate blockchain data into your app with a familiar REST API  
Increment - Create and embed custom charts with no-code analytics  
To get started, sign up and visit the developer documentation.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
DipDup  
DipDup is a Python framework for building smart contract indexers. It helps developers focus on business logic instead of writing a boilerplate to store and serve data. DipDup-based indexers are selective, which means only required data is requested. This approach allows to achieve faster indexing times and decreased load on underlying APIs.  
To get started, visit the documentation or follow the quickstart guide.  
Envio  
Envio is a full-featured data indexing solution that provides application developers with a seamless and efficient way to index and aggregate real-time and historical blockchain data for any EVM. The indexed data is easily accessible through custom GraphQL queries, providing developers with the flexibility and power to retrieve specific information.  
Envio HyperSync is an indexed layer of the Base blockchain for the hyper-speed syncing of historical data (JSON-RPC bypass). What would usually take hours to sync ~100,000 events can now be done in the order of less than a minute.  
Designed to optimize the user experience, Envio offers automatic code generation, flexible language support, multi-chain data aggregation, and a reliable, cost-effective hosted service.  
To get started, visit the documentation or follow the quickstart guide.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
GhostGraph  
GhostGraph makes it easy to build blazingly fast indexers (subgraphs) for smart contracts.  
GhostGraph is the first indexing solution that lets you write your index transformations in Solidity. Base dApps can query data with GraphQL using our hosted endpoints.  
To get started, you can sign up for an account and follow this quickstart guide on how to create, deploy, and query a GhostGraph.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
The Indexing Company  
The Indexing Company provides indexing as a service, capable of indexing any chain (EVM and non-EVM) with an RPC endpoint and integrating off-chain data within the same infrastructure.  
Our services include data transformations, aggregations, and streamlined data flows, allowing teams to develop their products faster while saving on developer resources, time, and money. Our solution is ideal for teams needing advanced data engineering for modular chain setups, multi-chain products, L1/L2/L3 chains and AI.  
To get started contact us here.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Moralis  
Moralis offers comprehensive data APIs for crypto, offering both indexed and real-time data across 15+ chains. Moralis' APIs include portfolio and wallet balances, NFT data, token data, price data, candlestick data, net worth data, and a lot more. All of the data is enriched with things like metadata, parsed events and address labels.  
To get started with Moralis, you can sign up for an account, visit the Moralis documentation, or check out their tutorials on Youtube.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Nexandria  
Nexandria API offers access to complete historical on-chain data at blazing speeds, arbitrary granularity (as low as block-level) and at viable unit economics (think web2 level costs). Our technology lets you generate subgraphs on the fly, unlocking unique endpoints like a statement of all the balance transfers for all the tokens, or a list of all the neighbors of an address with all the historical interaction details or a portfolio balance graph covering all the tokens across arbitrary time/block ranges.  
References:  
API Documentation  
Sign-up  
Supported Networks  
Base Mainnet  
Shovel  
Shovel is an open source tool for synchronizing Ethereum data to your Postgres database. Shovel can index block data, transaction data, and decoded event data. A single Shovel can index multiple chains simultaneously. Shovel is configured via a declarative JSON config file – no custom functions to save indexed data to your database.  
Find out more in the Shovel Docs  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Subsquid  
Subsquid is a decentralized hyper-scalable data platform optimized for providing efficient, permissionless access to large volumes of data. It currently serves historical on-chain data, including event logs, transaction receipts, traces, and per-transaction state diffs. Subsquid offers a powerful toolkit for creating custom data extraction and processing pipelines, achieving an indexing speed of up to 150k blocks per second.  
To get started, visit the documentation or see this quickstart with examples on how to easily create subgraphs via Subsquid.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
SubQuery  
SubQuery is a data indexer that provides developers with fast, reliable, decentralized, and customized APIs for accessing rich indexed data from over 80+ ecosystems (including Base) within their projects.  
SubQuery provides the ability to aggregate this data across multiple blockchains, all within a single project.  
Other advantages of SubQuery includes performance with multiple RPC endpoint configurations, multi-worker capabilities and a configurable caching architecture.  
To get started, visit the developer documentation or follow this step-by-step guide on how to index any smart contract on Base.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
The Graph  
The Graph is an indexing protocol for organizing blockchain data and making it easily accessible with GraphQL.  
Base applications can use GraphQL to query open APIs called subgraphs, to retrieve data that is indexed on the network. With The Graph, you can build serverless applications that run entirely on public infrastructure.  
To get started, visit the documentation or see this quickstart on how to create, deploy, and query a subgraph.  
Supported Networks  
Base Mainnet  
Flair  
Flair is a real-time and historical custom data indexing for any EVM chain.  
It offers reusable indexing primitives (such as fault-tolerant RPC ingestors, custom processors and aggregations, re-org aware database integrations) to make it easy to receive, transform, store and access your on-chain data.  
To get started, visit the documentation or clone the starter boilerplate template and follow the instructions.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/ethers  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
ethers.js  
thirdweb SDK  
viem  
web3.js  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
ethers.js  
ethers.js is a JavaScript library that allows developers to interact with EVM-compatible blockchain networks.  
You can use ethers.js to interact with smart contracts deployed on the Base network.  
Install  
To install ethers.js run the following command:  
npm install --save ethers  
Setup  
Before you can start using ethers.js, you need to import it into your project.  
Add the following line of code to the top of your file to import ethers.js:  
const ethers = require('ethers');  
Connecting to Base  
You can connect to Base by instantiating a new ethers.js JsonRpcProvider object with a RPC URL of the Base network:  
const ethers = require('ethers');  
  
const url = 'https://mainnet.base.org';  
const provider = new ethers.providers.JsonRpcProvider(url);  
INFO  
To alternatively connect to Base Sepolia (testnet), change the above URL from https://mainnet.base.org to https://sepolia.base.org.  
Reading data from the blockchain  
Once you have created a provider, you can use it to read data from the Base network.  
For example, you can use the getBlockNumber method to get the latest block:  
async function getLatestBlock() {  
 const latestBlock = await provider.getBlockNumber();  
 console.log(latestBlock);  
}  
Writing data to the blockchain  
In order to write data to the Base network, you need to create a Signer.  
You can create a Signer by instantiating a new ethers.js Wallet object, providing it with a private key and Provider.  
const privateKey = 'PRIVATE\_KEY';  
const signer = new ethers.Wallet(privateKey, provider);  
INFO  
PRIVATE\_KEY is the private key of the wallet to use when creating the signer.  
Interacting with smart contracts  
You can use ethers.js to interact with a smart contract on Base by instantiating a Contract object using the ABI and address of a deployed contract:  
const abi = [  
… // ABI of deployed contract  
];  
  
const contractAddress = "CONTRACT\_ADDRESS"  
  
// read only  
const contract = new ethers.Contract(contractAddress, abi, provider);  
For write-only contracts, provide a Signer object instead of a Provider object:  
// write only  
const contract = new ethers.Contract(contractAddress, abi, signer);  
INFO  
CONTRACT\_ADDRESS is the address of the deployed contract.  
Once you have created a Contract object, you can use it to call desired methods on the smart contract:  
async function setValue(value) {  
 const tx = await contract.set(value);  
 console.log(tx.hash);  
}  
  
async function getValue() {  
 const value = await contract.get();  
 console.log(value.toString());  
}  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/foundry  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Foundry  
Hardhat  
thirdweb CLI  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Foundry  
Foundry is a smart contract development toolchain.  
With Foundry you can manage your dependencies, compile your project, run tests, deploy smart contracts, and interact with the chain from the command-line and via Solidity scripts.  
Check out the Foundry Book to get started with using Foundry with Base.  
Using Foundry with Base  
Foundry supports Base out-of-the-box.  
Just provide the Base RPC URL and Chain ID when deploying and verifying your contracts.  
Mainnet  
Deploying a smart contract  
forge create ... --rpc-url=https://mainnet.base.org/  
Verifying a smart contract  
forge verify-contract ... --chain-id 8453  
Testnet  
Deploying a smart contract  
forge create ... --rpc-url=https://sepolia.base.org/  
Verifying a smart contract  
forge verify-contract ... --chain-id 84532  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/hardhat  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Foundry  
Hardhat  
thirdweb CLI  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Hardhat  
Hardhat is an Ethereum development environment for flexible, extensible, and fast smart contract development.  
You can use Hardhat to edit, compile, debug, and deploy your smart contracts to Base.  
Using Hardhat with Base  
To configure Hardhat to deploy smart contracts to Base, update your project’s hardhat.config.ts file by adding Base as a network:  
networks: {  
 // for mainnet  
 "base-mainnet": {  
 url: 'https://mainnet.base.org',  
 accounts: [process.env.PRIVATE\_KEY as string],  
 gasPrice: 1000000000,  
 },  
 // for Sepolia testnet  
 "base-sepolia": {  
 url: "https://sepolia.base.org",  
 accounts: [process.env.PRIVATE\_KEY as string],  
 gasPrice: 1000000000,  
 },  
 // for local dev environment  
 "base-local": {  
 url: "http://localhost:8545",  
 accounts: [process.env.PRIVATE\_KEY as string],  
 gasPrice: 1000000000,  
 },  
 },  
 defaultNetwork: "base-local",  
INFO  
For a complete guide on using Hardhat to deploy contracts on Base, see Deploying a Smart Contract.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/network-faucets  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Network Faucets  
Coinbase Developer Platform  
The Coinbase Developer Platform Faucet provides free testnet ETH on Base Sepolia - one claim per 24 hours.  
INFO  
Requests to Coinbase Developer Platform's Faucet are limited to one claim per 24 hours.  
thirdweb Faucet  
The thirdweb Faucet provides free testnet ETH on Base Sepolia - one claim per 24 hours.  
INFO  
The thirdweb faucet allows developers to connect their wallet through EOA or social logins and claim Base Sepolia testnet funds.  
Superchain Faucet  
The Superchain Faucet provides testnet ETH for all OP Chains, including Base.  
INFO  
The Superchain faucet allows developers to authenticate via their onchain identity. Developers that choose to authenticate via their onchain identity can claim more testnet ETH versus traditional faucets. For more information, see the FAQ.  
Alchemy Faucet  
The Alchemy Faucet is a fast and reliable network faucet that allows users with a free Alchemy account to request testnet ETH on Base Sepolia.  
INFO  
Requests to Alchemy's Base Sepolia Faucet are limited to one claim per 24 hours.  
Bware Labs Faucet  
Bware Labs Faucet is an easy to use faucet with no registration required. You can use Bware Labs Faucet to claim Base Sepolia testnet ETH for free - one claim per 24 hours.  
INFO  
Requests to Bware Labs Faucet are limited to one claim per 24 hours.  
QuickNode Faucet  
QuickNode Faucet is an easy to use Multi-Chain Faucet. You can use QuickNode Faucet to claim Base Sepolia testnet ETH for free - one drip per network every 12 hours.  
INFO  
Requests to QuickNode Faucet are limited to one drip every 12 hours.  
LearnWeb3 Faucet  
LearnWeb3 Faucet is a multi-chain faucet by LearnWeb3. You can use the LearnWeb3 faucet to claim Base Sepolia testnet ETH for free - one claim every 24 hours.  
INFO  
Requests to LearnWeb3 faucet are limited to one claim per 24 hours.  
Ethereum Ecosystem Faucet  
The Base Sepolia Faucet is a free & easy to use testnet faucet for Base Sepolia with very generous drips that doesn't require users to log in. It's run by Ethereum Ecosystem.  
INFO  
Each wallet is restricted to receiving 0.5 ETH from this faucet every 24 hours.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/node-providers  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Node Providers  
Coinbase Developer Platform (CDP)  
CDP provides an RPC endpoint that runs on the same node infrastructure that powers Coinbase's retail exchange, meaning you get the rock solid reliability of our retail exchange as a developer. CDP gives you a free, rate limited RPC endpoint to begin building on Base.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
1RPC  
1RPC is the first and only on-chain attested privacy preserving RPC that eradicates metadata exposure and leakage when interacting with blockchains. 1RPC offers free and paid plans with additional features and increased request limits.  
Supported Networks  
Base Mainnet  
Alchemy  
Alchemy is a popular API provider and developer platform. Its robust, free tier offers access to enhanced features like SDKs, JSON-RPC APIs, and hosted mainnet and testnet nodes for Base.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
All That Node  
All That Node is a comprehensive multi-chain development suite, designed to support multiple networks from a single platform. They offer free and paid plans with additional features and increased request limits.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Ankr  
Ankr provides private and public RPC endpoints for Base, powered by a globally distributed and decentralized network of nodes. They offer free and paid plans with increased request limits.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Blast  
Blast provides fast and reliable decentralized blockchain APIs by partnering with third-party Node Providers. Blast offers users the ability to generate their own dedicated RPC endpoint for Base.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Blockdaemon  
Blockdaemon offers access to hosted Base nodes with a free plan at $0/month via the Ubiquity Data API Suite. Extra costs may be incurred depending on usage.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
BlockPI  
BlockPI is a high-quality, robust, and efficient RPC service network that provides access to Base nodes with free and paid plans.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Chainstack  
Chainstack allows developers to run high-performing Base nodes and APIs in minutes. They offer elastic Base RPC nodes that provide personal, geographically diverse, and protected API endpoints, as well as archive nodes to query the entire history of the Base Mainnet. Get started with their free and paid pricing plans.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
DRPC  
DRPC offers access to a distributed network of independent third-party partners and public nodes for Base. They provide a free tier that allows for an unlimited amount of requests over public nodes, or a paid tier which provides access to all providers, as well as other additional features.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
GetBlock  
GetBlock is a Blockchain-as-a-Service (BaaS) platform that provides instant API access to full nodes for Base. They offer free, pay per use, and unlimited pricing plans.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
NodeReal  
NodeReal is a blockchain infrastructure and services provider that provides instant and easy-access to Base node APIs.  
Supported Networks  
Base Mainnet  
Nodies DLB  
Nodies DLB provides highly performant RPC Services for Base, as well as all other OP-stacked chains. They offer free public endpoints, Pay-As-You-Go, and enterprise pricing plans.  
Supported Networks  
Base Mainnet  
Base Testnet (Available on request)  
NOWNodes  
NOWNodes is a Web3 development tool that provides shared and dedicated no rate-limit access to Base RPC full nodes.  
Supported Networks  
Base Mainnet  
OnFinality  
OnFinality provides high performance archive access to Base Mainnet and Base Sepolia, with a generous free tier and high rate limits, as well as Trace and Debug APIs, available to paid plans.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
QuickNode  
QuickNode offers access to hosted Base nodes as part of their free Discover Plan. You can configure add-ons, like "Trace Mode" and "Archive Mode" for an additional cost by upgrading to one of their paid plans.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
RockX  
RockX offers a global blockchain node network and developer tools for onchain innovation. Start with our free Base RPC to access institutional-grade solutions.  
Supported Networks  
Base Mainnet  
Stackup  
Stackup is a leading ERC-4337 infrastructure platform. You can access hosted Base nodes with built-in account abstraction tools like bundlers and paymasters.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
SubQuery  
SubQuery is a globally distributed, decentralized network of RPC nodes, offering generous free public endpoints and higher access through Flex Plans  
Supported Networks  
Base Mainnet  
Tenderly Web3 Gateway  
Tenderly Web3 Gateway provides a fast and reliable hosted node solution with a built-in suite of developer tooling and infrastructure building blocks covering your whole development lifecycle. Develop, test, deploy, and monitor your onchain app on the Base network with both free and paid plans.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Unifra  
Unifra is a Web3 developer platform that provides tools, APIs, and node infrastructure, and provides access to Base nodes that are nodes are reliable, scalable, and easy to use.  
Supported Networks  
Base Mainnet  
Validation Cloud  
Validation Cloud is the world’s fastest node provider according to Compare Nodes. With 50 million compute units available for use without a credit card and a scale tier that never has rate limits, Validation Cloud is built to support your most rigorous and low-latency workloads.  
Supported Networks  
Base Mainnet  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/onboarding  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
User Onboarding  
Crossmint  
Crossmint is an enterprise-grade web3 development platform that lets you deploy smart contracts, create email wallets, enable credit-card and cross chain payments, and use APIs to create, distribute, sell, store, and edit NFTs. By abstracting away the core complexities of the Blockchain, Crossmint allows you to build NFT applications without requiring any blockchain experience or holding cryptocurrency, and making the blockchain invisible to end users. Crossmint enables you to provide a Web2 experience for your Web3 apps and onboard users.  
Dynamic  
Dynamic is a login platform designed for seamless user onboarding. It offers smart and simple login flows for both crypto-native and non-native users. Dynamic features support for non-custodial embedded wallets and consolidating multiple wallets under a single user account. The Dynamic platform is compatible with most EVM chains, including Base.  
Openfort  
Openfort is an infrastructure provider designed to simplify the development of games and gamified experiences across their suite of API endpoints. Authenticated users can instantly access the embedded, non-custodial smart account natively in the game and sign blockchain transactions with one button. The Oepnfort platform is compatible with most EVM chains, including Base.  
Use Auth Guide to allow several onboarding methods into your game regardless of the platform.  
Privy  
Privy is a library designed for progressive user onboarding and authentication. It enables users to connect to your app using traditional methods such as email addresses, phone numbers, or social profiles, as well as through web3 methods like crypto wallets. Additionally, Privy supports embedded wallets, eliminating the need for users to have a self-custodial wallet prior to exploring your app. Privy is compatible with most EVM chains, including Base.  
You can get started with Privy here, and check out these starter repos for building a Progressive Web App (PWA) on Base and using the Base Paymaster with Privy.  
Particle Network  
Particle Network is the intent-centric, modular access layer of Web3. With Particle's Smart Wallet-as-a-Service, developers can curate a seamless user experience through modular and customizable EOA/AA embedded wallet components. Using MPC-TSS for key management, Particle can streamline user onboarding via familiar web2 accounts - such as Google accounts, email addresses, and phone numbers. Particle Network's Smart Wallet-as-a-Service is compatible with most EVM chains, including Base.  
Sequence  
Sequence is an all-in-one development platform for integrating web3 into games. Onboard, monetize, grow, and retain players with Sequence’s award-winning technology including: non-custodial Embedded Wallets, white labeled marketplaces and marketplace API, indexer, relayer, node gateway, Unity/Unreal/React Native/Mobile SDKs, transaction API, contract deployment, analytics, and more. Learn more here and start creating on Sequence Builder now.  
thirdweb  
thirdweb is the full stack open source web3 solution for bringing web3 into ANY consumer application on ANY platform. Utilize our wide range of sdks on web, mobile, Unity/Unreal or through our cloud hosted engine service! Connect your users with EOA or social logins, create contracts for marketplaces or tokenize in-game items, handle thousands of transactions to build apps that scale, and provide a fiat onramper for your users.  
WalletKit  
WalletKit is an all-in-one platform for adding smart, gasless wallets to your app. WalletKit offers pre-built components for onboarding users with email and social logins, which can be integrated in under 15 minutes using their React SDK or the wagmi connector. Alternatively, build completely bespoke experiences for your users using WalletKit's Wallets API.  
WalletKit is compatible with most EVM chains, including Base. It has integrated support for ERC 4337 and comes with a paymaster and bundler included, requiring no extra setup.  
You can check out the WalletKit documentation here. Start building for free on the Base testnet today.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/onramps  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Onramps  
Coinbase Onramp  
Coinbase Onramp is a fiat-to-crypto onramp that allows users to buy or transfer crypto directly from self-custody wallets and apps. Coinbase Onramp supports 60+ fiat currencies with regulatory compliance and licensing, as well as 100+ cryptocurrencies, including ETH on Base. Get started here to use the Coinbase Developer Platform.  
MoonPay  
MoonPay is a crypto onramp that provides global coverage, seamless revenue sharing, and zero risk of fraud or chargebacks. MoonPay supports 30+ fiat currencies and 110+ cryptocurrencies, including ETH on Base.  
Onramp  
Onramp is a fiat-to-crypto payment gateway, which helps users seamlessly convert fiat currency to the desired cryptocurrency. Onramp currently supports 300+ cryptocurrencies and 20+ blockchain networks, including ETH on Base.  
Ramp  
Ramp is an onramp and offramp that empowers users to buy & sell crypto inside your app. Ramp supports 40+ fiat currencies and 90+ crypto assets, including ETH on Base.  
Transak  
Transak is a developer integration toolkit to let users buy/sell crypto in any app, website or web plugin. It is available across 170 cryptocurrencies on 75+ blockchains, including ETH on Base.  
Alchemy Pay  
Alchemy Pay (ACH) is a payment solutions provider that seamlessly connects fiat and crypto economies for global consumers, merchants, developers, and institutions.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/oracles  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Oracles  
API3  
API3 is building secure first-party oracles for Base.  
API3 is live with:  
dAPIs: First-party aggregated data feeds sourced directly from the data providers.  
Airnode: The first-party serverless Oracle solution to bring any REST API onchain.  
QRNG: Quantum Random Number Generator for verifiable quantum RNG onchain.  
Supported Networks  
Base Mainnet  
Chainlink  
Chainlink provides a number of price feeds for Base.  
See this guide to learn how to use the Chainlink feeds.  
INFO  
To use Chainlink datafeeds, you may need LINK token.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Chronicle  
Chronicle provides a number of Oracles for Base.  
See this guide to learn how to use the Chronicle Oracles.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
DIA  
DIA provides 2000+ price feeds for Base. See this guide to learn how to use the DIA feeds.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
Gelato  
Gelato VRF (Verifiable Random Function) provides a unique system offering trustable randomness on Base.  
See this guide to learn how to get started with Gelato VRF.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
ORA  
ORA provides an Onchain AI Oracle for Base.  
See this guide to learn how to use ORA Onchain AI Oracle.  
Supported Networks  
Base Mainnet  
Pyth  
Pyth offers 250+ price feeds for Base.  
See this guide to learn how to use the Pyth feeds.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
RedStone  
RedStone provides 1200+ price feeds for Base.  
See this guide to learn how to use the RedStone feeds.  
Supported Networks  
Base Mainnet  
Supra  
Supra provides VRF and decentralized oracle price feeds that can be used for onchain and offchain use-cases such as spot and perpetual DEXes, lending protocols, and payments protocols. Supra’s oracle chain and consensus algorithm makes it one of the fastest-to-finality oracle providers, with layer-1 security guarantees. The pull oracle has a sub-second response time. Aside from speed and security, Supra’s rotating node architecture gathers data from 40+ data sources and applies a robust calculation methodology to get the most accurate value. The node provenance on the data dashboard also provides a fully transparent historical audit trail. Supra’s Distributed Oracle Agreement (DORA) paper was accepted into ICDCS 2023, the oldest distributed systems conference.  
Visit the Supra documentation to learn more about integrating Supra's oracle and VRF into your Base project.  
Supported Networks  
Base Mainnet  
Base Sepolia (Testnet)  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/registry-api  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Onchain Registry API  
Onchain Registry FAQ  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Onchain Registry API  
INFO  
The base url for our API endpoints is https://base.org/api/registry/. The use of Onchain Registry API is governed by the license terms outlined in our Terms & Conditions.  
Instructions  
Users of this API can use the /entries and /featured endpoints to display Onchain Registry entries on their own surfaces  
If your team would like to use referral codes to point your users to entries, we recommend appending your referral code to the link provided in the target\_url field  
If your team would like to filter entries based on where they are hosted or by creator, we recommend implementing logic based on the target\_url and creator\_name fields  
Endpoints  
GET /entries  
This endpoint will display all Onchain Registry entries subject to any query parameters set below  
Query Parameters  
Name Type Description  
page number The page number (default 1)  
limit number The number of entries per page (default 10)  
category array The category or categories of the entries of interest  
(Options: Games, Social, Creators, Finance, Media)  
curation string The entry’s level of curation  
(Options: Featured, Curated, Community)  
Response  
 "data": [  
 {  
 "id": "7AsRdN8uf601fCkH1e084F",  
 "category": "Creators",  
 "content": {  
 "title": "Based Project",  
 "short\_description": "Short description of this based project with max char count of 30",  
 "full\_description": "Full description of this based project with max char count of 200",  
 "image\_url": "https://base.org/image.png",  
 "target\_url": "https://base.org/target-page",  
 "cta\_text": "Mint",  
 "function\_signature": "mint(uint256)",  
 "contract\_address": "0x1FC10ef15E041C5D3C54042e52EB0C54CB9b710c",  
 "token\_id": "2",  
 "token\_amount": "0.01",  
 "featured": true,  
 "creator\_name": "Base",  
 "creator\_image\_url": "https://base.org/creator-image.png",  
 "curation": "featured",  
 "start\_ts": "2024-06-25T04:00:00Z",  
 "expiration\_ts": "2024-07-29T00:00:00Z"  
 },  
 "updated\_at": null,  
 "created\_at": "2024-07-10T18:20:42.000Z"  
 },  
 {  
 "id": "8fRbdN8uf601fCkH1e084F",  
 "category": "Games",  
 "content": {  
 "title": "Based Project II",  
 "short\_description": "Short description of this second based project with max char count of 30",  
 "full\_description": "Full description of this second based project with max char count of 200",  
 "image\_url": "https://base.org/image2.png",  
 "target\_url": "https://base.org/second-target-page",  
 "cta\_text": "Mint",  
 "function\_signature": "mint(uint256)",  
 "contract\_address": "0x1FC10ef15E041C5D3C54042e52EB0C54CB9b710c",  
 "token\_id": "1",  
 "token\_amount": "0.005",  
 "featured": false,  
 "creator\_name": "Base",  
 "creator\_image\_url": "https://base.org/creator-image2.png",  
 "curation": "community",  
 "start\_ts": "2024-06-25T04:00:00Z",  
 "expiration\_ts": "2024-07-29T00:00:00Z"  
 },  
 "updated\_at": "2024-07-11T18:20:42.000Z",  
 "created\_at": "2024-07-10T18:20:42.000Z"  
 }  
 ],  
 "pagination": {  
 "total\_records": 2,  
 "current\_page": 1,  
 "total\_pages": 1,  
 "limit": 10  
 }  
}  
GET /featured  
This endpoint will display a single Onchain Registry entry that is being actively featured  
Response  
 "data": {  
 "id": "7AsRdN8uf601fCkH1e084F",  
 "category": "Creators",  
 "content": {  
 "title": "Based Project",  
 "short\_description": "Short description of this based project with max char count of 30",  
 "full\_description": "Full description of this based project with max char count of 200",  
 "image\_url": "https://base.org/image.png",  
 "target\_url": "https://base.org/target-page",  
 "cta\_text": "Mint",  
 "function\_signature": "mint(uint256)",  
 "contract\_address": "0x1FC10ef15E041C5D3C54042e52EB0C54CB9b710c",  
 "token\_id": "2",  
 "token\_amount": "0.01",  
 "featured": true,  
 "creator\_name": "Base",  
 "creator\_image\_url": "https://base.org/creator-image.png",  
 "curation": "featured",  
 "start\_ts": "2024-06-25T04:00:00Z",  
 "expiration\_ts": "2024-07-29T00:00:00Z"  
 },  
 "updated\_at": null,  
 "created\_at": "2024-07-10T18:20:42.000Z"  
 }  
}  
Entry Schema  
Name Type Description  
id string Unique entry ID  
category string The category of the entry  
(Options: Games, Social, Creators, Finance, Media)  
title string The title of the entry  
short\_description string Short version of the entry description (max 30 char)  
full\_description string Full version of the entry description (max 200 char)  
image\_url string URL of the entry’s featured image  
target\_url string URL for the entry’s desired user action  
cta\_text string This is the type of user action for the entry  
(Options: Play, Mint, Buy, Trade, Explore)  
function\_signature string The function signature associated with the desired user action on the entry’s contract  
contract\_address string The contract address associated with the entry  
token\_id string The token ID if this is an ERC-1155  
token\_amount string The price of the entry’s desired user action  
featured boolean A true or false based on whether the entry is actively featured  
creator\_name string The name of the entry’s creator  
creator\_image\_url string The logo of the entry’s creator  
curation string The entry’s level of curation  
  
Options:  
Featured - one entry per day with top placement  
Curated - community entries being  
Community - all other community entries  
start\_ts string The UTC timestamp that the entry is open to users  
expiration\_ts string The UTC timestamp that the entry is no longer open to users  
updated\_at string || null The UTC timestamp that the entry was last updated (null if the entry has not been updated since creation)  
created\_at string The UTC timestamp that the entry was created  
Terms & Conditions  
We grant third parties a non-exclusive, worldwide, royalty-free license to use the Onchain Registry API solely for the purpose of integrating it into their applications or services. This license does not extend to any data or content accessed through the Onchain API, which remains the sole responsibility of the third party. By using the Onchain Registry API, third parties agree to comply with our license terms and any applicable laws and regulations as set forth in Coinbase Developer Platform Terms of Service. We make no warranties regarding the Onchain Registry API, and users accept all risks associated with its use. The Onchain App Registry API is an Early Access Product per Section 18 of the Coinbase Developer Platform Terms of Service and the Coinbase Prohibited Use Policy, and all terms and conditions therein govern your use of the Onchain Registry API.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/registry-faq  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Onchain Registry API  
Onchain Registry FAQ  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Onchain Registry FAQ  
What happens after I submit my project?  
We want to keep the Registry as open and permissionless as we can, but we also need to run entries through a lightweight review process to make sure we are protecting our end users as much as reasonably possible. All reviews are queued based on order of submission. Review times will vary based on the complexity of the entry.  
How do I edit my entry if I made a mistake?  
To edit your original entry, use our Registry edit form. This form will ask you for your original entry ID, which you can find at the bottom of your original entry's confirmation email. On the edit form, you only need to fill in the fields that you would like to change.  
Why was my entry rejected?  
Some common examples of projects we cannot feature at this time include:  
Expired NFT mints  
ERC-404 collections (e.g. tokens that can be used as fractionalized NFT assets)  
Platforms that offer investment strategies/financial advice or involve the offer, sale or distribution of unregistered securities in violation of applicable securities laws  
Services, products or platforms that do not comply with the Coinbase Prohibited Use Policy  
How can I pull entries from the Registry?  
We offer an Onchain Registry API that gives users the ability to display Onchain Registry entries on their own surfaces. Our Onchain Registry API docs can be found here.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/thirdweb-cli  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Foundry  
Hardhat  
thirdweb CLI  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
thirdweb CLI  
thirdweb provides an interactive command line interface, allowing you to create, build and deploy your smart contracts and apps.  
You can use the thirdweb CLI to create and deploy smart contracts to the Base network.  
Visit the thirdweb documentation for more instructions on using the thirdweb CLI.  
Creating a project  
Create a new project with thirdweb installed and configured:  
npx thirdweb create  
INFO  
When you create a project for smart contracts or web3 apps there are various configurable options.  
For contracts, some options are:  
Create a new contract project using Hardhat or Forge  
Add a new contract to an existing project  
Start from an audited contract base, and add optional extensions  
For contracts, some options are:  
Front end applications using Next, CRA or Vite  
Backend applications using Node.js or Express.js  
Choice of TypeScript or JavaScript variants  
Deploying a smart contract  
Deploy your smart contracts to the Base network:  
npx thirdweb deploy  
INFO  
To deploy to the Base network, after running npx thirdweb deploy, visit the provided dashboard URL and select Base from the Network dropdown.  
INFO  
For a complete guide on using the thirdweb CLI to create and deploy contracts on Base, see Deploy a smart contract on Base testnet.  
Publishing a smart contract  
Publish and share a versioned release of your contract onto thirdweb’s registry:  
npx thirdweb publish  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/thirdweb-sdk  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
ethers.js  
thirdweb SDK  
viem  
web3.js  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
thirdweb SDK  
thirdweb SDK is a library that enables developers to build web3 applications and interact with any EVM-compatible blockchain.  
You can use the thirdweb SDK to build apps and interact with smart contracts deployed on the Base network.  
The thirdweb SDK is available in various programming languages, including: React, React Native, TypeScript, Python, Go, and Unity.  
Visit the thirdweb documentation for more instructions on using the thirdweb SDKs.  
Install  
To install the thirdweb SDK, run:  
npm install @thirdweb-dev/sdk ethers@5  
Initializing the SDK with Base  
To get started using the SDK, you must first initialize an instance of ThirdWebSDK, and connect to the Base network by passing in the Base chain.  
To initialize the SDK with the Base network and get a contract:  
import { Base } from '@thirdweb-dev/chains';  
import { ThirdwebSDK } from '@thirdweb-dev/sdk/evm';  
  
const sdk = new ThirdwebSDK(Base);  
const contract = await sdk.getContract('0x0000000000000000000000000000000000000000');  
INFO  
The code snippet above uses the React SDK. The thirdweb SDKs are also available in React Native, TypeScript, Python, Go, and Unity.  
If alternatively you'd like to initialize the SDK with Base Sepolia (testnet), use the following code instead:  
import { BaseSepoliaTestnet } from '@thirdweb-dev/chains';  
import { ThirdwebSDK } from '@thirdweb-dev/sdk/evm';  
  
const sdk = new ThirdwebSDK(BaseSepoliaTestnet);  
const contract = await sdk.getContract('0x0000000000000000000000000000000000000000');  
Interacting with smart contracts  
Once you initialize the SDK and connect to a smart contract deployed to Base, you can start calling functions on it using the SDK.  
INFO  
Any interaction you make with a smart contract will be made from the connected wallet automatically.  
Using contract extension functions  
The thirdweb SDK provides convenience functions when your smart contract uses extensions. This is the easiest way to read data and write transactions with your smart contracts.  
For example, if your contract implements the ERC721 extension, you can utilize all of the functions of the corresponding erc721 standard in the SDK.  
As an example, below is a code snippet that uses useOwnedNFTs hook to get a list of NFTs owned by a single wallet address:  
import { useOwnedNFTs } from '@thirdweb-dev/react';  
  
const { data, isLoading, error } = useOwnedNFTs(contract, '{{wallet\_address}}');  
Usage  
import { useOwnedNFTs, useContract, useAddress } from '@thirdweb-dev/react';  
  
// Your smart contract address  
const contractAddress = '{{contract\_address}}';  
  
function App() {  
 const address = useAddress();  
 const { contract } = useContract(contractAddress);  
 const { data, isLoading, error } = useOwnedNFTs(contract, address);  
}  
For more examples on using contract extension functions, visit the thirdweb developer documentation.  
Reading contract data  
If your contract doesn’t use any extensions, or you want to directly call functions on your smart contract to read data, you can use the useContractRead hook.  
Read data on your contract from a connected wallet:  
const { contract } = useContract('{{contract\_address}}');  
const { data: myData, isLoading } = useContractRead(contract, 'myFunction');  
Writing transactions  
If your contract doesn’t use any extensions, or you want to directly call functions on your smart contract to write data, you can use the useContractWrite hook.  
Make transactions on your contract from a connected wallet:  
const { contract } = useContract('{{contract\_address}}');  
const { mutateAsync: myFunctionAsync } = useContractWrite(contract, 'myFunction');  
const tx = await myFunctionAsync(['argument1', 'argument2']); // Call the function  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/viem  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
ethers.js  
thirdweb SDK  
viem  
web3.js  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
viem  
INFO  
Viem is currently only available on Base Sepolia testnet.  
viem a TypeScript interface for Ethereum that provides low-level stateless primitives for interacting with Ethereum.  
You can use viem to interact with smart contracts deployed on Base.  
Install  
To install viem run the following command:  
npm install --save viem  
Setup  
Before you can start using viem, you need to setup a Client with a desired Transport and Chain.  
import { createPublicClient, http } from 'viem';  
import { base } from 'viem/chains';  
  
const client = createPublicClient({  
 chain: base,  
 transport: http(),  
});  
INFO  
To use Base, you must specify base as the chain when creating a Client.  
To use Base Sepolia (testnet), replace base with baseSepolia.  
Reading data from the blockchain  
Once you have created a client, you can use it to read and access data from Base using Public Actions  
Public Actions are client methods that map one-to-one with a "public" Ethereum RPC method (eth\_blockNumber, eth\_getBalance, etc.)  
For example, you can use the getBlockNumber client method to get the latest block:  
const blockNumber = await client.getBlockNumber();  
Writing data to the blockchain  
In order to write data to Base, you need to create a Wallet client (createWalletClient) and specify an Account to use.  
import { createWalletClient, custom } from 'viem'  
import { base } from 'viem/chains'  
  
const [account] = await window.ethereum.request({ method: 'eth\_requestAccounts' })  
  
const client = createWalletClient({  
 account,  
 chain: base,  
 transport: custom(window.ethereum)  
})  
  
client.sendTransaction({ ... })  
INFO  
In addition to making a JSON-RPC request (eth\_requestAccounts) to get an Account, viem provides various helper methods for creating an Account, including: privateKeyToAccount, mnemonicToAccount, and hdKeyToAccount.  
To use Base Sepolia (testnet), replace base with baseSepolia.  
Interacting with smart contracts  
You can use viem to interact with a smart contract on Base by creating a Contract instance using getContract and passing it the contract ABI, contract address, and Public and/or Wallet Client:  
import { getContract } from 'viem';  
import { wagmiAbi } from './abi';  
import { publicClient } from './client';  
  
// 1. Create contract instance  
const contract = getContract({  
 address: 'CONTRACT\_ADDRESS',  
 abi: wagmiAbi,  
 publicClient,  
});  
  
// 2. Call contract methods, listen to events, etc.  
const result = await contract.read.totalSupply();  
INFO  
CONTRACT\_ADDRESS is the address of the deployed contract.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/tools/web3  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
ethers.js  
thirdweb SDK  
viem  
web3.js  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
web3.js  
web3.js is a JavaScript library that allows developers to interact with EVM-compatible blockchain networks.  
You can use web3.js to interact with smart contracts deployed on the Base network.  
Install  
To install web3.js run the following command:  
npm install web3  
Setup  
Before you can start using web3.js, you need to import it into your project.  
Add the following line of code to the top of your file to import web3.js:  
//web3.js v1  
const Web3 = require('web3');  
  
//web3.js v4  
const { Web3 } = require('web3');  
Connecting to Base  
You can connect to Base by instantiating a new web3.js Web3 object with a RPC URL of the Base network:  
const { Web3 } = require('web3');  
  
const web3 = new Web3('https://mainnet.base.org');  
INFO  
To alternatively connect to Base Sepolia (testnet), change the above URL from https://mainnet.base.org to https://sepolia.base.org.  
Accessing data  
Once you have created a provider, you can use it to read data from the Base network.  
For example, you can use the getBlockNumber method to get the latest block:  
async function getLatestBlock(address) {  
 const latestBlock = await web3.eth.getBlockNumber();  
 console.log(latestBlock.toString());  
}  
Deploying contracts  
Before you can deploy a contract to the Base network using web3.js, you must first create an account.  
You can create an account by using web3.eth.accounts:  
const privateKey = “PRIVATE\_KEY”;  
const account = web3.eth.accounts.privateKeyToAccount(privateKey);  
INFO  
PRIVATE\_KEY is the private key of the wallet to use when creating the account.  
Interacting with smart contracts  
You can use web3.js to interact with a smart contract on Base by instantiating a Contract object using the ABI and address of a deployed contract:  
const abi = [  
… // ABI of deployed contract  
];  
  
const contractAddress = "CONTRACT\_ADDRESS"  
  
const contract = new web3.eth.Contract(abi, contractAddress);  
Once you have created a Contract object, you can use it to call desired methods on the smart contract:  
async function setValue(value) {  
 // write query  
 const tx = await contract.methods.set(value).send();  
 console.log(tx.transactionHash);  
}  
  
async function getValue() {  
 // read query  
 const value = await contract.methods.get().call();  
 console.log(value.toString());  
}  
INFO  
For more information on deploying contracts on Base, see Deploying a Smart Contract.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/docs/using-base  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
About Base  
Using Base  
Notices  
Preparing for fault proofs on Base Sepolia  
Building on Base  
Quickstart: Deploy on Base  
Network Information  
Base Contracts  
Fees  
Differences between Ethereum and Base  
Decentralizing Base with Optimism  
Tools  
Node Providers  
Block Explorers  
Network Faucets  
Oracles  
Data Indexers  
Cross-chain  
Account Abstraction  
Onramps  
User Onboarding  
Superchain Bridges  
Onchain Registry  
Basenames  
Toolchains  
Clients  
Tokens  
Bridging an L1 token to Base  
Adding tokens to Coinbase Wallet  
Contracts  
Security  
Status  
Brand Kit  
Terms of Service  
Privacy Policy  
Cookie Policy  
Using Base with your wallet  
Coinbase Wallet  
The Coinbase Wallet browser extension provides support for Base by default.  
To use Base with Coinbase Wallet:  
Open the Coinbase Wallet browser extension and log in to your account.  
Connect to an app using Coinbase Wallet.  
Open the network selection menu by clicking the network icon in the upper right-hand corner.  
Select Base.  
Your active network should now be switched to Base.  
Other wallets  
Base can be added as a custom network to any EVM-compatible wallet (i.e. MetaMask).  
MetaMask  
To add Base as a custom network to MetaMask:  
Open the MetaMask browser extension.  
Open the network selection dropdown menu by clicking the dropdown button at the top of the extension.  
Click the Add network button.  
Click Add a network manually.  
In the Add a network manually dialog that appears, enter the following information for Base mainnet:  
Name Value  
Network Name Base Mainnet  
Description The public mainnet for Base.  
RPC Endpoint https://mainnet.base.org  
Chain ID 8453  
Currency Symbol ETH  
Block Explorer https://base.blockscout.com/  
Tap the Save button to save Base as a network.  
You should now be able to connect to the Base by selecting it from the network selection dropdown menu.  
Testnet  
Coinbase Wallet browser extension provides support for Base Sepolia testnet by default.  
To use Base Sepolia with Coinbase Wallet:  
Open the Coinbase Wallet browser extension and log in to your account.  
Connect to an app using Coinbase Wallet.  
Open the network selection menu by clicking the network icon in the upper right-hand corner.  
Click the More networks button.  
Navigate to the Testnets tab.  
Select Base Sepolia.  
Your active network should now be switched to Base testnet.  
Other wallets  
Base Sepolia can be added as a custom network to any EVM-compatible wallet (i.e. MetaMask).  
MetaMask  
To add Base Sepolia as a custom network to MetaMask:  
Open the MetaMask browser extension.  
Open the network selection dropdown menu by clicking the dropdown button at the top of the extension.  
Click the Add network button.  
Click Add a network manually.  
In the Add a network manually dialog that appears, enter the following information for the Base Sepolia testnet:  
Name Sepolia  
Network Name Base Sepolia  
RPC Endpoint https://sepolia.base.org  
Chain ID 84532  
Currency Symbol ETH  
Block Explorer https://sepolia-explorer.base.org  
Tap the Save button to save Base Sepolia as a network.  
You should now be able to connect to the Base testnet by selecting it from the network selection dropdown menu.  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/account-abstraction-with-biconomy  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Account Abstraction on Base using Biconomy  
This page will guide you through the process of implementing Account Abstraction in your Base projects using Biconomy paymasters, bundlers, and smart accounts.  
Objectives  
By the end of this tutorial you should be able to do the following:  
Set up a smart contract project for Base using Foundry  
Set up a Next.js frontend project using create next-app  
Setup user login and authentication using Particle Network  
Setup a Biconomy paymaster and bundler  
Create a gasless transaction  
Prerequisites  
Foundry  
This tutorial requires you to have Foundry installed.  
From the command-line (terminal), run: curl -L https://foundry.paradigm.xyz | bash  
Then run foundryup, to install the latest (nightly) build of Foundry  
For more information, see the Foundry Book installation guide.  
Coinbase Wallet  
This tutorial requires you to have a wallet. You can create a wallet by downloading the Coinbase Wallet browser extension:  
Download Coinbase Wallet  
Wallet funds  
To complete this tutorial, you will need to fund a wallet with ETH on Base Goerli.  
The ETH is required for covering gas fees associated with deploying smart contracts to the network.  
To fund your wallet with ETH on Base Goerli, visit a faucet listed on the Base Faucets page.  
What is Biconomy?  
Biconomy is a toolkit that offers a full-stack solution for Account Abstraction, including smart accounts, paymasters for sponsoring gas fees, and bundlers for bundling user operations into a single transaction.  
High-level concepts  
Account Abstraction  
Account Abstraction (ERC-4337) allows users to use Smart Contract wallets instead of traditional Externally Owned Account (EOA) wallets.  
Smart Accounts  
A smart account (also known as a smart contract wallet) is a wallet that stores and manages digital assets (ERC-20 tokens, NFTs, etc.) using a smart contract.  
User Operations  
A user operation is a pseudo-transaction object sent by a smart account that describes a transaction to be sent. Multiple user operations are eventually bundled together and initiated as a single real transaction by a bundler.  
Paymaster  
A paymaster is a special smart contract that allows applications to “sponsor user operations”, meaning it will pay for the gas fees associated with the resulting transaction.  
Bundler  
A special node that monitors a mempool of user operations and bundles multiple user operations into a single transaction.  
INFO  
To learn more about Account Abstraction and the concepts outlined above, see ERC-4337.  
Creating and deploying a smart contract  
Before you begin, you need to set up a smart contract development environment using Foundry.  
To create a new project, first create a new directory named myproject, and change it to your current working directory:  
mkdir myproject  
cd myproject  
Creating a Foundry project  
Next, within the myproject directory create a new directory named contracts, and change it to your current working directory:  
mkdir contracts  
cd contracts  
Then create a new Foundry project by running the following command:  
forge init  
This will create a Foundry project with the following basic layout:  
.  
├── foundry.toml  
├── script  
├── src  
└── test  
INFO  
The command creates a boilerplate Solidity smart contract file named src/Counter.sol. This is the primary contract you will use for this tutorial.  
Compiling the smart contract  
Compile the smart contract to ensure it builds without any errors.  
To compile your smart contract, run:  
forge build  
Setting up a wallet as the deployer  
Before you can deploy your smart contract to various chains you will need to set up a wallet to be used as the deployer.  
To do so, you can use the cast wallet import command to import the private key of the wallet into Foundry's securely encrypted keystore:  
cast wallet import deployer --interactive  
After running the command above, you will be prompted to enter your private key, as well as a password for signing transactions.  
CAUTION  
For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation. It is critical that you do NOT commit this to a public repo.  
To confirm that the wallet was imported as the deployer account in your Foundry project, run:  
cast wallet list  
Deploying the smart contract  
To deploy the smart contract, you can use the forge create command. The command requires you to specify the smart contract you want to deploy, an RPC URL of the network you want to deploy to, and the account you want to deploy with.  
INFO  
Your wallet must be funded with ETH on the Base Goerli testnet to cover the gas fees associated with the smart contract deployment. Otherwise, the deployment will fail.  
To get testnet ETH, see the prerequisites.  
To deploy the smart contract to the Base Goerli testnet, run the following command:  
forge create ./src/Counter.sol:Counter --rpc-url https://goerli.base.org --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet’s private key.  
After running the command above, the contract will be deployed on the Base Goerli test network. You can view the deployment status and contract by using a block explorer.  
Setting up the Paymaster and Bundler  
To setup the paymaster and bundler for your project, you will need to visit the Biconomy Dashboard and complete the following steps.  
Registering a paymaster  
Add and register a Paymaster by completing the following steps:  
Visit the sign in to the Biconomy Dashboard  
From the dashboard, select the Paymasters tab and click Add your first Paymaster  
Provide a Name for your paymaster  
Select Base Goerli from the Network dropdown  
Click Register  
You should now have a registered Biconomy paymaster.  
INFO  
The API Key and Paymaster URL for the paymaster are provided under the Overview tab in the Biconomy Dashboard.  
Setting up the paymaster gas tank  
Set up and fund the paymaster's gas tank by completing the following steps:  
From the dashboard, navigate to the Paymasters tab  
Click Setup gas tank on the paymaster  
Navigate to Gas-Tank > Deposit, and click Set up gas tank  
Sign the message with your connected wallet to set up the gas tank  
Click Go to deposit  
Enter the amount of ETH you wish to deposit  
Click Deposit  
ETH should now be deposited into the gas tank for your paymaster. You can visit the Withdraw tab at a later time if you wish to withdraw the funds.  
Setting up the paymaster policies  
Set up and fund the paymaster's gas tank by completing the following steps:  
From the dashboard, navigate to the Paymasters tab  
Select the paymaster to configure  
Navigate to Policies > Contracts, and click Add your first contract  
Add the Name and the Smart contract address for your contract  
Select the increment and setNumber write methods as methods to sponsor  
Click Add Smart Contract  
Setting up a bundler  
Visit the sign in to the Biconomy Dashboard  
From the dashboard, select the Bundlers tab  
INFO  
At the time of writing this tutorial, the Bundler service is still under development, however a Bundler URL is provided for testing out UserOperations on test networks. You can specify the chain ID 84531 to use the Bundler URL on Base Goerli testnet.  
Setting up the frontend  
Creating a Next.js project  
After you set up your paymaster and bundler from the Biconomy Dashboard, the next step is to create a Next.js project for your app's frontend.  
From the root of the myproject directory of your project, create a new Next.js project by running the following command:  
yarn create next-app  
cd my-app  
Installing the dependencies  
To use the paymaster and bundler that were setup from the Biconomy Dashboard, you will need to add a few dependencies to your Next.js project.  
To install Biconomy as a dependency to your project, run the following command:  
yarn add @biconomy/account @biconomy/bundler @biconomy/common @biconomy/core-types @biconomy/paymaster ethers@5.7.2  
Creating Biconomy smart accounts requires a signer from an EIP-1193 provider. Biconomy works with a variety of different social login and embedded wallet onboarding solutions that provide access to a signer that can be used for creating smart accounts. In this tutorial, you will use Particle Network for user authentication and getting a smart account signer.  
To install Particle Network as a dependency to your project, run the following command:  
yarn add @biconomy/particle-auth  
Updating the boilerplate code  
The main page (page.tsx) of the Next.js project created when running the yarn create next-app command contains a Home component. This component comes with a lot of code that is unnecessary for this tutorial.  
Replace the content of the page.tsx file with the following simplified code:  
'use client';  
  
import styles from './page.module.css';  
  
export default function Home() {  
 return (  
 <main className={styles.main}>  
 <div></div>  
 </main>  
 );  
}  
Adding social login using Particle Network  
Setting up Particle Network  
To get started adding social login into the app using Particle Network, import and initialize the Biconomy Particle Auth module in the page.tsx file as shown below:  
'use client';  
  
import styles from './page.module.css';  
import { ParticleAuthModule, ParticleProvider } from '@biconomy/particle-auth';  
  
const PARTICLE\_PROJECT\_ID = 'YOUR\_PARTICLE\_PROJECT\_ID';  
const PARTICLE\_CLIENT\_ID = 'YOUR\_PARTICLE\_CLIENT\_ID';  
const PARTICLE\_APP\_ID = 'YOUR\_PARTICLE\_APP\_ID';  
  
const particle = new ParticleAuthModule.ParticleNetwork({  
 projectId: PARTICLE\_PROJECT\_ID,  
 clientKey: PARTICLE\_CLIENT\_ID,  
 appId: PARTICLE\_APP\_ID,  
 wallet: {  
 displayWalletEntry: true,  
 },  
});  
  
export default function Home() {  
 return (  
 <main className={styles.main}>  
 <div></div>  
 </main>  
 );  
}  
INFO  
You will need to sign up for a Particle Network account and replace the values of PARTICLE\_PROJECT\_ID, PARTICLE\_CLIENT\_ID, and PARTICLE\_APP\_ID with your own project ID, client ID, and app ID respectively. You can find this information on the Particle Network Dashboard.  
Adding login functionality  
Next, add a Login button and login function that triggers the Particle Network login flow and gets a Web3Provider:  
'use client';  
  
import styles from './page.module.css';  
import { ParticleAuthModule, ParticleProvider } from '@biconomy/particle-auth';  
import { ethers } from 'ethers';  
  
const PARTICLE\_PROJECT\_ID = 'YOUR\_PARTICLE\_PROJECT\_ID';  
const PARTICLE\_CLIENT\_ID = 'YOUR\_PARTICLE\_CLIENT\_ID';  
const PARTICLE\_APP\_ID = 'YOUR\_PARTICLE\_APP\_ID';  
  
const particle = new ParticleAuthModule.ParticleNetwork({  
 projectId: PARTICLE\_PROJECT\_ID,  
 clientKey: PARTICLE\_CLIENT\_ID,  
 appId: PARTICLE\_APP\_ID,  
 wallet: {  
 displayWalletEntry: true,  
 },  
});  
  
export default function Home() {  
 const login = async () => {  
 try {  
 const userInfo = await particle.auth.login();  
 const particleProvider = new ParticleProvider(particle.auth);  
 const web3Provider = new ethers.providers.Web3Provider(particleProvider, 'any');  
 } catch (error) {  
 console.error(error);  
 }  
 };  
  
 return (  
 <main className={styles.main}>  
 <div>  
 <button onClick={login}>Login</button>  
 </div>  
 </main>  
 );  
}  
Creating Smart Accounts using Biconomy  
Initializing the paymaster and bundler  
Before you can implement the rest of the login flow and create a smart account for the logged in user, you will need to specify a paymaster and bundler.  
To initialize the paymaster and bundler, add the following lines of code:  
'use client';  
  
import styles from './page.module.css';  
import { ParticleAuthModule, ParticleProvider } from '@biconomy/particle-auth';  
import { ethers } from 'ethers';  
  
import { IBundler, Bundler } from '@biconomy/bundler';  
import { IPaymaster, BiconomyPaymaster } from '@biconomy/paymaster';  
import { ChainId } from '@biconomy/core-types';  
import { DEFAULT\_ENTRYPOINT\_ADDRESS } from '@biconomy/account';  
  
const PARTICLE\_PROJECT\_ID = 'YOUR\_PARTICLE\_PROJECT\_ID';  
const PARTICLE\_CLIENT\_ID = 'YOUR\_PARTICLE\_CLIENT\_ID';  
const PARTICLE\_APP\_ID = 'YOUR\_PARTICLE\_APP\_ID';  
  
const PAYMASTER\_URL = 'YOUR\_PAYMASTER\_URL';  
const BUNDLER\_URL = 'YOUR\_BUNDLER\_URL';  
  
const particle = new ParticleAuthModule.ParticleNetwork({  
 projectId: PARTICLE\_PROJECT\_ID,  
 clientKey: PARTICLE\_CLIENT\_ID,  
 appId: PARTICLE\_APP\_ID,  
 wallet: {  
 displayWalletEntry: true,  
 },  
});  
  
const paymaster: IPaymaster = new BiconomyPaymaster({  
 paymasterUrl: PAYMASTER\_URL,  
});  
  
const bundler: IBundler = new Bundler({  
 chainId: ChainId.BASE\_GOERLI\_TESTNET,  
 entryPointAddress: DEFAULT\_ENTRYPOINT\_ADDRESS,  
 bundlerUrl: BUNDLER\_URL,  
});  
  
export default function Home() {  
 const login = async () => {  
 try {  
 const userInfo = await particle.auth.login();  
 const particleProvider = new ParticleProvider(particle.auth);  
 const web3Provider = new ethers.providers.Web3Provider(particleProvider, 'any');  
 } catch (error) {  
 console.error(error);  
 }  
 };  
  
 return (  
 <main className={styles.main}>  
 <div>  
 <button onClick={login}>Login</button>  
 </div>  
 </main>  
 );  
}  
INFO  
Replace the values of PAYMASTER\_URL, BUNDLER\_URL with the URLs for your paymaster and bundler respectively. You can find this information on the Biconomy Dashboard  
Creating the smart account  
Once the paymaster and bundler instances have been created, you’re ready to create a smart account for the user that is logging in.  
You can use the BiconomySmartAccountV2.create function to create a new smart account for the user.  
To create a smart account for the user on Base Goerli testnet that uses the Biconomy paymaster and bundler add the following code:  
...  
import {  
 BiconomySmartAccountV2,  
 DEFAULT\_ENTRYPOINT\_ADDRESS,  
} from '@biconomy/account';  
 import {  
 ECDSAOwnershipValidationModule,  
 DEFAULT\_ECDSA\_OWNERSHIP\_MODULE,  
} from '@biconomy/modules';  
  
...  
  
export default function Home() {  
  
...  
  
 const login = async () => {  
 try {  
 const userInfo = await particle.auth.login();  
 const particleProvider = new ParticleProvider(particle.auth);  
 const web3Provider = new ethers.providers.Web3Provider(  
 particleProvider,  
 'any',  
 );  
  
 const validationModule = await ECDSAOwnershipValidationModule.create({  
 signer: web3Provider.getSigner(),  
 moduleAddress: DEFAULT\_ECDSA\_OWNERSHIP\_MODULE,  
 });  
  
 let biconomySmartAccount = await BiconomySmartAccountV2.create({  
 chainId: ChainId.BASE\_GOERLI\_TESTNET,  
 bundler: bundler,  
 paymaster: paymaster,  
 entryPointAddress: DEFAULT\_ENTRYPOINT\_ADDRESS,  
 defaultValidationModule: validationModule,  
 activeValidationModule: validationModule,  
 });  
  
 const accountAddress = await biconomySmartAccount.getAccountAddress();  
 } catch (error) {  
 console.error(error);  
 }  
 };  
  
  
 return (  
 <main className={styles.main}>  
 <div>  
 <button onClick={login}>Login</button>  
 </div>  
 </main>  
 );  
}  
Saving the provider and smart account to state  
Later in this tutorial, you will use the provider and user’s smartAccount to execute transactions on the deployed smart contract. Store the provider and smartAccount to React state so you can use it later.  
To store the provider and smartAccount, add the following code:  
...  
import { useState } from 'react';  
  
...  
  
export default function Home() {  
  
 const [loading, setLoading] = useState(false);  
 const [provider, setProvider] = useState(null);  
 const [smartAccount, setSmartAccount] = useState(null);  
 const [address, setAddress] = useState('');  
...  
  
 const login = async () => {  
 try {  
 setLoading(true);  
 const userInfo = await particle.auth.login();  
 const particleProvider = new ParticleProvider(particle.auth);  
 const web3Provider = new ethers.providers.Web3Provider(  
 particleProvider,  
 'any',  
 );  
  
 const validationModule = await ECDSAOwnershipValidationModule.create({  
 signer: web3Provider.getSigner(),  
 moduleAddress: DEFAULT\_ECDSA\_OWNERSHIP\_MODULE,  
 });  
  
  
 let biconomySmartAccount = await BiconomySmartAccountV2.create({  
 chainId: ChainId.BASE\_GOERLI\_TESTNET,  
 bundler: bundler,  
 paymaster: paymaster,  
 entryPointAddress: DEFAULT\_ENTRYPOINT\_ADDRESS,  
 defaultValidationModule: validationModule,  
 activeValidationModule: validationModule,  
 });  
  
 const accountAddress = await biconomySmartAccount.getAccountAddress();  
 setProvider(web3Provider);  
 setSmartAccount(biconomySmartAccount);  
 setAddress(accountAddress);  
 setLoading(false);  
 } catch (error) {  
 console.error(error);  
 }  
 };  
  
 return (  
 <main className={styles.main}>  
 <div>  
 {!loading && !address && <button onClick={login}>Login</button>}  
 {loading && <p>Loading...</p>}  
 {address && <h2>Smart Account Address: {address}</h2>}  
 </div>  
 </main>  
 );  
}  
INFO  
The code above is also updated to hide and display the login button and smart account address of the user, depending on if the user is logged in or not.  
Executing a gasless transaction  
Now that the app is able to create smart accounts for each logged in user, lets provide the ability for a user to interact with the deployed smart contract.  
Creating a Counter component  
To allow users to interact with the deployed Counter smart contract, create a new directory named src/components and create a new file named Counter.tsx with the following content:  
'use client';  
  
import React, { useState, useEffect } from 'react';  
import { ethers } from 'ethers';  
import { PaymasterMode } from '@biconomy/paymaster';  
import abi from '../utils/abi.json';  
  
const CONTRACT\_ADDRESS = 'YOUR\_CONTRACT\_ADDRESS';  
  
export default function Counter({ smartAccount, provider }) {  
 const [number, setNumber] = useState(0);  
 const [contract, setContract] = useState(null);  
  
 useEffect(() => {  
 const counterContract = new ethers.Contract(CONTRACT\_ADDRESS, abi, provider);  
 setContract(counterContract);  
 }, []);  
  
 const getNumber = async () => {  
 const currentNumber = await contract.number();  
 setNumber(currentNumber.toNumber());  
 };  
  
 const increment = async () => {  
 const incrementTx = new ethers.utils.Interface(['function increment()']);  
 const data = incrementTx.encodeFunctionData('increment');  
  
 const transaction = {  
 to: CONTRACT\_ADDRESS,  
 data: data,  
 };  
  
 try {  
 const userOp = await smartAccount.buildUserOp([transaction], {  
 paymasterServiceData: {  
 mode: PaymasterMode.SPONSORED,  
 },  
 });  
 const userOpResponse = await smartAccount.sendUserOp(userOp);  
 const transactionDetails = await userOpResponse.wait();  
 console.log('Transaction details:', transactionDetails);  
 console.log('Transaction hash:', transactionDetails.receipt.transactionHash);  
 } catch (e) {  
 console.error('Error executing transaction:', e);  
 }  
 };  
  
 return (  
 <>  
 <div>Current number: {number}</div>  
 <button onClick={() => increment()}>Increment</button>  
 </>  
 );  
}  
INFO  
Replace the value of CONTRACT\_ADDRESS with the address for your deployed Counter.sol contract.  
Code explanation  
The code above is a simple React component that interacts with the deployed Counter smart contract.  
A contract instance is initialized and stored in React state when the component first renders  
useEffect(() => {  
 const counterContract = new ethers.Contract(contractAddress, abi, provider);  
 setContract(counterContract);  
}, []);  
The component also has two functions called getNumber and increment. getNumber reads the number member variable of the smart contract, and increment makes a call to the increment function of the smart contract:  
const getNumber = async () => {  
 const currentNumber = await contract.number();  
 setNumber(currentNumber.toNumber());  
};  
  
const increment = async () => {  
 const incrementTx = new ethers.utils.Interface(['function increment()']);  
 const data = incrementTx.encodeFunctionData('increment');  
  
 const transaction = {  
 to: contractAddress,  
 data: data,  
 };  
  
 try {  
 const userOp = await smartAccount.buildUserOp([transaction], {  
 paymasterServiceData: {  
 mode: PaymasterMode.SPONSORED,  
 },  
 });  
 const userOpResponse = await smartAccount.sendUserOp(userOp);  
 const transactionDetails = await userOpResponse.wait();  
 } catch (e) {  
 console.error('Error executing transaction:', e);  
 }  
};  
Adding the contract ABI  
Initializing a contract instance requires the contract's Application Binary Interface (ABI) to be provided. The code for the Counter component in the previous section imports a file called abi.json:  
import abi from '../utils/abi.json';  
This file does not exist yet, so you will need to add it.  
To add the ABI, create a new directory named src/utils and create a new file named abi.json with the following content:  
[  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "newNumber",  
 "type": "uint256"  
 }  
 ],  
 "name": "setNumber",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [],  
 "name": "increment",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [],  
 "name": "number",  
 "outputs": [  
 {  
 "internalType": "uint256",  
 "name": "",  
 "type": "uint256"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 }  
]  
INFO  
If your deployed contract is verified, you can also get the ABI for the contract from BaseScan.  
Updating the Home component  
Now that the Counter component has been created, add it to the Home component, and pass it the provider and user's smartAccount:  
...  
import Counter from '@/component/Counter';  
  
export default function Home() {  
  
...  
  
 return (  
 <main>  
 <div>  
 {!loading && !address && <button onClick={login}>Login</button>}  
 {loading && <p>Loading...</p>}  
 {address && <h2>Smart Account Address: {address}</h2>}  
 <Counter smartAccount={smartAccount} provider={provider}/>  
 </div>  
 </main>  
 );  
}  
Executing the transaction  
With all of the components set up, you are ready to run and test the app by logging in and executing a gasless transaction.  
Perform the following steps:  
Run the application by running yarn dev  
Click the Login button to login and create a smart account  
Once logged in, click the Increment button to execute the increment() function of the smart contract  
Upon a successful transaction, observe the number update.  
Conclusion  
Congratulations! You have successfully learned how to implement Account Abstraction in your Base projects using Biconomy.  
To learn more about Account Abstraction and Biconomy, check out the following resources:  
Biconomy Documentation  
Account Abstraction on Base  
Tags:account abstraction  
ON THIS PAGE  
Prerequisites  
What is Biconomy?  
High-level concepts  
Coinbase Wallet  
Wallet funds  
Creating and deploying a smart contract  
Creating a Foundry project  
Setting up the Paymaster and Bundler  
Registering a paymaster  
Setting up the paymaster gas tank  
Setting up the paymaster policies  
Setting up a bundler  
Setting up the frontend  
Creating a Next.js project  
Installing the dependencies  
Updating the boilerplate code  
Setting up a bundler  
Adding social login using Particle Network  
Setting up Particle Network  
Adding login functionality  
Updating the boilerplate code  
Creating Smart Accounts using Biconomy  
Initializing the paymaster and bundler  
Creating the smart account  
Executing a gasless transaction  
Creating a Counter component  
Adding the contract ABI  
Updating the Home component  
Conclusion  
Creating a Counter component  
Adding the contract ABI  
Updating the Home component  
Executing the transaction  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/account-abstraction-with-particle-network  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Account Abstraction on Base using Particle Network  
Particle Network is a Smart Wallet-as-a-Service provider on Base, providing a modular Account Abstraction stack, allowing developers to use a variety of Paymasters, Bundlers, or smart accounts along with social logins.  
This document will guide you through the process of using Particle Network within your Base application, building a simple React project using create-react-app, Particle Auth Core, and Particle's AA SDK.  
Objectives  
By the end of this guide, you should be able to:  
Use Particle Auth Core to generate an Externally-Owned-Account (EOA) via a social login  
Assign a chosen smart account to the EOA generated by Particle Auth Core  
Set up a Bundler and Paymaster  
Construct and execute a gasless transaction  
Prerequisites  
Wallet funds  
This guide requires you to have ETH on Base Sepolia, which will be used to showcase the execution of a gasless burn transaction.  
To fund your wallet with ETH on Base Sepolia, visit one of the faucets listed on the Base Faucets page.  
Familiarity with modern, frontend web development  
In this example, you'll be building a React-based application using create-react-app. It's recommended that you have some level of familiarity with the basics of working with React.  
Understanding Particle Network  
Wallet-as-a-Service  
Particle Network provides a large suite of SDKs centered around the reduction of account-based friction.  
In this case, "account-based friction" refers to barriers-to-entry that some Web3 users may face as they onboard into an application and begin managing a wallet.  
This friction, in the context of this guide, can be placed within two distinct categories:  
The login process. Often, decentralized applications that tend to be more consumer-facing prefer login flows that aren't dependent upon a user downloading and managing a traditional wallet, as this can be a pain point for some.  
The rigidity of standard accounts. Externally Owned Accounts, or EOAs, are often quite rigid in how they operate. They're secured by one key and limited to a strict range of functions, thus developers (and therefore users) are confined to relatively low-level interaction with applications.  
Wallet-as-a-Service (WaaS) aims to solve the first of these two points, the login process. WaaS solutions provide an alternative to standard wallets, typically allowing users to use applications through accounts generated by social logins (such as Google, email, or phone). The interfaces for interacting with these accounts are also often embedded within the applications, resulting in a consistent, application-specific experience.  
WaaS providers other than Particle Network include Web3Auth, Privy, and Magic, among others.  
Account Abstraction  
Particle Network also aims to tackle the second friction point described above: account flexibility.  
Account Abstraction refers to a transition away from standard account structures, EOAs, to smart accounts. Smart accounts are contracts that act as a wallet, providing users with an account that feels equivalent to an EOA but is intrinsically programmable (due to it being a smart contract) and thus more flexible.  
The most popular modern implementation of Account Abstraction is ERC-4337, which enables Account Abstraction without any consensus-layer protocol changes. It does this through numerous components of supporting infrastructure, including a Bundler and Paymaster.  
Particle Network describes its Account Abstraction stack as modular, referring to cross-compatibility with any provider of Bundlers, Paymasters, or smart accounts). Particle Network's Account Abstraction SDK runs and uses its own Bundler and Paymaster, with built-in support for Biconomy's Paymaster. However, Particle has made it simple to plug into external infrastructure and components, such as Paymasters or Bundlers from providers like Stackup or Pimlico.  
Wallet-as-a-Service + Account Abstraction  
Leveraging Account Abstraction directly with Wallet-as-a-Service allows users to onboard through social logins into embedded wallets that use smart accounts, not EOAs, allowing for a greater degree of flexibility.  
Particle Network does this by allowing developers to use its Account Abstraction SDK alongside its Wallet-as-a-Service SDK (Particle Auth) to facilitate the intersection between both technologies, as we'll cover in this guide.  
INFO  
To learn more about Account Abstraction and the concepts outlined above, see ERC-4337, or Base’s Introduction to Account Abstraction guide.  
Setting up the frontend  
This guide will go through the process of creating a React-based application through the create-react-app template, as shown below.  
Creating a React project  
To begin, we'll need to initialize a standard create-react-app project.  
Within your IDE of choice, run the following command, replacing {name} with the name of your project.  
npx create-react-app {name}  
This will create a React project within a directory under the name you set within {name}. This directory should contain the following structure by default:  
{name}/  
├── node\_modules/  
├── public/  
├── src/  
├── .gitignore  
├── package.json  
├── package-lock.json  
└── README.md  
Throughout this guide, we'll primarily be working with the src folder and the files within it. src will contain the following by default:  
src/  
├── App.css  
├── App.js  
├── App.test.js  
├── index.css  
├── index.js  
├── logo.svg  
└── reportWebVitals.js  
This structure won't function properly out of the box as our example will be using JSX, a syntax extension that requires a special file type. Thus, you'll need to change the file extension of your App.js and index.js files to either .jsx (JavaScript) or .tsx (TypeScript).  
We will use these two files, App.tsx and index.tsx, within this guide.  
Configuring & Initializing Particle Network  
Installing the dependencies  
Before jumping into the application itself, it's important to walk through a few vital configurations required for Particle Network's SDKs to function properly.  
Within this example, we'll be using three core libraries from Particle, these include:  
@particle-network/auth-core-modal, to directly initiate a social login and drive the usage of an embedded wallet.  
@particle-network/aa, for configuring, assigning, and deploying a smart account.  
@particle-network/chain, to allow Base to be used within this example.  
To add these as dependencies within your project, run one of the two following commands at the root of your project.  
yarn add @particle-network/auth-core-modal @particle-network/aa @particle-network/chains  
  
# OR  
  
npm install @particle-network/auth-core-modal @particle-network/aa @particle-network/chains  
In addition to the above libraries from @particle-network, we'll be using Ethers for core functions, such as retrieving the user's balance, sending a gasless transaction, and so on.  
For the sake of simplicity, we'll be using Ethers v5.7.2, the last release before the Ethers v6 upgrade. Although, if you'd like to use a newer version (v6), comments noting the new syntax of v6 will be left on the code snippets throughout this guide.  
Thus, you'll need to install Ethers through either of the following commands:  
yarn add ethers@5.7.2  
  
# OR  
  
npm install ethers@5.7.2  
Setting up the Particle Network dashboard  
As you'll find in a moment, every library from Particle Network requires three key values for authentication. These are:  
Your projectId, assigned to a project created through the Particle dashboard.  
Your clientKey, similarly assigned to a project created through the dashboard, but serving a different purpose.  
Your appId, retrieved through the creation of an application (within a project) on the dashboard.  
Configuring both @particle-network/auth-core-modal and @particle-network/aa will require the retrieval and utilization of these three values.  
To create a project and an application through the Particle dashboard:  
Navigate to https://dashboard.particle.network.  
Create a new account (with your email).  
Click "Add New Project" and enter the name of your project.  
Copy and save the "Project ID" and "Client Key."  
Create a new application. For this example, we’ll select "Web."  
Once again, enter the name of your project, alongside the domain where you intend to host this application. If you don't have a domain in mind, feel free to use any filler domain (such as 'base.org'), as it won't affect any underlying functionalities.  
Copy the "App ID" shown after creating an application.  
With these values retrieved, it's recommended that you assign them to environment variables within your application, such as REACT\_APP\_PROJECT\_ID, REACT\_APP\_CLIENT\_KEY, and REACT\_APP\_APP\_ID.  
Configuring Particle Auth Core  
Now that you've installed @particle-network/auth-core-modal (among other dependencies) and retrieved your project ID, client key, and app ID, you're ready to configure and therefore initialize the Particle Auth Core SDK.  
As mentioned, we'll be working out of two files within this guide:  
App.jsx/tsx, containing our core application logic (such as initiating social login and executing the transaction).  
index.jsx/tsx, used in this example for configuring AuthCoreContextProvider from @particle-network/auth-core-modal, the core configuration object for Particle Auth Core.  
AuthCoreContextProvider is a React component used to define these three aforementioned values, customize the embedded wallet modal and enable account abstraction within it. This will wrap our primary application component (App from App.jsx/tsx), therefore allowing Particle Auth Core to be used through various hooks within App.  
To achieve this, AuthCoreContextProvider will take the following parameters:  
projectId, clientKey, and appId. These are the required values previously retrieved from the Particle dashboard.  
erc4337, used to define the type of smart account you intend to use, ensuring it's displayed and reflected within the embedded wallet interface.  
wallet, for customizing the embedded wallet interface through the restriction of supported chains, color options, etc.  
With this in mind, an example of what your index.jsx/tsx file may look like given the usage of AuthCoreContextProvider has been included below.  
import React from 'react';  
import ReactDOM from 'react-dom/client';  
  
import { AuthType } from '@particle-network/auth-core';  
import { BaseSepolia } from '@particle-network/chains';  
import { AuthCoreContextProvider } from '@particle-network/auth-core-modal';  
  
import App from './App';  
  
// Optional, needed for some environments  
import('buffer').then(({ Buffer }) => {  
 window.Buffer = Buffer;  
});  
// -----  
  
ReactDOM.createRoot(document.getElementById('root') as HTMLElement).render(  
 <React.StrictMode>  
 <AuthCoreContextProvider  
 options={{  
 projectId: process.env.REACT\_APP\_PROJECT\_ID,  
 clientKey: process.env.REACT\_APP\_CLIENT\_KEY,  
 appId: process.env.REACT\_APP\_APP\_ID,  
 erc4337: {  
 // The name of the smart account you'd like to use  
 // SIMPLE, BICONOMY, LIGHT, or CYBERCONNECT  
 name: 'SIMPLE',  
 // The version of the smart account you're using  
 // 1.0.0 for everything except Biconomy, which can be either 1.0.0 or 2.0.0  
 version: '1.0.0',  
 },  
 wallet: {  
 // Set to false to remove the embedded wallet modal  
 visible: true,  
 customStyle: {  
 // Locks the chain selector to Base Sepolia  
 supportChains: [BaseSepolia],  
 },  
 },  
 }}  
 >  
 <App />  
 </AuthCoreContextProvider>  
 </React.StrictMode>,  
);  
Setting Up Your Application  
Importing and configuring hooks  
The second core component of this application is App.jsx/tsx, which will contain logic achieving the following:  
Configuration and assignment of a smart account (SimpleAccount in this example).  
Construction of a custom Ethers provider, using a custom AA-enabled EIP-1193 provider object.  
Initiation of social login.  
Execution of a gasless transaction.  
To achieve this, we'll be using a combination of hooks (from @particle-network/auth-core-modal) and base functions on Ethers, which will be automatically powered by AA through the custom EIP-1193 provider object.  
These hooks include useEthereum for the retrieval of the standard EOA-based provider object, useConnect for managing social logins, and useAuthCore to retrieve the user’s information (after social login).  
To begin building App.jsx/tsx, you'll need to define the relevant functions from these hooks through a process similar to the example below:  
import React, { useState, useEffect } from 'react';  
import { useEthereum, useConnect, useAuthCore } from '@particle-network/auth-core-modal';  
import { BaseSepolia } from '@particle-network/chains';  
import { AAWrapProvider, SendTransactionMode, SmartAccount } from '@particle-network/aa';  
import { ethers } from 'ethers';  
  
import './App.css';  
  
const App = () => {  
 // Standard, EOA-based 1193 provider  
 const { provider } = useEthereum();  
 // Used for initiating social login and disconnecting users (post-login)  
 const { connect, disconnect } = useConnect();  
 // Automatically loaded with relevant user information after logging in  
 const { userInfo } = useAuthCore();  
};  
Configuring the Smart Account  
Choosing and defining a smart account  
The EOA generated through the social login process will be used as the Signer for the smart account specified within this configuration, this will then be reflected through the embedded wallet modal (through its former selection within AuthCoreContextProvider).  
As mentioned, Particle Network supports a variety of smart accounts directly through its AA SDK, these include:  
Light Account (by Alchemy).  
Biconomy V1 and V2.  
SimpleAccount (eth-infinitism).  
CyberConnect.  
We'll be using SimpleAccount in this case. Although, note that you can change which smart account you use at any time through one line of code. This line of code exists on the SmartAccount object (imported from @particle-network/aa). SmartAccount acts as the central point for initializing a smart account.  
To configure SmartAccount, you'll be using the following parameters:  
projectId, clientKey, and appId. These were used within AuthCoreContextProvider and can be retrieved from the Particle dashboard through the same procedure.  
aaOptions, which contains accountContracts. Within accountContracts, you'll need to define a property corresponding with the smart account you'd like to use, i.e. BICONOMY, LIGHT, CYBERCONNECT, or SIMPLE.  
This property contains chainIds and version.  
See the snippet below for an example of a constructed SmartAccount object:  
import React, { useState, useEffect } from 'react';  
import { useEthereum, useConnect, useAuthCore } from '@particle-network/auth-core-modal';  
import { BaseSepolia } from '@particle-network/chains';  
import { AAWrapProvider, SendTransactionMode, SmartAccount } from '@particle-network/aa';  
import { ethers } from 'ethers';  
  
import './App.css';  
  
const App = () => {  
 ...  
  
 const smartAccount = new SmartAccount(provider, {  
 projectId: process.env.REACT\_APP\_PROJECT\_ID,  
 clientKey: process.env.REACT\_APP\_CLIENT\_KEY,  
 appId: process.env.REACT\_APP\_APP\_ID,  
 aaOptions: {  
 accountContracts: {  
 SIMPLE: [{ chainIds: [BaseSepolia.id], version: '1.0.0' }]  
 // BICONOMY: [{ chainIds: [BaseSepolia.id], version: '1.0.0' }]  
 // BICONOMY: [{ chainIds: [BaseSepolia.id], version: '2.0.0' }]  
 // LIGHT: [{ chainIds: [BaseSepolia.id], version: '1.0.0' }]  
 // CYBERCONNECT: [{ chainIds: [BaseSepolia.id], version: '1.0.0' }]  
 }  
 }  
 });  
};  
Constructing a custom Ethers object  
There are two primary mechanisms to interact with the smart account defined previously. These are:  
Directly through the constructed SmartAccount object, such as with {your object}.sendTransaction.  
Through a Web3 library, such as Ethers or Web3.js. These libraries can facilitate interaction through a standardized provider object, allowing for a consistent syntax and setup through any wallet or account structure.  
As mentioned, we'll be using Ethers within this guide. You'll be able to use any typical Ethers method, such as sendTransaction or getBalance through the AA-enabled provider we construct with AAWrapProvider (imported from @particle-network/aa).  
Essentially, we'll construct an instance of AAWrapProvider, passing in the previously defined SmartAccount object alongside an object representing the method selected to pay gas fees. This will allow Ethers to directly load the smart account and drive transactions/signatures through Particle's embedded wallet.  
You can achieve this in one line of code, e.g.:  
// For Ethers V6, use ethers.BrowserProvider instead; the syntax below won't work.  
const customProvider = new ethers.providers.Web3Provider(  
 new AAWrapProvider(smartAccount, SendTransactionMode.Gasless),  
 'any',  
);  
SendTransactionMode has three options. They are:  
SendTransactionMode.Gasless, which will request gas sponsorship on every transaction sent through Ethers. By default, this will be through Particle's Omnichain Paymaster. If you don't have enough USDT deposited in the Paymaster to cover the gas fees, or if the transaction fails to meet your sponsorship conditions (set on the Particle dashboard), the user will pay the gas fees themselves.  
SendTransactionMode.UserPaidNative, the default method used if SendTransactionMode is missing from AAWrapProvider. This forces the user to pay gas fees themselves.  
SendTransactionMode.UserSelect, which allows a user to select which gas fee payment mechanism they use (ERC-20 token, native token, or request sponsorship).  
In this example, we're using SendTransactionMode.Gasless. Because this example uses Base Sepolia, all transactions will automatically be sponsored.  
Initiating Social Login  
Calling the connect function  
At this point, you've configured Particle Auth Core, initialized a smart account of choice, and constructed a custom Ethers provider to drive on-chain interaction.  
As covered earlier, one of the core benefits of Wallet-as-a-Service is the ability to onboard users through their social accounts. In this case, users’ social login will directly create or log them into a smart account (if they have already created one through Particle Auth), allowing for the immediate usage of account abstraction.  
To initiate the social login process programmatically, you'll need to use the connect function, defined from the useConnect hook imported earlier (from @particle-network/auth-core-modal).  
Upon calling connect, a user will be brought through the social login process, after which an EOA will be generated (through MPC-TSS) and used as a signer for the smart account.  
connect takes the following parameters:  
socialType, the specific social login mechanism you'd like users to go through. If left as an empty string, a generalized social login modal will be shown. Otherwise, use strings such as 'google', 'twitter', 'email', etc.  
chain, an object (imported from @particle-network/chains) corresponding with the chain you'd like to use. In this example, it'll be BaseSepolia.  
Ideally, some logic should be set in place to ensure connect isn't called if a user has already logged in. This can be done by only calling connect on the condition that userInfo (from useAuthCore) is undefined, indicating that the user isn't logged in.  
Below is an example of calling connect (within the conditional described above).  
const handleLogin = async (authType) => {  
 if (!userInfo) {  
 await connect({  
 socialType: authType,  
 chain: BaseSepolia,  
 });  
 }  
};  
In most applications, connect (or handleLogin in this example), will be bound to a "Login" or "Connect" button, as will be done here.  
Executing a Gasless Transaction  
Constructing a transaction  
Because we're using Ethers in this guide, constructing and executing a gasless transaction (intrinsically gasless through the previously defined SendTransactionMode) is identical to the flow you're likely already familiar with. However, it's important to note that transactions sent through ERC-4337 account abstraction do not follow standard transaction structures, these are called UserOperations.  
Typically, UserOperations follow lower-level, alternative structures. Although, through the usage of AAWrapProvider, the conversion between a simple transaction object (with to, value, data, etc.) to a complete UserOperation is handled automatically, allowing you to send transactions as you would normally.  
Thus, we'll be constructing a simple transaction (tx) adhering to the following structure:  
to, the recipient address. For this example, we can burn a small amount of ETH on Base Sepolia, which means the recipient will be 0x000000000000000000000000000000000000dEaD.  
value, the value being sent in wei. Because of the default denomination in wei, this will be set as ethers.utils.parseEther("0.001").  
If you intend on interacting with a contract, data can also be filled out (or within Ethers, a Contract object can be built).  
Therefore, your tx object should look like this:  
const executeUserOp = async () => {  
 ...  
  
 const tx = {  
 to: "0x000000000000000000000000000000000000dEaD",  
 value: ethers.utils.parseEther("0.001"),  
 };  
  
 ...  
};  
Executing a transaction  
Now that you've defined a standard transaction object, you'll need to execute it. Once again, due to the usage of Ethers, this is quite straightforward.  
We'll be using a signer object retrieved from {your provider}.getSigner() to call the sendTransaction method, which simply takes the tx object we constructed a moment ago.  
Upon calling signer.sendTransaction(tx), the user will be prompted to confirm the transaction (sign a UserOperation hash) through an application-embedded popup. After doing so, the transaction will immediately be sent on Base Sepolia.  
To reflect the transaction hash after its confirmation on-chain, you can call the wait method on the variable you saved signer.sendTransaction(tx) to. The resulting object will contain a transactionHash value.  
See the example below for a visualization of this process:  
const executeUserOp = async () => {  
 const signer = customProvider.getSigner();  
  
 const tx = {  
 to: '0x000000000000000000000000000000000000dEaD',  
 value: ethers.utils.parseEther('0.001'),  
 };  
  
 const txResponse = await signer.sendTransaction(tx);  
 const txReceipt = await txResponse.wait();  
  
 return txReceipt.transactionHash;  
};  
Mapping to JSX  
You've now initiated social login (through handleLogin), assigned a smart account (through SmartAccount), and executed a gasless transaction (through executeUserOp).  
To present all of this to the user and allow them to interact with these functions for themselves, you'll need to map handleLogin and executeUserOp to the JSX of your App component. This will format the frontend that a user interacts with to test this application.  
Essentially, this displays either "Sign in with Google" or "Sign in with Twitter" through custom buttons that are only shown if the user hasn't logged in (determined through the state of userInfo). Upon logging in, the user can either call executeUserOp or disconnect (which was defined from useConnect).  
Below is an example of what your App.jsx/tsx file may look at this point. At the bottom of this snippet you'll find the JSX:  
import React, { useState, useEffect } from 'react';  
  
import { useEthereum, useConnect, useAuthCore } from '@particle-network/auth-core-modal';  
import { BaseSepolia } from '@particle-network/chains';  
import { AAWrapProvider, SmartAccount, SendTransactionMode } from '@particle-network/aa';  
  
import { ethers } from 'ethers';  
  
import './App.css';  
  
const App = () => {  
 const { provider } = useEthereum();  
 const { connect, disconnect } = useConnect();  
 const { userInfo } = useAuthCore();  
  
 // Initializes and assigns a smart account to the EOA resulting from social login  
 const smartAccount = new SmartAccount(provider, {  
 projectId: process.env.REACT\_APP\_PROJECT\_ID,  
 clientKey: process.env.REACT\_APP\_CLIENT\_KEY,  
 appId: process.env.REACT\_APP\_APP\_ID,  
 aaOptions: {  
 accountContracts: {  
 SIMPLE: [{ chainIds: [BaseSepolia.id], version: '1.0.0' }],  
 },  
 },  
 });  
  
 // Enables interaction with the smart account through Ethers  
 const customProvider = new ethers.providers.Web3Provider(  
 new AAWrapProvider(smartAccount, SendTransactionMode.Gasless),  
 'any',  
 );  
  
 // Initiates social login according to authType  
 const handleLogin = async (authType) => {  
 if (!userInfo) {  
 await connect({  
 socialType: authType,  
 chain: BaseSepolia,  
 });  
 }  
 };  
  
 // Executes a gasless burn of 0.001 ETH  
 const executeUserOp = async () => {  
 const signer = customProvider.getSigner();  
  
 const tx = {  
 to: '0x000000000000000000000000000000000000dEaD',  
 value: ethers.utils.parseEther('0.001'),  
 };  
  
 const txResponse = await signer.sendTransaction(tx);  
 const txReceipt = await txResponse.wait();  
  
 return txReceipt.transactionHash;  
 };  
  
 // The JSX  
 return (  
 <div className="App">  
 <div className="logo-section">  
 <img src="https://i.imgur.com/EerK7MS.png" alt="Logo 1" className="logo logo-big" />  
 <img src="https://i.imgur.com/1RV3pMV.png" alt="Logo 2" className="logo" />  
 </div>  
 {!userInfo ? (  
 <div className="login-section">  
 <button className="sign-button" onClick={() => handleLogin('google')}>  
 Sign in with Google  
 </button>  
 <button className="sign-button" onClick={() => handleLogin('twitter')}>  
 Sign in with Twitter  
 </button>  
 </div>  
 ) : (  
 <div className="profile-card">  
 <h2>{userInfo.name}</h2>  
 <div className="button-section">  
 <button className="sign-message-button" onClick={executeUserOp}>  
 Execute User Operation  
 </button>  
 <button className="disconnect-button" onClick={() => disconnect()}>  
 Logout  
 </button>  
 </div>  
 </div>  
 )}  
 </div>  
 );  
};  
  
export default App;  
With everything complete, you're ready to run your application to test it. To do so, go through the following process:  
Navigate to the root of your project.  
Run either npm run start or yarn start.  
Once running, log in with either your Google or Twitter.  
Fund the address displayed on the wallet modal in the bottom right.  
Click "Execute User Operation".  
Conclusion  
Congratulations! You've just built an application from scratch, onboarding users into smart accounts through social logins using Particle Network.  
To learn more about using Particle Network on Base, take a look at the following resources:  
Biconomy Guide (which uses Particle Network)  
Account Abstraction on Base  
Particle Network Documentation  
Particle Network 101: Developer Experience  
Tags:account abstraction  
ON THIS PAGE  
Prerequisites  
Understanding Particle Network  
Wallet-as-a-Service  
Account Abstraction  
Setting up the frontend  
Creating a React project  
Account Abstraction  
Configuring & Initializing Particle Network  
Installing the dependencies  
Setting Up Your Application  
Importing and configuring hooks  
Setting up the Particle Network dashboard  
Configuring Particle Auth Core  
Configuring the Smart Account  
Choosing and defining a smart account  
Initiating Social Login  
Calling the connect function  
Constructing a custom Ethers object  
Executing a Gasless Transaction  
Constructing a transaction  
Mapping to JSX  
Constructing a transaction  
Executing a transaction  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/account-abstraction-with-privy-and-base-paymaster  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Account Abstraction on Base using Privy and the Base Paymaster  
This tutorial shows you how to use Privy, Alchemy's Account Kit, and the Base Paymaster to enable your users to use onchain apps without creating a wallet on their own, or even needing to pay for gas fees!  
Objectives  
By the end of this tutorial, you should be able to:  
Intro to Account Abstraction  
Explain how Account Abstraction can improve user experience for onchain apps  
Describe the difference between contract accounts and EOAs, the limitations of contract accounts, and how EIP-4337 uses UserOperations to mitigate these limitations  
Outline how Account Abstraction works, and how users interact with smart contract wallets  
Intro to Privy  
Implement Privy's quick start to add onchain authentication to a NextJS application  
Compare Privy's progressive authentication strategy with traditional wallet-based authentication  
Use Privy's PrivyProvider context and usePrivy hook to implement basic authentication via an email address, SMS, EOA, and/or social auth  
Utilize Privy's Embedded Wallets to enable users to utilize wallet-based actions without having to connect to an external wallet or leave your application  
Implementing the Paymaster  
Describe how a third party can use a paymaster to sponsor gas  
Modify Privy's Base paymaster example example to work in another app, using an EOA to allow a user to call a smart contract function without requiring the user to pay any gas  
Prerequisites  
1. Be familiar with modern, frontend web development  
In this tutorial, we'll be working with a React frontend built with Next.js. While you don't need to be an expert, we'll assume that you're comfortable with the basics.  
2. Possess a general understanding of the EVM and smart contract development  
This tutorial assumes that you're reasonably comfortable writing basic smart contracts. If you're just getting started, jump over to our Base Learn guides and start learning!  
Intro do Account Abstraction  
ERC-4337, also known as Account Abstraction, is a standard that allows smart contracts to initiate transactions, thus enabling any logic that users want to implement to be encoded into the smart contract wallet itself for execution on Ethereum.  
Account Abstraction has the potential to be a massive game-changer for onchain user experience and many believe it will play a key role in bringing the next billion users onchain.  
The Problem Account Abstraction Solves  
Onchain applications are difficult to use for many people, as they require a lengthy onboarding process for the average internet user who already onchain.  
For example,the preliminary steps a user needs to go through before they can use an onchain app are:  
Create a wallet (Coinbase Wallet, Metamask, etc)  
Store the wallet mnemonic safely, without losing it or compromising it  
Sign a slightly frightening message to connect to an onchain app  
Try to do anything with the app and get a popup asking them to approve a transaction  
Attempt to do so, learn transactions cost gas, and that they don't have any  
Learn that gas is a fee that users must pay in ETH to use onchain apps  
Attempt to buy ETH, possibly buying the wrong flavor of ETH in the wrong location  
Try the transaction again  
Repeat until they finally find the correct path  
For widespread adoption of onchain applications, this confusing and alienating process has to change.  
That's where Account Abstraction comes in. It allows you to improve the onboarding and usage flow for your users:  
User goes to the onchain app and authenticates (using email, EOA, or social auth)  
User uses the app  
Transactions happen under the hood via a smart contract wallet and the app developer sponsors the user's gas fees until after they're onboarded.  
To summarize, Account Abstraction enables smart contract accounts to initiate user operations, similar to how an EOA would initiate a transaction. However, unlike EOAs, smart contract accounts are programmable and can enable a number of incredible features, such as:  
Sponsored Transactions: Allow application owners to cover the users gas fees with a paymaster or allow a user to use something other than ETH (USDC, for example) to cover gas  
Arbitrary Verification Logic: Verify transactions with custom logic  
Account Recovery: Create account recovery features for when user lose private keys  
Batching Transactions: Change the user experience so that multiple transactions can be submitted at once  
Overview of ERC-4337 - Account Abstraction  
"Account Abstraction" comes from ERC-4337. The proposal itself is a dense read, but one of stated goals of the proposal is to:  
allow users to use smart contract wallets containing arbitrary verification logic instead of EOAs as their primary account. Completely remove any need at all for users to also have EOAs (as status quo SC wallets and EIP-3074 both require)  
In other words, the proposal seeks to allow users to use smart contract wallets instead of EOAs to transact onchain.  
INFO  
You're working with something so new that the vocabulary hasn't settled yet. You'll often see the terms "smart contract wallet", "smart contract account" or just "smart account" used interchangeably.  
At first glance, you may be asking yourself if using smart contract accounts solve our problem. What's the difference between a smart contract account (or contract account) and an Externally Owned Account?  
According to ethereum.org, the differences are:  
Externally Owned Account  
Creating an account costs nothing  
Can initiate transactions  
Transactions between externally-owned accounts can only be ETH/token transfers  
Made up of a cryptographic pair of keys: public and private keys that control account activities  
Contract Accounts  
Creating a contract has a cost because you're using network storage  
Can only send transactions in response to receiving a transaction  
Transactions from an external account to a contract account can trigger code which can execute many different actions, such as transferring tokens or even creating a new contract  
Contract accounts don't have private keys. Instead, they are controlled by the logic of the smart contract code  
Smart contract wallets would solve our problems, but, as stated above, they can't initiate transactions. Since they cannot initiate transactions, users still need EOAs and those EOAs still need to pay for gas with ETH.  
There were two options to resolve this problem:  
Change the protocol - hard fork!  
Change transactions, upstream (ERC-4337)  
In a nutshell, ERC-4337 doesn't change how Ethereum transactions work. Cryptographically signed instructions from accounts still initiate transactions to update the state of the Ethereum network.  
What ERC-4337 changes is everything upstream of that signed transaction. It does this by introducing a new user intent layer that acts as a proxy for an EOA. This layer allows users to initiate transactions, with highly customizable smart contract wallets, but without using an EOA. The Ethereum network still receives what it received before - signed transactions.  
The result is a better experience for the user without changing the Ethereum protocol.  
How Account Abstraction Works  
With typical Ethereum transactions an EOA initiates and signs a transaction. That transaction is sent to Ethereum's Public Mempool, is validated and added to a block, onchain.  
Note that the following steps are primarily happening before any of that.  
1. Smart Contract Wallet Creation  
First, a new smart contract wallet must be created for a user. This wallet is owned by its creator who is designated as the signer. This signer must validate any of its future operations.  
This wallet may come with a variety of features, but it must be able to at least validate UserOperationss with a function called, validateUserOp. validateUserOp will check each UserOperation's signature, increment the nonce, and handle the operation's fees.  
2. User Operation Creation  
From the smart contract account, UserOperationss are created. These are not yet transactions, but rather represent intents from the user. These intents can represent any onchain user operation.  
UserOperation includes the details of the transaction such as sender, nonce, gas limit, max fee per gas, paymaster data (if applicable), and a signature.  
3. Signature Generation:  
The UserOperation is then signed using the private key associated with the initiating account. This signature serves to authenticate the transaction and validate that it was indeed initiated by the owner of the smart contract account.  
4. Alt Mempool:  
ERC-4337 introduces an Alt Mempool where operations are stored until they're picked up by a Bundler. The Alt Mempool is not very different from the transaction pool typically used in Ethereum, but this mempool exists earlier on in the transaction and holds user operations, where Ethereum's mempool holds signed transactions.  
5. Bundler and Operation Submission:  
Nodes on the Ethereum network have the option to serve as a Bundler, a role that involves collecting multiple signed UserOperations and consolidating them into a single transaction, called a bundle transaction. These bundle transactions are then directed towards a universal smart contract, called the EntryPoint.  
The submission of the signed UserOperation to the EntryPoint contract can be done directly or through a paymaster, which is a contract that agrees to cover the cost of operations for certain users.  
6. Operation Validation:  
The Bundler triggers a function named handleOps on the EntryPoint smart contract, which receives the bundle transaction. The EntryPoint then calls validateUserOp for each account within this bundle transaction.  
Each smart contract wallet is then required to implement an additional function and execute the actual operation sent by the EntryPoint contract.  
7. Operation Execution:  
Once the operation has been validated and the fees have been handled, the operation is executed on the Ethereum network.  
Intro to Privy  
Privy makes authentication and user-management in onchain apps easier by bridging onchain and offchain user data. In preparation to dive into our Base Paymaster example app, this lesson will cover the basics of Privy.  
In this tutorial, you'll quick start and review a sample application where users can authenticate with Privy.  
Overview of Privy  
Privy self-describes as "a simple toolkit for progressive authentication in web3". In this app, you'll primarily use Privy for its authentication and user-handling capabilities, but some of Privy's most popular features include:  
Authentication Options: Privy allows developers to configure how users authenticate. This can be with a crypto wallet, an email address, phone number or social profiles. Conveniently, Privy handles sessions and provides all necessary authentication methods.  
Progressive Onboarding: Privy creates a user object for each session. Before authentication, this user object is null, but as the user interacts with your application it will progressively associate more user information with this object. For example, users can start by authenticating with their email address and later add their wallet address or any other user information as the application requires.  
Embedded Wallets: Embedded wallets are "self-custodial Ethereum wallets that are embedded into your app. This allows your users to take wallet-based actions without ever leaving your site. Embedded wallets are the easiest way to unlock your full product experience for users who don't have, or don't want to connect, their own wallet." Developers can simply configure Privy to automatically created an Embedded Wallet on login or they can be pregenerated on the backend.  
CAUTION  
Embedded wallets are still EOAs and should not be confused with smart contract wallets. A user may have both. During development, this division can lead to confusing situations where msg.sender is not the address you were expecting.  
Privy Quick Start  
As with most onchain frontend connector libraries, you can use Privy's Quick Start to jump start your development with their platform. In this example, you'll use the NextJS starter.  
Setup  
First, navigate to the repository: https://github.com/privy-io/create-next-app and clone the repo.  
git clone https://github.com/privy-io/create-next-app  
Next, cd create-next-app and install dependencies with yarn.  
Setting Your App Id  
To use Privy, you'll need your own environment variables. In your terminal, run:  
cp .env.example .env.local  
This will create a .env.local file in your project's root. This is where you'll add your Privy App ID:  
NEXT\_PUBLIC\_PRIVY\_APP\_ID=<your-privy-app-id>  
INFO  
Note - to get an App ID, you'll need to request one and access it at [https://console.privy.io/]. This process can take a while, but for the Base community, Privy will expedite this process! Developers can send an email to base@privy.io with:  
Your app name  
The email address you want as admin  
A one liner on what you're building  
Starting the App  
Finally, run yarn dev and navigate to [http://localhost:3000] to see the starter application.  
Privy Login Walkthrough  
Before exploring the code, test the app. First, you should see this login page:  
After clicking "Log in" you'll see the following modal:  
By default, you can login with a wallet, or email.  
After logging in, you'll be redirected to the /dashboard page, where the demo app will allow you to connect a number of other accounts to your user object:  
If you navigate to console.privy.io, you'll see that Privy stores all your users and their data here.  
PrivyProvider  
Diving into the code, first look at the PrivyProvider inside of \_app.jsx:  
<PrivyProvider  
 appId={process.env.NEXT\_PUBLIC\_PRIVY\_APP\_ID || ''}  
 onSuccess={() => router.push('/dashboard')}  
>  
 <Component {...pageProps} />  
</PrivyProvider>  
PrivyProvider uses React Context and wraps any components that will use the usePrivy hook.  
Additionally, it's here that you can pass an optional config property to enable more authentication methods.  
Add a config property to the <PrivyProvider /> in \_app.jsx with 'github' and 'sms' as the login options:  
<PrivyProvider  
 appId={process.env.NEXT\_PUBLIC\_PRIVY\_APP\_ID || ''}  
 onSuccess={() => router.push('/dashboard')}  
 config={{  
 loginMethods: ['github', 'sms'],  
 }}  
>  
 <Component {...pageProps} />  
</PrivyProvider>  
Refresh to see that authentication is only possible now through Github or SMS:  
You can find a full list of loginMethods in the docs for PrivyClientConfig.  
The usePrivy Hook  
The primary method you'll use to utilize Privy's authentication features is usePrivy. Open pages/dashboard.tsx to see the methods decomposed from usePrivy in the starter, and how they are used.  
A full list of the fields and methods returned from usePrivy are documented here.  
The useWallets Hook  
To access wallet data for currently authenticated user, use the useWallets hook:  
import { ConnectedWallet, useWallets } from '@privy-io/react-auth';  
  
const { wallets } = useWallets();  
  
// wallets = [  
// {  
// "address": "0x<address>",  
// "type": "wallet",  
// "verifiedAt": "2023-11-28T19:01:41.000Z",  
// "chainType": "ethereum",  
// "chainId": "eip155:84531",  
// "walletClient": "unknown",  
// "walletClientType": "coinbase\_wallet",  
// "connectorType": "coinbase\_wallet"  
// },  
// {  
// "address": "0x<address>",  
// "type": "wallet",  
// "verifiedAt": "2023-11-28T20:09:23.000Z",  
// "chainType": "ethereum",  
// "chainId": "eip155:1",  
// "walletClient": "unknown",  
// "walletClientType": "metamask",  
// "connectorType": "injected"  
// },  
// ],  
As you can see, a user may connect multiple wallets to Privy, including Embedded Wallets.  
Embedded Wallets  
Lastly, configure your starter app to create an Embedded Wallet for your users on login.  
As stated in the Privy docs,  
Embedded wallets are self-custodial Ethereum wallets that are embedded into your app. This allows your users to take wallet-based actions without ever leaving your site. Embedded wallets are the easiest way to unlock your full product experience for users who don't have, or don't want to connect, their own wallet.  
When configuring your app to create embedded wallets on login, you have 2 options:  
users-without-wallets: This will create embedded wallets for all use who did not login with an external wallet  
all-users: This will create an additional embedded wallet for all users, regardless if they have linked an external wallet  
Inside of \_app.tsx, update your PrivyProvider:  
<PrivyProvider  
 appId={process.env.NEXT\_PUBLIC\_PRIVY\_APP\_ID || ''}  
 onSuccess={() => router.push('/dashboard')}  
 config={{  
 embeddedWallets: {  
 createOnLogin: 'all-users'  
 }  
 }}  
>  
Log out of your application, then log in again and see that your user has an additional linkedAccount which is the Privy Embedded Wallet:  
{  
 "address": "0xD5063967BA703D485e3Ca40Ecd61882dfa5F49b2",  
 "type": "wallet",  
 "verifiedAt": "2023-11-28T20:52:22.000Z",  
 "chainType": "ethereum",  
 "chainId": "eip155:1",  
 "walletClient": "privy",  
 "walletClientType": "privy",  
 "connectorType": "embedded",  
 "recoveryMethod": "privy"  
},  
More information on Privy's Embedded Wallets, including information about the addresses, signing transactions, funding the wallet and more, can be found here: https://docs.privy.io/guide/frontend/embedded/overview  
Implementing the Paymaster  
A paymaster is a type of smart contract account, introduced in ERC-4337, that is able to pay for gas on behalf of another account. In this step-by-step, you'll modify an example created by Privy, move it to another onchain app, and use it to call a smart contract function. Along the way, you'll encounter and resolve some of the confusing pitfalls associated with working with smart contract accounts.  
CAUTION  
The tutorial below does not account for recent changes to the Base Paymaster. Please reference the linked repo and adjust. We'll update the tutorial soon!  
Reviewing the Example  
Start by reviewing the paymaster example. The address in the about section of the Github page links to a deployed version of the app. It's the same app you get from Privy's Quick Start, with the addition of a mint button (the versions may be a little older).  
The app is limited to social auth, so log in with either your Google account or email. You'll see the dashboard, with the addition of a Mint NFT button at the top.  
Click the button and you'll see a toast notification informing you of updates to the transaction status. Note that this happens without you needing to approve a transaction or fund a wallet!  
Click to see the transaction when it's done to open BaseScan. If you missed it, mint another NFT, it's not like you're paying gas!  
Reviewing the Transaction and Contract  
The transaction page should appear fairly standard. You can see from it that an NFT was minted by the NFT Contract and transferred to the smart wallet address listed on the dashboard. Digging in a little more, you'll see some things that might be different than what you'd expect.  
Tokens Transferred and the NFT Contract  
In the ERC-721 Tokens Transferred section, click the link to NFT Name (NFT) to open up the overview page for the token. You'll see a list of transfers, with yours likely on the top. Click the address for the contract to open up the view for the contract itself.  
You may be surprised to see that there are very few transactions listed for this contract, despite the list of transfers you can see on the token page, or the Events tab. Currently, Etherscan and BaseScan won't display transactions done via the paymaster.  
CAUTION  
Blockchain explorers are service providers that provide information about the state of various blockchains. They are not a source of truth.  
It is possible for onchain activity to be missing from these services.  
The Bundler (Transaction Sender)  
Return to the transactions summary and look at the From: field. It will contain 0x1e6754b227c6ae4b0ca61d82f79d60660737554a. What is this address? It's not your smart wallet address or signer address. If you mint another NFT from a different login, you'll get the same sender.  
This address is the bundler, which is a special node that bundles user operations from the alt mempool into a single transaction and sends them to the single EntryPoint contract.  
EntryPoint  
The EntryPoint contract for Base Goerli is located at 0x5ff137d4b0fdcd49dca30c7cf57e578a026d2789. Strangely, in your transaction receipt you'll see that the transaction includes a transfer of ETH from the EntryPoint to the bundler. This transaction is how the bundler gets compensated for performing the service of bundling user ops and turning them into transactions -- the EntryPoint calculates the gas used by user ops and multiplies that by the fee percentage and send it to the bundler.  
Review the Example Code  
Return to the paymaster example and review the readme. The section on Copying into your code lists the three files you'll need to copy over to implement the paymaster in your own app. All three are extensively documented via comments. You'll also want to review how the demo app uses these to call a function.  
SmartAccountContext.tsx  
The first file, hooks/SmartAccountContext.tsx uses a React Context provider to create a SmartAccountProvider`` and pass it into your app. You can see it in use in \_app.tsx, with the regular PrivyProvider` around it. Review the file in detail.  
Starting on line 63, the exported SmartAccountProvider does the following:  
Fetch the user's wallets and find their Privy wallet. This wallet is provided, and if need be created, by the PrivyProvider  
Set up state variables to manage and share connection status and the smart account itself  
Initialize an RPC client for the Base Paymaster (on Goerli). The URL is hardcoded in lib/constants.ts  
Initialize an ERC-4337 RPC client for Alchemy's network. This network is where the bundler address comes from  
Create a smart wallet. In this case, the signer is your EOA embedded wallet created by Privy and fetched in the first step  
The EOA address is displayed in the example app as YOUR SIGNER ADDRESS  
Initialize an Alchemy provider for the smart account signer, using Alchemy's Account Kit.  
This creates the smart account and its address, which is displayed in the example app as YOUR SMART WALLET ADDRESS  
Finally, the sendSponsoredUserOperation function takes a traditional transaction, turns it into a user operation, adds the data for the paymaster to pay the gas, signs it, and sends it. Whew!  
If you want a deeper dive into the inner workings of this process, review the helper functions in user-operations.ts.  
How to Call a Smart Contract Function with the Paymaster  
Open pages/dashboard.tsx and take a look at the onMint function on line 32. This function is used as the oncClick handler for the Mint button at the top of the dashboard.  
If you're used to working with wagmi, you'll find the process of sending and awaiting for confirmation of a transaction a little on the manual side. Most of this will be familiar if you've used viem directly, or have worked with Ethers.  
When a user clicks, the app first creates a viem RpcTransactionRequest for the mint function on the smart contract. The smartContractAddress is supplied by the SmartAccountProvider, and the ABI and contract NFT\_ADDRESS are loaded from lib/constants.ts:  
{  
 from: smartAccountAddress,  
 to: NFT\_ADDRESS,  
 data: encodeFunctionData({  
 abi: ABI,  
 functionName: "mint",  
 args: [smartAccountAddress],  
 }),  
}  
The app then updates the toast component to update the users while it awaits first the userOpHash, then the transactionHash, indicating that transaction has completed successfully. It then updates the link in the toast to send the user to that transaction on Goerli BaseScan.  
Implementing the Paymaster in your own App  
Create a new project using Privy's create-next-app template, and complete the setup instructions in the readme.  
Add an environment variable for NEXT\_PUBLIC\_ALCHEMY\_API\_KEY and paste in the an API key for a Base Goerli app. If you need a key, go to add an app and create a new one.  
Copying and Updating the Source Files  
Copy the hooks and lib folders into your new project. You'll need to install some more dependencies. Use npm or yarn to add:  
viem  
react-dom  
@alchemy/aa-accounts  
@alchemy/aa-alchemy  
@alchemy/aa-core  
Open SmartAccountContext.tsx in your project. You'll see an error for getDefaultLightAccountFactory. The name of this function has been updated to getDefaultLightAccountFactoryAddress. Change it in the import, and where it is used in the file in the call to LightSmartContractAccount.  
Updating to Use the User's Wallet  
The app is currently configured to find and use the user's embedded Privy wallet as the signer. To change this, modify the instantiation of the SmartAccountProvider. Instead of finding the user's Privy wallet:  
// Old code to change  
  
// Get a list of all of the wallets (EOAs) the user has connected to your site  
const { wallets } = useWallets();  
// Find the embedded wallet by finding the entry in the list with a `walletClientType` of 'privy'  
const embeddedWallet = wallets.find((wallet) => wallet.walletClientType === 'privy');  
Simply grab the first wallet in the list (you'll want to do something more elegant for a production app):  
// Updated Code  
  
// Get a list of all of the wallets (EOAs) the user has connected to your site  
const { wallets } = useWallets();  
  
// Grab the first wallet on the list  
// TODO: Implement the option to allow the user to choose another wallet  
const wallet = wallets[0];  
Then, update the call at the bottom of useEffect to createSmartWallet if there is an embeddedWallet to instead create it if there is a wallet, using that wallet. You'll also need to update the dependency in the dependency array.  
useEffect(() => {  
 // Other code  
  
 if (wallet) createSmartWallet(wallet);  
}, [wallet?.address]);  
Configuring the PrivyProvider and Adding SmartAccountProvider  
By default, the PrivyProvider allows logging in with a wallet or email address. To limit it to only the wallet, update the config. You can also set the default chain here. You'll need to import baseGoerli to do so.  
You also need to import and wrap the app with SmartAccountProvider, imported from hooks/SmartAccountContext.tsx.  
DANGER  
The @alchemy/aa-core package also exports SmartAccountProvider and this export takes precedence when VSCode attempts to help you by automatically adding the import. You'll know you've got the wrong one if SmartAccountProvider generates an error that:  
'SmartAccountProvider' cannot be used as a JSX component.  
Its instance type 'SmartAccountProvider<Transport>' is not a valid JSX element.  
<PrivyProvider  
 appId={process.env.NEXT\_PUBLIC\_PRIVY\_APP\_ID || ''}  
 onSuccess={() => router.push('/dashboard')}  
 config={{  
 loginMethods: ['wallet'],  
 defaultChain: baseGoerli,  
 }}  
>  
 <SmartAccountProvider>  
 <Component {...pageProps} />  
 </SmartAccountProvider>  
</PrivyProvider>  
Checking Progress  
Grab the snippet from the original demo that displays the user's addresses, and add it to dashboard.tsx in the new project:  
<p className="mt-6 font-bold uppercase text-sm text-gray-600">  
 Your Smart Wallet Address  
</p>  
<a  
 className="mt-2 text-sm text-gray-500 hover:text-violet-600"  
 href={`${BASE\_GOERLI\_SCAN\_URL}/address/${smartAccountAddress}#tokentxnsErc721`}  
>  
 {smartAccountAddress}  
</a>  
<p className="mt-6 font-bold uppercase text-sm text-gray-600">  
 Your Signer Address  
</p>  
<a  
 className="mt-2 text-sm text-gray-500 hover:text-violet-600"  
 href={`${BASE\_GOERLI\_SCAN\_URL}/address/${eoa?.address}`}  
>  
 {eoa?.address}  
</a>  
Paste it above the <p> for the User Object window.  
You'll need to import BASE\_GOERLI\_SCAN\_URL from constants.ts. The useSmartAccount hook returns smartAccountProvider and eoa. Import it and add it under the usePrivy hook. You don't need them just yet, but go ahead and decompose smartAccountProvider and sendSponsoredUserOperation as well:  
const router = useRouter();  
const {  
 ready,  
 authenticated,  
 user,  
 logout,  
 linkEmail,  
 linkWallet,  
 unlinkEmail,  
 linkPhone,  
 unlinkPhone,  
 unlinkWallet,  
 linkGoogle,  
 unlinkGoogle,  
 linkTwitter,  
 unlinkTwitter,  
 linkDiscord,  
 unlinkDiscord,  
} = usePrivy();  
const { smartAccountAddress, smartAccountProvider, sendSponsoredUserOperation, eoa } =  
 useSmartAccount();  
Run the app. You'll now see your familiar wallet address as YOUR SIGNER ADDRESS!  
  
The app sometimes gets confused with login state after you've made changes to `config`. If you see the `Log In` button but clicking it does nothing, try manually navigating to `localhost:3000/dashboard` or clearing the cache.  
  
Calling a Smart Contract Function  
You've adjusted the foundation of the app to allow you to use the Base Goerli Paymaster with your normal wallet as the signer. Now, it's time to call a smart contract function.  
Start by using the mint function in the original example. In the DashboardPage component, add a state variable holding an empty element:  
const [transactionLink, setTransactionLink] = useState(<></>);  
Then, add a variant of the original onMint function that sets this variable and has the code related to the toast removed.  
Note: make sure you change your wallet address in args to make sure the NFT is sent to your EOA wallet address!  
const onMint = async () => {  
 // The mint button is disabled if either of these are undefined  
 if (!smartAccountProvider || !smartAccountAddress) return;  
  
 try {  
 // From a viem `RpcTransactionRequest` (e.g. calling an ERC-721's `mint` method),  
 // build and send a user operation. Gas fees will be sponsored by the Base Paymaster.  
 const userOpHash = await sendSponsoredUserOperation({  
 from: smartAccountAddress,  
 to: NFT\_ADDRESS,  
 data: encodeFunctionData({  
 abi: ABI,  
 functionName: 'mint',  
 args: [eoa?.address],  
 }),  
 });  
  
 // Once we have a hash for the user operation, watch it until the transaction has  
 // been confirmed.  
 const transactionHash = await smartAccountProvider.waitForUserOperationTransaction(userOpHash);  
  
 setTransactionLink(  
 <a href={`${BASE\_GOERLI\_SCAN\_URL}/tx/${transactionHash}`}>  
 Successfully minted! Click here to see your transaction.  
 </a>,  
 );  
 } catch (error) {  
 setTransactionLink(<p>{'Mint failed with error: ' + error}</p>);  
 }  
};  
Finally, above where you added the addresses, add a button to call the function, and display the link to the transaction:  
<button  
 onClick={onMint}  
 className="rounded-md bg-violet-600 px-4 py-2 text-sm text-white hover:bg-violet-700 disabled:bg-violet-400"  
 disabled={!smartAccountProvider || !smartAccountAddress}  
>  
 Mint NFT  
</button>;  
{  
 transactionLink;  
}  
Run it and confirm it works. You need the full transaction receipt for the process to finish, so expect to wait as long as 10 or 15 seconds.  
CAUTION  
For simplicity, we've stripped out the code to disable the button while it is minting. You'll want to implement your own solution to avoid confusing your users!  
Calling Another Function  
The Base Paymaster on Goerli is very permissive. To call another function, all you need to do is to change the RpcTransactionRequest in sendSponsoredUserOperation to match the address, abi, function name, and arguments of your function on your smart contract.  
For example, to call the claim function in the Weighted Voting contract we've used in other tutorials, you'd simply need to import the Hardhat-style artifact for the contract and use it to call the function:  
const userOpHash = await sendSponsoredUserOperation({  
 from: smartAccountAddress,  
 to: weightedVoting.address as `0x${string}`,  
 data: encodeFunctionData({  
 abi: weightedVoting.abi,  
 functionName: 'claim',  
 }),  
});  
CAUTION  
The function in this example can only be called once per address. It will then fail, because one wallet cannot claim more than one batch of tokens.  
Conclusion  
In this article, we've explored the transformative potential of Account Abstraction for the Ethereum ecosystem, highlighting how it enables smart contract accounts to initiate transactions without altering the core protocol. This innovation, coupled with the utilization of Privy for streamlined user onboarding and secure data management, marks a significant advancement towards reducing onboarding friction for onchain applications. Through a practical implementation involving Privy and the Base Paymaster, we demonstrated how users can perform onchain actions without incurring gas fees, showcasing the adaptability and user-centric benefits of these technologies. This tutorial not only sheds light on the technical workings of Account Abstraction but also illustrates its practical application in enhancing the blockchain user experience.  
Tags:account abstraction  
ON THIS PAGE  
Prerequisites  
1. Be familiar with modern, frontend web development  
2. Possess a general understanding of the EVM and smart contract development  
Implementing the Paymaster  
Intro do Account Abstraction  
The Problem Account Abstraction Solves  
Overview of ERC-4337 - Account Abstraction  
Intro to Privy  
Overview of Privy  
Privy Quick Start  
Privy Login Walkthrough  
Implementing the Paymaster  
Reviewing the Example  
Reviewing the Transaction and Contract  
Review the Example Code  
Implementing the Paymaster in your own App  
The usePrivy Hook  
The useWallets Hook  
Embedded Wallets  
Conclusion  
Reviewing the Example  
Reviewing the Transaction and Contract  
Review the Example Code  
Implementing the Paymaster in your own App  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/add-frames-to-basename  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Add Frames to A Basename  
Welcome to the exciting world of Frames for your Basename! This tutorial will guide you step-by-step through adding a frame to your Basename, allowing you to personalize and display unique content on the Base.  
What Are Frames?  
Frames are dynamic visual elements that display custom content on your Basename using any Open Graph spec extensions. You may be familiar with frames from platforms like Warpcast, where they are used to enhance social interactions like minting NFTs onchain, sharing information, and more. Open Frames work similarly, but they are independent of Warpcast and are the best way for adding frames to your Basename. With Open Frames, anyone can create and display content directly on their Basename.  
Prerequisites  
Before you begin, ensure you have the following:  
A registered Basename. If you do not have one, claim a basename now.  
A small amount of ETH to submit the text record onchain  
Navigate to Your Basename  
To get started, head over to a Basename that you own. For example:  
https://www.base.org/name/devrel  
Look for the Frame Banner  
Once you're on your profile, you’ll be greeted with a new banner inviting you to pin a frame to your profile. Look for the call-to-action button and click Try it now.  
Explore the Frame Selection Panels  
You'll now see two panels to help you choose and preview frames for your profile. The left panel contains pre-built frames created by the Base community, while the right panel allows you to preview how the selected frame will look on your Basenames profile.  
Choose a Frame  
You can select any frame from the available options. For this tutorial, we’ll use the pre-built "Pay Me" Frame by Paycaster, which allows anyone to gift you USDC or ETH.  
Click on the dropdown menu to select the "Pay Me" Frame.  
Preview the Frame  
Once you've selected the frame, click on the Show preview button to see how it will appear on your profile.  
Add the Frame to Your Profile  
If you're happy with the preview, it's time to add the frame to your profile! Click the Add frame button.  
You'll be prompted to confirm the transaction in your wallet. Approve the transaction, and the page will automatically reload.  
View Your Updated Profile  
After the page reloads, you’ll see your profile with the newly added frame. Congratulations! You’ve successfully added a frame to your Basenames profile.  
Feel free to customize your Frames further or explore new designs from the community!  
Resources  
For more information on Frames and how to customize them, check out the following resources:  
Open Frames Spec  
Frames Debugger  
Openframes Website  
Open Graph spec  
Warpcast  
Tags:account abstractionPaymaster  
ON THIS PAGE  
Prerequisites  
Navigate to Your Basename  
Look for the Frame Banner  
Explore the Frame Selection Panels  
Choose a Frame  
Preview the Frame  
Add the Frame to Your Profile  
View Your Updated Profile  
Resources  
Resources  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/build-with-thirdweb  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Building an onchain app using thirdweb  
In this tutorial you will learn how to build an app on Base using the thirdweb platform.  
To achieve this, you will deploy a smart contract for a NFT collection and create an NFT gallery app for viewing the metadata details of each NFT within the collection.  
Objectives  
By the end of this tutorial, you should be able to:  
Create an NFT collection and mint new NFTs using thirdweb.  
Develop an NFT gallery app using a prebuilt thirdweb templates.  
Prerequisites  
1. Setting Up a Coinbase Wallet  
To begin developing an app on Base, you first need to set up a web3 wallet. We recommend using the Coinbase Wallet, which can be easily created by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
2. Wallet Funding  
Blockchain transactions, including deploying smart contracts, necessitate a gas fee. Therefore, you will need to fund your wallet with ETH to cover those gas fees.  
For this tutorial, you will be deploying a contract to the Base Sepolia test network. You can fund your wallet with Base Sepolia ETH using one of the faucets listed on the Base Network Faucets page.  
Creating an NFT Collection  
Before developing an app, you need to create an NFT collection via thirdweb.  
Follow these steps to set up your NFT collection:  
Visit the thirdweb dashboard.  
Click the Connect Wallet button located in the upper right corner to connect your wallet.  
From the dashboard, select Browse contracts to explore a list of deployable smart contracts.  
Navigate to the NFTs section and select the NFT Collection smart contract.  
Click the Deploy now button.  
Provide the required details for your NFT collection:  
Contract metadata (i.e. image, name, symbol, description)  
Network (Choose Base Sepolia Testnet)  
Click Deploy Now.  
INFO  
For production / mainnet deployments select Base (mainnet) as the network rather than Base Sepolia.  
Post-deployment, you can manage your smart contract via the thirdweb dashboard.  
Currently, your NFT Collection lacks NFTs. To populate our upcoming NFT Gallery app, we will need to create several NFTs.  
Follow the steps below to mint new NFTs:  
Visit the thirdweb dashboard.  
From the dashboard, select View contracts to view all your previously deployed contracts.  
Select the NFT Collection smart contract you deployed.  
Navigate to the NFTs tab on the left-hand sidebar.  
Click Mint.  
Fill in the metadata details for the NFT (name, media, description, properties).  
Click Mint NFT.  
Repeat these steps to mint as many NFTs as you'd like.  
Building an NFT Gallery App  
With an NFT Collection in place, it's time to construct an NFT Gallery App. The thirdweb CLI provides various prebuilt and starter templates for popular app use-cases, which can significantly expedite your app development process.  
In this tutorial, we'll use the thirdweb CLI to generate a new app project using the NFT Gallery template.  
Run the following command:  
npx thirdweb create --template nft-gallery  
By default, the template is configured for an NFT collection on the Ethereum Mainnet. We will modify the code to adapt our NFT collection on the Base Sepolia Testnet.  
Follow these steps to update the template:  
Open the project using your preferred code editor.  
Open the src/consts/parameters.ts file.  
Update the contractAddress variable to your NFT collection's contract address (found on the thirdweb dashboard).  
Update the chain variable to base-sepolia.  
Update the blockExplorer variable to https://sepolia.basescan.org.  
Open the src/main.tsx file.  
Replace the file contents with the following code:  
 import React from "react";  
import ReactDOM from "react-dom/client";  
import App from "./App";  
import "./index.css";  
import { ThirdwebProvider } from "@thirdweb-dev/react";  
import { BaseSepoliaTestnet } from "@thirdweb-dev/chains";  
  
  
ReactDOM.createRoot(document.getElementById("root") as HTMLElement).render(  
 <React.StrictMode>  
 <ThirdwebProvider activeChain={BaseSepoliaTestnet}>  
 <App />  
 </ThirdwebProvider>  
 </React.StrictMode>,  
);  
The above code imports and uses BaseSepoliaTestnet to be the activeChain.  
INFO  
For production / mainnet deployments, update the information above so that the chain variable is base (step ii), the blockExplorer is https://basescan.org (step iii), and update both instances of BaseSepoliaTestnet to Base in the example javascript code.  
Running the Application  
With the updated Base Sepolia Testnet chain and your NFT collection's address, you can view your NFT collection from the application.  
To start the application, run the following command from the root directory:  
yarn dev  
Conclusion  
Congratulations on reaching the end of this tutorial! You've now learned how to create an NFT collection using Thirdweb, mint new NFTs, and build an NFT gallery app on the Base blockchain!  
As a next step, check out other prebuilt smart contracts and starter templates provided by the thirdweb platform that can help you build your next onchain app on Base.  
Tags:smart contractsnft  
ON THIS PAGE  
Prerequisites  
Creating an NFT Collection  
1. Setting Up a Coinbase Wallet  
2. Wallet Funding  
Building an NFT Gallery App  
Running the Application  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/coinbase-smart-wallet  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Coinbase Smart Wallet  
The Coinbase Smart Wallet is a new application of account abstraction that makes it easy to give the exact same onboarding to both native crypto users who have their own EOA wallet, and new users who would benefit from an easier onboarding to crypto. For the former, the wallet works almost exactly the way they are used to. For the latter, passkeys are used to create a wallet and securely save the keys.  
Most importantly, this wallet will work for the user across all apps using the Smart Wallet, across many different chains! Think of it as an onchain version of the social logins that most regular users now expect in their apps.  
Objectives  
By the end of this tutorial you should be able to:  
Connect users to your onchain app using the Coinbase Smart Wallet  
Integrate the Smart Wallet with wagmi  
Provide an easy path for your users to buy crypto and put it in their wallet  
Compare and contrast the Smart Wallet with other forms of account abstraction  
Take advantage of the Base Gasless Campaign to onboard your first users with free gas  
Prerequisites  
Random Color NFT Contract  
The app you build during this tutorial interacts with the Random Color NFT contract built in the Simple Onchain NFTs tutorial. You'll either need to complete that tutorial, deploy a copy of the contract provided at the bottom, or interact with our copy, deployed on Base Sepolia at 0x59c35beE5eAdeEDDc2c34d419459243bD70AFD72. You can also find the contract ABI in the tutorial link.  
ERC-721 Tokens  
This tutorial assumes that you are able to write, test, and deploy your own ERC-721 tokens using the Solidity programming language. If you need to learn that first, check out our content in Base Learn or the sections specific to ERC-721 Tokens!  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. Check out our tutorial on deploying with Vercel if you need a refresher!  
Onchain Apps  
The tutorial assumes you're comfortable with the basics of deploying an app and connecting it to a smart contract. If you're still learning this part, check out our tutorials in Base Learn for Building an Onchain App.  
Building an Onchain App with the Smart Wallet  
If you're an experienced frontend developer looking to build a production-ready site, check out the BOAT template. You can also use the wagmi template if you want to start with a smaller and simpler site.  
Follow the instructions to spin up a new app. Using the wagmi template, pick:  
Name as you see fit. The script does allow you to use . to install in the current folder  
React  
Next  
Initial Testing  
Run pnpm install then pnpm run dev to start the site. First, open it with your regular browser. If you have a wallet browser extension installed, you'll see two buttons to connect:  
The reason is that the wagmi template maps all the connectors it can detect to a button on the page. In this case, it has both the injected provider from your browser extension, and the Smart Wallet provider you added doing the setup steps. You'll fix this UX quirk shortly.  
Try connecting, and you'll find that it works exactly like you're expecting it to as a user who has a wallet already and is familiar with onchain apps.  
Testing as a New User  
Next, open up a private or incognito window in your browser and visit your site on localhost:3000. If you've enabled your wallet extension in this mode, you'll need to disable it.  
Here, you should see only one button. The magic happens when you click it. Pretend you are an onchain newbie and click the Create a smart wallet button.  
Select any of the options to create a new passkey.  
That's it, you've created a new wallet and logged into your app!  
The next time you try this, the app will automatically log you in. If you want to create a new wallet, use the Disconnect button, then the Coinbase Wallet Button, then click the gear and log out of the wallet.  
Improving the Experience  
By using the Smart Wallet, you've made it easy for both new and experienced users to connect with your onchain app, but there are some improvements you can make that will benefit both types of users.  
Open page.tsx and find the <div> with the Connect section:  
<div>  
 <h2>Connect</h2>  
 {connectors.map((connector) => (  
 <button key={connector.uid} onClick={() => connect({ connector, chainId })} type="button">  
 {connector.name}  
 </button>  
 ))}  
 <div>{status}</div>  
 <div>{error?.message}</div>  
</div>  
Add a <header> element to the top (or as appropriate for the UI/UX library you are using). Move the Connect <div> inside, and replace the connectors.map function with one that finds the Smart Wallet and uses it to connect. First, create the function:  
const createWallet = useCallback(() => {  
 const coinbaseWalletConnector = connectors.find(  
 (connector) => connector.id === 'coinbaseWalletSDK',  
 );  
 if (coinbaseWalletConnector) {  
 connect({ connector: coinbaseWalletConnector, chainId });  
 }  
}, [connectors, connect]);  
Then update the <header> to use it. You can also move the Disconnect button into the header. Only show the appropriate button, depending on the connection state.  
<header>  
 <div>  
 <h2>Connect</h2>  
 {account.status === 'disconnected' && (  
 <button onClick={() => createWallet()} type="button">  
 Connect  
 </button>  
 )}  
 {account.status === 'connected' && (  
 <button type="button" onClick={() => disconnect()}>  
 Disconnect  
 </button>  
 )}  
 <div>{'Status: ' + status}</div>  
 <div>{error?.message}</div>  
 </div>  
</header>  
Showing the User's Balance  
Experienced users will know that they need funds for gas and payments to interact with your app, but new users may not. Both benefit from seeing their balance shown on the app. Import useBalance and initialize it below useAccount:  
const account = useAccount();  
const balance = useBalance({ address: account.address });  
Then add it to the display when the user is logged in. You'll need to extract it from the object returned by useBalance, and you'll want to truncate some decimals. It's also a good idea to indicate the chain the user is on.  
Create a helper function to show the balance with four decimals:  
function weiToEtherString(wei: bigint) {  
 const ether = formatEther(wei);  
 return parseFloat(ether).toFixed(4).toString();  
}  
Then use it to display the user's balance:  
<div>  
 {account.status === 'connected' &&  
 'Balance: (' + account.chain?.name + ') ' + weiToEtherString(balance?.data?.value || BigInt(0))}  
</div>  
Making it Easy to Fund Wallets  
One of the most confusing things about onchain apps for new users is the number of steps and the amount of knowledge it takes to get the right amount of the right currency on the right network to be able to use these apps.  
The most important thing you can do to help your app and every app on Base is to make it easy for anyone to fund their wallet.  
In doing so, you make it much easier for people to onboard to your app, and you can take advantage of being a part of a community of developers doing the same.  
Together, we can create a gateway to bring the world onchain!  
Start by logging in to the Coinbase Developer Platform (CDP). Create an account if you don't have one yet.  
Find your App Id by going to Project Settings. It's listed as Project ID.  
You must initialize the Onramp feature. Click on the tab for Onramp, then Edit your Display wallet name. It takes a few minutes to process.  
Once you have that, you can use the One Click pay feature to set up a transaction that will allow the user to use their Coinbase retail account to easily fund their new wallet. These funds will be available for them to use on any app on Base that uses the Smart Wallet.  
To add this, first add a helper function to build the link:  
const APP\_ID = 1234; // Replace with your Project Id  
  
function buildOneClickURL() {  
 return `https://pay.coinbase.com/buy/one-click?appId=${APP\_ID}&defaultAsset=ETH&defaultPaymentMethod=ACH\_BANK\_ACCOUNT&destinationWallets=[{"address":"${account.address}","blockchains":["base"]}]&fiatCurrency=usd&presetFiatAmount=25&quoteId=fund-wallet-button`;  
}  
Then add a new button that opens the URL in a new window:  
<div>  
 {account.status === 'connected' && (  
 <button onClick={() => window.open(buildOneClickURL())}>Fund Wallet (Uses Real Money!)</button>  
 )}  
</div>  
DANGER  
This link creates real transactions taking payment in real money for real ETH on Base. During testing, you may want to hide this feature, or make it clear that it is not using testnet funds.  
For your production app, it would also be a good idea to add a tooltip explaining:  
Onchain apps use cryptocurrency to make it easy for users to send and receive payments to one another, or the app. They also use cryptocurrency to allow users to directly pay for their own computation and resource usage in a transparent manner. As such, you need funds to use this app. You can add them with the button above, and you'll be able to use these funds on any app within the Base ecosystem that uses the smart wallet.  
Connecting to the Contract  
Use a blockchain explorer to mint a few NFTs on your contract if you haven't yet.  
Add a new folder in app for components then add a component called nftList in a file of the same name. Import the address and ABI for your deployed Random Color NFT contract. Also import useAccount and useReadContract from wagmi:  
import { useAccount, useReadContract } from 'wagmi';  
import contractData from '../contracts/RandomColorNFT.json';  
Build the component and return a list of the tokens owned by the connected address, as well as the metadata for that token. First, add a type matching the struct you added to your contract:  
type NFTData = {  
 tokenId: bigint;  
 metadata: string;  
};  
Then add a state variable to hold the list of NFTs, and fetch them with useReadContract:  
export function NFTList() {  
 const account = useAccount();  
 const [nfts, setNfts] = useState<NFTData[]>([]);  
  
 const { data: nftData, refetch: refetchNftData } = useReadContract({  
 abi: contractData.abi,  
 address: contractData.address as `0x${string}`,  
 functionName: "getNFftsOwned",  
 args: [account.address],  
 });  
  
 useEffect(() => {  
 if (nftData) {  
 const newNfts = nftData as NFTData[];  
 setNfts(newNfts);  
 }  
 }, [nftData]);  
  
 return (  
 <div>  
 <h2>NFTs</h2>  
 {/\* TODO \*/}  
 </div>  
 );  
}  
Interpreting the Metadata and Image  
Add a type and helper function to convert the base64 encoded metadata to JSON:  
type JSONMetadata = {  
 name: string;  
 description: string;  
 image: string;  
};  
  
function getJsonMetadata(nft: NFTData) {  
 const base64String = nft.metadata.split(',')[1];  
 const jsonString = atob(base64String);  
 return JSON.parse(jsonString) as JSONMetadata;  
}  
The image is already in a format usable by <img> tags!  
Displaying a List of NFTs  
Now that you can extract your metadata and image from your data, use it to build a render function for your NFTs:  
function renderNft(nft: NFTData) {  
 const metadata = getJsonMetadata(nft);  
 return (  
 <div key={nft.tokenId.toString()}>  
 <h3>{metadata.name}</h3>  
 <p>{metadata.description}</p>  
 <img src={metadata.image} alt={metadata.name} />  
 </div>  
 );  
}  
And use it to display them:  
return (  
 <div>  
 <h2>NFTs</h2>  
 <div>{`Address: ${account.address}`}</div>  
 <div>{nfts.map((nft) => renderNft(nft))}</div>  
 </div>  
);  
Add further styling or use a library to improve the appearance.  
Testing with Smart Wallet  
Test with both your normal wallet in the browser, and use incognito or private mode to test with the Smart Wallet.  
Don't forget to mint some NFTs for the Smart Wallet address!  
Everything should work as expected for both.  
Adding a Mint Button  
Import and set up functions to write to your contract and wait for the receipt:  
const { data: writeData, writeContract } = useWriteContract();  
const { data: receipt } = useWaitForTransactionReceipt({  
 hash: writeData,  
});  
Wait for the receipt, and use it to trigger a refetch of the NFT data. Doing so will update the user's list of NFTs after they buy a new one:  
useEffect(() => {  
 if (receipt) {  
 refetchNftData();  
 }  
}, [receipt]);  
Finally, add a button allowing the user to purchase a new NFT:  
<button  
 onClick={() =>  
 writeContract({  
 abi: contractData.abi,  
 address: contractData.address as `0x${string}`,  
 functionName: 'mintTo',  
 args: [account.address],  
 })  
 }  
>  
 Mint NFT  
</button>  
CAUTION  
Reminder: We've constructed our contract to require an explicit address for the recipient in the mintTo function.  
Test it with your normal wallet. Everything should work as expected. Now, test it with the Smart Wallet. You don't need to fund the wallet (on testnet).  
Not only does the transaction work in a way that's easy for new users, but for a currently, Base is automatically sponsoring transactions done through the Smart Wallet (on testnet only).  
But you can also get thousands of dollars of gas sponsorship with a few setup steps during the Base Gasless Campaign. Also check out the Paymaster docs to see how to sponsor gas beyond this.  
Interacting with the Contract  
When using the Smart Wallet, you can effortlessly use the same code for both EOA wallet users and Smart Wallet users.  
Try adding a button that sends the NFT to another user. You can use one of the known testing addresses for now. Update renderNFT:  
 function renderNft(nft: NFTData) {  
 const metadata = getJsonMetadata(nft);  
 return (  
 <div key={nft.tokenId.toString()}>  
 <h3>{metadata.name}</h3>  
 <p>{metadata.description}</p>  
 <img src={metadata.image} alt={metadata.name} />  
 <button  
 onClick={() =>  
 writeContract({  
 abi: contractData.abi,  
 address: contractData.address as `0x${string}`,  
 functionName: "transferFrom",  
 args: [account.address, "0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266", nft.tokenId],  
 })  
 }  
 >Test Transfer</button>  
 </div>  
 );  
 }  
Test with both the EOA and Smart Wallet. Everything works as expected!  
Conclusion  
In this tutorial, you've learned how to connect users to your onchain app with the Coinbase Smart Wallet. You've seen how the wallet works seamlessly for both EOA and Smart Wallet users. It provides the experience expected for both audiences, without you needing to do extra development to accommodate both paths. You've also learned of a method to make it easy for both new and experienced users to fund their wallets with ETH on Base that they can use on any app in the ecosystem. Finally, you've learned where you can find resources for applying for the Base Gasless Campaign. This program will give you a gas subsidy that you can pass on to your users.  
Tags:nftSmart WalletPaymaster  
ON THIS PAGE  
Prerequisites  
Building an Onchain App with the Smart Wallet  
Initial Testing  
Testing as a New User  
Vercel  
Onchain Apps  
Improving the Experience  
Showing the User's Balance  
Making it Easy to Fund Wallets  
Connecting to the Contract  
Interpreting the Metadata and Image  
Displaying a List of NFTs  
Interacting with the Contract  
Interpreting the Metadata and Image  
Displaying a List of NFTs  
Testing with Smart Wallet  
Adding a Mint Button  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/complex-onchain-nfts  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Complex Onchain NFTs  
Many NFTs are dependent on offchain metadata and images. Some use immutable storage locations, such as IPFS. Others use traditional web locations, and many of these allow the owner of the contract to modify the URL returned by a contract when a site or user attempts to retrieve the location of the token art and metadata. This power isn't inherently bad, because we probably want someone to be able to fix the contract if the storage location goes down. However, it does introduce a requirement to trust the contract owner.  
Although challenging, it is possible to write a smart contract that contains all the necessary logic and data to generate json metadata and SVG images, entirely onchain. It will be expensive to deploy, but will be as cheap as simpler contracts to mint!  
In this tutorial, we'll show you how to do this to create your own fully-onchain art project, similar to our sample project.  
Objectives  
By the end of this tutorial you should be able to:  
Programmatically generate and return json metadata for ERC-721 tokens  
Deterministically construct unique SVG art in a smart contract  
Generate deterministic, pseudorandom numbers  
Prerequisites  
ERC-721 Tokens  
This tutorial assumes that you are able to write, test, and deploy your own ERC-721 tokens using the Solidity programming language. If you need to learn that first, check out our content in Base Learn or the sections specific to ERC-721 Tokens!  
Vector Art  
You'll need some familiarity with the SVG art format and a basic level of ability to edit and manipulate vector art. If you don't have this, find an artist friend and collaborate!  
Creating the Art Assets  
To start, you'll need to build a few vector art assets and mock up an example of what your NFT might look like. Later, you'll cut these up and format them in a way that your smart contract will use to assemble unique NFTs for each minter.  
The mockup needs to have all of the elements you plan to have in the NFT, and you should be able to manually move things around or change colors to make it so that you can create the range of variation you want. For example:  
A gradient sky in which the colors are randomized  
One of three styles of sun, always in the same spot in the upper right corner  
One to five clouds placed randomly in the upper half of the canvas  
A wide mountain ridge that will always be in the middle, but slide side to side to show a different part for each NFT  
An ocean in the foreground that is always the same  
If you are an artist, or are working with one, you can use the vector drawing tool of your choice to assemble your mockup. If not, you can use a number of stock art or AI tool options to assist you. If you do, make sure you understand any relevant laws or terms of service!  
You can also work from ours: Sample Art  
Either way, you should end up with something similar to this:  
The Art of Making it Fit  
You'll notice that the SVG is probably way too big to be placed in a smart contract. The example is 103 KB, so you'll have to be clever to make this work.  
You'll accomplish this task by splitting each element out of the mockup and deploying them into separate smart contracts. To do so, individually export each element, and make sure that the exported pieces are no bigger than about 15 KB. That way, you'll have enough space to fit each piece within the 24KiB limit for compiled bytecode.  
If you're working with the sample, you'll end up with individual SVGs for:  
Sun 1: 9 KB  
Sun 2: 9 KB  
Sun 3: 9 KB  
Ocean: 17 KB  
Mountain: 14 KB  
Cloud: 6 KB  
Sky: 802 bytes  
If you don't have the tools to do this, you can find these files here: Sample Art  
Contract Architecture  
You'll need to build and deploy a number of contracts for this project. They'll be organized in this architecture:  
Deploying this many contracts will have a cost associated with it, but once they're deployed, this contract will cost the same as any other NFT contract. Remember, pure and view functions called outside the blockchain don't cost any gas. This means that you can use multiple contracts to assemble a relatively large graphic without additional costs!  
Building the Contracts  
Create a new project using the toolkit of your choice, and add a contract called LandSeaSkyNFT. Import OpenZeppelin's ERC-721, inherit from it, and set it up with the constructor, a mint function, and a counter to keep track of the token ID:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";  
  
contract LandSeaSkyNFT is ERC721 {  
 uint public counter;  
  
 constructor() ERC721("Land, Sea, and Sky", "LSS") {}  
  
 function mint() public {  
 counter++;  
 \_safeMint(msg.sender, counter);  
 }  
}  
Overriding the \_baseURI() Function  
Normally, you'd override \_baseURI() with the base URL for the location you select to keep your NFT metadata. This could be a website, IPFS folder, or many other possible locations.  
Since this contract will be generating the .json file directly, instead set it to indicate this to the browser:  
function \_baseURI() internal pure override returns (string memory) {  
 return "data:application/json;base64,";  
}  
Importing the Base64 Library  
As indicated above, you'll be returning the json metadata in Base64 format. OpenZeppelin has a utility contract to do this. You'll also need the Strings library. Go ahead and import them:  
import "@openzeppelin/contracts/utils/Base64.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
INFO  
Base64 allows the transport of binary data over the web in a reliable way. It is not a compression algorithm, and actually increased the data size by a 4/3 ratio.  
Planning the override for the tokenURI() Function  
Next, set up your tokenURI function override. You'll need to write some other contracts to make this work, but you can write most of the code and stub out a plan for the rest to:  
Check and ensure the token ID exists  
Compile the json metadata for the token, including:  
The "name" of the NFT  
A "description"  
The "image"  
Base64 encode the above, combine it with the \_baseURI and return it.  
function tokenURI(uint \_tokenId) public view override returns (string memory) {  
 if(\_tokenId > counter) {  
 revert InvalidTokenId(\_tokenId); // Don't forget to add the error above!  
 }  
  
 string memory json = Base64.encode(  
 bytes(  
 string(  
 abi.encodePacked(  
 '{"name": "Land, Sea, and Sky #: ',  
 Strings.toString(\_tokenId),  
 '", "description": "Land, Sea, and Sky is a collection of generative art pieces stored entirely onchain.", "image": "data:image/SVG+xml;base64,',  
 "TODO: Build the SVG with the token ID as the seed",  
 '"}'  
 )  
 )  
 )  
 );  
  
 return string(abi.encodePacked(\_baseURI(), json));  
}  
CAUTION  
Getting the quotes and commas correct when the json is broken apart like this is challenging. When debugging, look here first!  
Test Your Progress  
Test your function by writing a simple test to mint an NFT, then call and log the output of the tokenURI function. You should get something similar to:  
string: data:application/json;base64,eyJuYW1lIjogIkxhbmQsIFNlYSwgYW5kIFNreSAjMSIsICJkZXNjcmlwdGlvbiI6ICJMYW5kLCBTZWEsIGFuZCBTa3kgaXMgYSBjb2xsZWN0aW9uIG9mIGdlbmVyYXRpdmUgYXJ0IHBpZWNlcyBzdG9yZWQgZW50aXJlbHkgb25jaGFpbi4iLCAiaW1hZ2UiOiAiZGF0YTppbWFnZS9zdmcreG1sO2Jhc2U2NCxUT0RPOiBCdWlsZCB0aGUgU1ZHIHdpdGggdGhlIHRva2VuIElEIGFzIHRoZSBzZWVkIn0=  
To see if it worked, you'll need to use a manual method to decode the base64 data; everything after the comma:  
eyJuYW1lIjogIkxhbmQsIFNlYSwgYW5kIFNreSAjMSIsICJkZXNjcmlwdGlvbiI6ICJMYW5kLCBTZWEsIGFuZCBTa3kgaXMgYSBjb2xsZWN0aW9uIG9mIGdlbmVyYXRpdmUgYXJ0IHBpZWNlcyBzdG9yZWQgZW50aXJlbHkgb25jaGFpbi4iLCAiaW1hZ2UiOiAiZGF0YTppbWFnZS9zdmcreG1sO2Jhc2U2NCxUT0RPOiBCdWlsZCB0aGUgU1ZHIHdpdGggdGhlIHRva2VuIElEIGFzIHRoZSBzZWVkIn0=  
You can use the terminal: echo -n '<string to decode>' | base64 --decode  
Do so, and you'll get:  
{"name": "Land, Sea, and Sky #: 1", "description": "Land, Sea, and Sky is a collection of generative art pieces stored entirely onchain.", "image": ": Build the SVG with the token ID as the seed"}  
Building the SVG  
Next, you need to build logic to compile a real, working SVG from the pieces you've saved. You'll also need to add some variation based on the ID of the NFT.  
SVG Renderer Contract  
Add a new file and contract called SVGRenderer. It doesn't need a constructor, but it will need the Strings library:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
contract SVGRenderer {  
  
}  
Open the exemplar SVG in a code editor, and using it as an example, build out a function that uses abi.encodePacked to build everything from the SVG except the actual art. That's much too big for one contract, so add stubs instead.  
Depending on the tool you used to make the SVG, there may be unneeded extras you can remove from these lines. You also don't need the items in <defs> or <styles>. You'll take advantage of the flexibility of the format to include those in the pieces returned by the supporting contract.  
function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<SVG xmlns='http://www.w3.org/2000/SVG' viewBox='0 0 1024 1024'>",  
 // TODO: Add the clouds,  
 // TODO: Add the sun,  
 // TODO: Add the land,  
 // TODO: Add the sea,  
 // TODO: Add the background,  
 "</SVG>"  
 )  
 );  
}  
Rendering the Sea  
The sea element of this NFT will be the same for all NFTs, so it makes sense to write that contract first. Create it called, SeaRenderer, with a function called render. The <g> element is the root of the different pieces of the SVG, so add that and a stub for the rest.  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
contract SeaRenderer {  
 function render() public pure returns (string memory) {  
 return // TODO: Render the sea  
 }  
}  
The next part is tricky, and a little messy. You'll need to combine parts of the individually exported SVG that has the sea art and all of its properties with the position data for this part of the art from the exemplar SVG. You'll then need to flatten it to a single line, and add it as a string constant.  
Start by opening the Ocean SVG. Change the viewBox to viewBox="0 0 1024 1024". Move the <defs> and <scripts> tag inside of the <g> tag. Open the SVG in the browser to make sure it hasn't broken.  
Next, delete the id and data-name from the top level <g> and experiment with the transform="translate(20,2.5)" property to move the art back down to the bottom of the viewport.  
With the sample art, <g transform="translate(0,700)"> should work.  
The last edits you need to make are critical - do a find/replace to change all of the cls-1 and similar classnames, to cls-land-1! Otherwise, the classes will override one another and nothing will be the right color. Also find all instances of linear-gradient and do the same.  
Make sure you change both the definitions, and where they're called!  
Finally, use the tool of your choice to minify only the outermost <g> tag and its contents. This will flatten the code to a single line and remove extra empty character spaces. Doing so makes it easier to add to your contract, and makes the data smaller. Add it as a constant string to SeaRenderer.sol:  
string constant SVG = '<g transform="translate(0,700)"<way more code></g>';  
You may need to do a find/replace and ensure you're using only one type of quote in the SVG.  
Replace your TODO with the constant.  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
string constant SVG = <A very long string!>;  
  
contract SeaRenderer {  
 function render() public pure returns (string memory) {  
 return SVG;  
 }  
}  
Test this function independently and make sure that if you paste the content inside a set of <SVG> tags, it renders as expected!  
Calling SeaRenderer  
Return to SVGRenderer.sol. Add an interface for the SeaRenderer. All of your renderer contracts will have a function called render that either takes a uint \_tokenId, or no arguments, and returns a string. Because of this, you can use a single interface for all the render contracts:  
interface ISVGPartRenderer {  
 function render() external pure returns (string memory);  
 function render(uint \_tokenId) external pure returns (string memory);  
}  
Add an instance inside the SVGRenderer contract for the SeaRenderer, and a constructor that takes an address for the SeaRenderer:  
constructor(address \_seaRenderer) {  
 seaRenderer = ISVGPartRenderer(\_seaRenderer);  
}  
Replace // TODO: Add the sea, with a call to your external function.  
function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<SVG xmlns='http://www.w3.org/2000/SVG' viewBox='0 0 1024 1024'>",  
 // TODO: Add the clouds,  
 // TODO: Add the sun,  
 // TODO: Add the land,  
 seaRenderer.render(),  
 // TODO: Add the background,  
 "</SVG>"  
 )  
 );  
}  
Finishing a First Pass  
Return to your LandSeaSkyNFT contract and add an interface for the SVGRenderer.  
interface ISVGRenderer {  
 function render(uint \_tokenId) external view returns (string memory);  
}  
Add an instance of it and update the constructor to set it:  
ISVGRenderer SVGRenderer;  
  
constructor(address \_SVGRenderer) ERC721("Land, Sea, and Sky", "LSS") {  
 SVGRenderer = ISVGRenderer(\_SVGRenderer);  
}  
INFO  
For testing purposes, it may be easier if you add functions to allow you to change these addresses after deployment. But the whole point of all this work is to make immutable, onchain NFTs, so be sure to delete them before you do your real deployment!  
Finally, swap your TODO with a line to Base64.encode a call to the renderer:  
string memory json = Base64.encode(  
 bytes(  
 string(  
 abi.encodePacked(  
 '{"name": "Land, Sea, and Sky #: ',  
 Strings.toString(tokenId),  
 '", "description": "Land, Sea, and Sky is a collection of generative art pieces stored entirely onchain.", "image": "data:image/SVG+xml;base64,',  
 Base64.encode(bytes(SVGRenderer.render(tokenId))),  
 '"}'  
 )  
 )  
 )  
);  
Test Deploy  
Now is a good time to deploy to testnet and see if this first pass is working as expected. If you're using Hardhat and Hardhat Deploy, you can use this script:  
import { HardhatRuntimeEnvironment } from 'hardhat/types';  
import { DeployFunction } from 'hardhat-deploy/types';  
  
const func: DeployFunction = async function (hre: HardhatRuntimeEnvironment) {  
 const { deployments, getNamedAccounts } = hre;  
 const { deploy } = deployments;  
 const { deployer } = await getNamedAccounts();  
  
 const SeaRenderer = await deploy('SeaRenderer', {  
 from: deployer,  
 });  
  
 const SVGRenderer = await deploy('SVGRenderer', {  
 from: deployer,  
 args: [SeaRenderer.address],  
 });  
  
 const LandSeaSkyNFT = await deploy('LandSeaSkyNFT', {  
 from: deployer,  
 args: [SVGRenderer.address],  
 });  
  
 await hre.run('verify:verify', {  
 address: LandSeaSkyNFT.address,  
 constructorArguments: [SVGRenderer.address],  
 contract: 'contracts/LandSeaSkyNFT.sol:LandSeaSkyNFT',  
 });  
};  
export default func;  
Deploy the contracts to Base Sepolia and verify them, or at least verify LandSeaSkyNFT (the above script will do this).  
Open the contract in Basescan, connect with your wallet, and mint some NFTs.  
Wait a few minutes, then open the testnet version of Opensea and look up your contract. It may take several minutes to show up, but when it does, if everything is working you'll see NFTs with the ocean part of the art! Neat!  
Adding the Sky Renderer  
Great work! Much of the hardest part is done. All you need to do now is add a renderer for each of the other elements, with the twist that you'll be doing customization inside the SVGs themselves. You'll have to do a little surgery!  
Preparing the SVG  
Open the sky SVG in both an editor, and the browser. As with for the sea, the first step is to change the viewport to 1024x1024, move the <defs> and <style> elements inside the top-level <g>, and transform/translate that group to the correct location (0,0 will work!).  
Change cls-1 to cls-sky-1 in both the definition and where it's used. Add sky to the linear-gradient as well.  
Delete the data and layer information for this group as well. You'll end up with:  
<svg xmlns="http://www.w3.org/2000/SVG" viewBox="0 0 1024 1024">  
 <g transform="translate(0,0)">  
 <defs>  
 <style>  
 cls-sky-1 {  
 fill: url(#linear-gradient-sky);  
 stroke-width: 0px;  
 }  
 </style>  
 <linearGradient  
 id="linear-gradient-sky"  
 x1="511.76"  
 y1="10.19"  
 x2="512.24"  
 y2="865.99"  
 gradientTransform="translate(63.89) scale(.88)"  
 gradientUnits="userSpaceOnUse"  
 >  
 <stop offset=".12" stop-color="#c391b4" />  
 <stop offset=".34" stop-color="#ce9f9b" />  
 <stop offset=".68" stop-color="#dfd061" />  
 <stop offset=".96" stop-color="#e3f9f7" />  
 </linearGradient>  
 </defs>  
 <path class="cls-sky-1" d="M1024,0v768.9H0V0h1024Z" />  
 </g>  
</svg>  
You have some design choices to make here. You could use deterministically chosen, but essentially random colors to make up the gradient. However, doing so will lead to the vast majority of NFTs having truly bizarre colors that don't look nice and don't look like they belong with the rest of the art.  
No one wants their sky to be a mix of cyan, teal, maroon, and brown!  
It might be better to add a gentle modification to the existing gradient colors and range. Start by minifying the top level <g> group and contents, then create two constant strings, one for everything before the first <stop> element, and one for everything after the last <stop> element.  
Neither string should have the <stop>s. You'll make those next.  
Building the Renderer Contract  
Add a new file and contract called SkyRenderer. Add your strings:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
string constant START = '<g transform="translate(0,0)"> <defs> <style>.cls-sky-1{fill: url(#linear-gradient-sky); stroke-width: 0px;}</style> <linearGradient id="linear-gradient-sky" x1="511.76" y1="10.19" x2="512.24" y2="865.99" gradientTransform="translate(63.89) scale(.88)" gradientUnits="userSpaceOnUse">';  
string constant END = '</linearGradient> </defs> <path class="cls-sky-1" d="M1024,0v768.9H0V0h1024Z"/> </g>';  
  
contract SkyRenderer {  
 function render(uint \_tokenId) public pure returns (string memory) {  
 return //TODO;  
 }  
}  
Next, add constants for the existing offset and stop-color properties:  
string constant OFFSET1 = ".12";  
string constant OFFSET2 = ".34";  
string constant OFFSET3 = ".68";  
string constant OFFSET4 = ".96";  
string constant COLOR1 = "#c391b4";  
string constant COLOR2 = "#ce9f9b";  
string constant COLOR3 = "#dfd061";  
string constant COLOR4 = "#e3f9f7";  
Now, stub out your return for the render function. It will use the built in method of using abi.encode and casting to string to combine all the parts and return them.  
function render(uint \_tokenId) public pure returns (string memory) {  
 return string(  
 abi.encodePacked(  
 START,  
 // TODO stop 1,  
 // TODO stop 2,  
 // TODO stop 3,  
 // TODO stop 4,  
 END  
 )  
 );  
}  
Do the same for a function to buildStop:  
function \_buildStop(  
 string memory \_offset,  
 string memory \_color,  
 uint \_tokenId,  
 uint \_stopNumber  
 ) internal pure returns (string memory) {  
 return string(  
 abi.encodePacked(  
 '<stop offset="',  
 // TODO tweaked offset,  
 '" stop-color="',  
 \_color,  
 '"/>'  
 )  
 );  
 }  
Now, you just need to figure out how to modify the properties based on the \_tokenId. It needs to be "random" in the sense that every NFT should be different, but it has to be deterministic, so that you get the same art every time you load the image.  
First, subtract 10 from the values and convert them to uints without decimals in each of your stop constants, and reduce the last to 80:  
uint constant OFFSET1 = 2;  
uint constant OFFSET2 = 24;  
uint constant OFFSET3 = 58;  
uint constant OFFSET4 = 80;  
You'll also need to update the parameter in \_buildStop.  
Add a function to \_buildOffsetValue. This will pick an integer between 0 and 20 for each offset, and add it to the modified offsets you just made. The result will be a change of + or 1 10 for each value (with the last being slightly different to keep it in range):  
function \_buildOffsetValue(  
 uint \_offset,  
 uint \_tokenId,  
 uint \_stopNumber  
 ) internal pure returns (string memory) {  
 bytes32 hash = keccak256(abi.encodePacked(\_offset, \_tokenId, \_stopNumber));  
 uint rand = uint(hash);  
 uint change = rand % 20; // Produces a number between 0 and 19  
 if(change >= 10) {  
 return string(  
 abi.encodePacked(  
 '.',  
 Strings.toString(\_offset + change)  
 )  
 );  
 } else {  
 return string(  
 abi.encodePacked(  
 '.',  
 '0', // 9 is .09, not .9  
 Strings.toString(\_offset + change)  
 )  
 );  
 }  
}  
This function uses hashing to create a pseudo-random number with the token id and stop as seeds, guaranteeing a consistent value, unique for each token and each stop within that token. It takes advantage of the way the offset property is interpreted - in this case, ".12+.20" == ".32".  
Finally, update your render function to call \_buildStop:  
function render(uint \_tokenId) public pure returns (string memory) {  
 return string(  
 abi.encodePacked(  
 START,  
 \_buildStop(OFFSET1, COLOR1, \_tokenId, 1),  
 \_buildStop(OFFSET2, COLOR2, \_tokenId, 2),  
 \_buildStop(OFFSET3, COLOR3, \_tokenId, 3),  
 \_buildStop(OFFSET4, COLOR4, \_tokenId, 4),  
 END  
 )  
 );  
}  
Incorporating the Sky Renderer  
Return to SVGRenderer.sol and add an instance of ISVGPartRenderer for the skyRenderer. Add an argument to the constructor and initialize it, then call the render function in place of your TODO for the background.  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
interface ISVGPartRenderer {  
 function render() external pure returns (string memory);  
 function render(uint \_tokenId) external pure returns (string memory);  
}  
  
contract SVGRenderer {  
  
 ISVGPartRenderer seaRenderer;  
 ISVGPartRenderer skyRenderer;  
  
 constructor(address \_seaRenderer, address \_skyRenderer) {  
 seaRenderer = ISVGPartRenderer(\_seaRenderer);  
 skyRenderer = ISVGPartRenderer(\_skyRenderer);  
 }  
  
 function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<SVG xmlns='http://www.w3.org/2000/SVG' viewBox='0 0 1024 1024'>",  
 // TODO: Add the clouds,  
 // TODO: Add the sun,  
 // TODO: Add the land,  
 skyRenderer.render(\_tokenId),  
 seaRenderer.render(),  
 "</SVG>"  
 )  
 );  
 }  
}  
Update your deploy script, then deploy and test as before.  
const SkyRenderer = await deploy('SkyRenderer', {  
 from: deployer,  
});  
  
const SVGRenderer = await deploy('SVGRenderer', {  
 from: deployer,  
 args: [SeaRenderer.address, SkyRenderer.address],  
});  
Test as before. Your NFTs now have the sky!  
Adding the LandRenderer  
Next up is the mountain part of the SVG. For this, you'll change the horizontal translation left to right to show a different part of the mountains for each NFT.  
Preparing the SVG  
Open the mountain SVG in both your browser and the editor. Once again, set the viewBox to 1024x1024 and move the <defs> and <styles> inside the top-level group (<g>).  
Find transform/translate values that first put the mountains so that they are at the bottom, and the left-most portion is shown, then the right-most. transform="translate(-150,350)" and transform="translate(-800,350)" are about right.  
Don't forget to add -land to the classnames!  
Writing the Contract  
Add a file and stub for the LandRenderer:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
contract LandRenderer {  
 function render(uint \_tokenId) public pure returns (string memory) {  
 return string(  
 abi.encodePacked(  
 '<g transform="translate(',  
 // TODO,  
 ',300)">',  
 END  
 )  
 );  
 }  
}  
Minify the top-level <g> element, and add a constant with everything after the opening <g> tag. Use similar techniques as before to generate an offset based on the token id, then build the SVG. You'll end up with something like this:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
string constant END = "<long SVG string>";  
  
contract LandRenderer {  
 function render(uint \_tokenId) public pure returns (string memory) {  
 return string(  
 abi.encodePacked(  
 '<g transform="translate(',  
 \_buildOffset(\_tokenId),  
 ',300)">',  
 END  
 )  
 );  
 }  
  
 function \_buildOffset(uint \_tokenId) internal pure returns (string memory) {  
 bytes32 hash = keccak256(abi.encodePacked(\_tokenId));  
 uint rand = uint(hash);  
 uint xOffset = (rand % 650) + 150; // Produces a number between 150 and 799  
 return string(abi.encodePacked("-", Strings.toString(xOffset)));  
 }  
}  
Incorporating the LandRenderer  
Update SVGRenderer:  
ISVGPartRenderer seaRenderer;  
ISVGPartRenderer skyRenderer;  
ISVGPartRenderer landRenderer;  
  
constructor(address \_seaRenderer, address \_skyRenderer, address \_landRenderer) {  
 seaRenderer = ISVGPartRenderer(\_seaRenderer);  
 skyRenderer = ISVGPartRenderer(\_skyRenderer);  
 landRenderer = ISVGPartRenderer(\_landRenderer);  
}  
  
function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<SVG xmlns='http://www.w3.org/2000/SVG' viewBox='0 0 1024 1024'>",  
 skyRenderer.render(\_tokenId),  
 landRenderer.render(\_tokenId),  
 seaRenderer.render(),  
 "</SVG>"  
 )  
 );  
}  
And the deploy script:  
const LandRenderer = await deploy('LandRenderer', {  
 from: deployer,  
});  
  
const SVGRenderer = await deploy('SVGRenderer', {  
 from: deployer,  
 args: [SeaRenderer.address, SkyRenderer.address, LandRenderer.address],  
});  
Test as before. It's starting to look really nice!  
Adding the Sun Renderer  
The sun renderer will use similar techniques as those you've already incorporated. The sun will be in the same place for all NFTs. Variation will come from each one having only one of the three suns shown in the exemplar art file.  
Preparing the SVGs  
For each of the three sun SVGs:  
Change the viewBox to 1024x1024  
Move the <defs> and <styles> into the first group  
Find the correct translation to put the sun in the upper right  
750, 100 should work with the sample art  
Add -sun to the classnames  
Writing the Contracts  
The tricky part here is that you can't fit all the suns into one contract. They're too big! Instead, split them into three, similar to the original ocean renderer. For example:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.20;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
  
string constant SVG = '<long SVG>'  
  
contract SunRenderer1 {  
 function render() public pure returns (string memory) {  
 return SVG;  
 }  
}  
Incorporating the SunRenderer  
Add the three SunRenderers as you have the other rendering contracts. You'll have to incorporate this one a little differently. Add a function that picks which SunRenderer to call, based on the NFT id.  
function pickSunRenderer(uint \_tokenId) public view returns (ISVGPartRenderer) {  
 bytes32 hash = keccak256(abi.encodePacked(\_tokenId));  
 uint rand = uint(hash);  
 uint sun = rand % 3;  
 if(sun == 0) {  
 return sunRenderer1;  
 } else if(sun == 1) {  
 return sunRenderer2;  
 } else {  
 return sunRenderer3;  
 }  
 }  
Make sure to put it after skyRenderer in the main render function!  
function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<SVG xmlns='http://www.w3.org/2000/SVG' viewBox='0 0 1024 1024'>",  
 skyRenderer.render(\_tokenId),  
 pickSunRenderer(\_tokenId).render(),  
 landRenderer.render(\_tokenId),  
 seaRenderer.render(),  
 "</SVG>"  
 )  
 );  
}  
Test it out. You've now got one of three suns in the sky for each NFT!  
Adding the Cloud Renderer  
On your own, try adding the clouds. The clouds renderer should:  
Randomly select between one and seven clouds  
Place those clouds randomly on the top half of the canvas  
Be on the layer on top of the sun and sky, but below the sea and mountains  
Conclusion  
In this tutorial, you've learned how to take advantage of the fact that offchain calls to your smart contracts don't use gas to create a significantly complex system to render complicated and unique NFTs, with the metadata and art existing entirely onchain. You've also explored some techniques for creating deterministic, pseudorandom numbers, and how to use them to add variation to your art. You've dealt with some of the many caveats and quirks of programmatically combining SVG files. Finally, you've practiced building multiple contracts that interact with one another.  
Tags:nft  
ON THIS PAGE  
Prerequisites  
Creating the Art Assets  
ERC-721 Tokens  
Vector Art  
The Art of Making it Fit  
Contract Architecture  
Building the Contracts  
Building the SVG  
SVG Renderer Contract  
Rendering the Sea  
Calling SeaRenderer  
Test Your Progress  
Finishing a First Pass  
Test Deploy  
Rendering the Sea  
Calling SeaRenderer  
Adding the Sky Renderer  
Preparing the SVG  
Adding the LandRenderer  
Preparing the SVG  
Writing the Contract  
Incorporating the LandRenderer  
Adding the Sun Renderer  
Preparing the SVGs  
Writing the Contracts  
Incorporating the SunRenderer  
Adding the Cloud Renderer  
Preparing the SVGs  
Writing the Contracts  
Incorporating the SunRenderer  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/convert-farcaster-frame-to-open-frame  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Convert Farcaster Frame to Open Frame using OnchainKit  
In this tutorial, we'll guide you through the process of converting a Farcaster Frame to an Open Frame using OnchainKit. This conversion will allow your frame to be more versatile and compatible with various platforms beyond Farcaster.  
Prerequisites  
Before starting this tutorial, ensure you have:  
An existing Farcaster Frame  
A frame server (we'll use Vercel in this example)  
Basic knowledge of Next.js and its App Router  
To convert your Farcaster Frame to an Open Frame, we need to make several changes:  
Update the frame metadata  
Modify the API endpoint (Frame server)  
Adjust button configurations  
Remove Farcaster-specific validations  
YOUR FRAME CAN SUPPORT FARCASTER AND OPEN FRAMES  
If you want to have support for Open Frames and Farcaster Frames, create a new route that is specific to the protocol you want.  
For example src/app/frame-fc is the route for farcaster frames and src/app/frame-of will render frames using the open frame spec.  
Let's go through each of these changes step by step.  
Update the Frame Metadata  
First, you need to update the frame metadata to indicate that it's an Open Frame. We'll use the isOpenFrame property from OnchainKit to specify the Open Frame protocol.  
Then, you will need to add a client protocol identifier which will be anonymous.  
Before:  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 label: 'Check expiration',  
 action: 'post',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/basename-starter-image.png`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: 'Enter a Basename TokenId',  
 },  
});  
After:  
const frameMetadata = getFrameMetadata({  
 isOpenFrame: true,  
 accepts: { anonymous: '1' },  
 buttons: [  
 {  
 label: 'Check expiration',  
 action: 'post',  
 target: `${NEXT\_PUBLIC\_URL}/api/openframe`,  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/basename-starter-image.png`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: 'Enter a Basename TokenId',  
 },  
});  
In the updated metadata, we've added two important properties:  
isOpenFrame: true: This indicates that our frame is now compatible with the Open Frame specification.  
accepts: { anonymous: '1' }: This is a required meta property for Open Frames. It specifies that our frame accepts anonymous interactions, meaning it doesn't require authentication.  
By including this property, we're essentially saying that our frame can be interacted with anonymously, making it more accessible and compatible with a wider range of clients and platforms beyond just Farcaster.  
Modifying the Frame Server  
Next, you need to update our Frame Server to handle Open Frame requests. This involves removing Farcaster-specific validations and adjusting how we process the incoming data.  
Frame servers are located in the src/app/api/ route of a Next.js project.  
Before:  
const body: FrameRequest = await req.json();  
const { isValid, message } = await getFrameMessage(body, {  
 neynarApiKey: 'NEYNAR\_ONCHAIN\_KIT',  
});  
  
if (!isValid) {  
 return new NextResponse('Message not valid', { status: 500 });  
}  
  
// Process the request...  
After:  
const body = await req.json();  
  
// Process the request using body.untrustedData instead of message  
// Remove Farcaster-specific validations  
  
console.log('body', body);  
  
let text = body.untrustedData.inputText;  
  
if (!text) {  
 return new NextResponse('No text', { status: 500 });  
}  
  
// Process the request...  
NOTE  
When converting from a Farcaster Frame to an Open Frame, the message validation process changes significantly. Farcaster Frames use signed messages to ensure authenticity, while Open Frames don't require this validation. You should be aware of this trade-off and implement additional security measures if needed.  
Adjust the Button Configurations  
For Open Frames, we need to use the target property for button actions instead of postUrl.  
Before:  
buttons: [  
 {  
 label: 'Check expiration',  
 action: 'post',  
 },  
],  
  
// additional code  
postUrl: <your-post-url>  
After:  
buttons: [  
 {  
 label: 'Check expiration',  
 action: 'post',  
 target: `${NEXT\_PUBLIC\_URL}/api/openframe`,  
 },  
],  
Removing Farcaster-specific Validations and Dependencies  
To convert your Farcaster Frame to an Open Frame, you need to remove Farcaster-specific validations and dependencies. This includes removing the getFrameMessage function and its related imports and usage. As well as any checks for FIDs (Farcaster IDs) or other Farcaster-only data.  
Before:  
import { FrameRequest, getFrameMessage, getFrameHtmlResponse } from '@coinbase/onchainkit/frame';  
  
const { isValid, message } = await getFrameMessage(body, {  
 neynarApiKey: 'NEYNAR\_ONCHAIN\_KIT',  
});  
  
if (!isValid || !message?.fid) {  
 return new NextResponse('Invalid request', { status: 400 });  
}  
  
return new NextResponse(  
 getFrameHtmlResponse({  
 //additional code  
 image: {  
 // image data  
 },  
 input: {  
 text: 'Enter a Basename TokenId',  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/openframe`,  
 }),  
);  
After:  
// Remove Farcaster-specific validations  
// Process the request using body.untrustedData  
  
import { FrameRequest, getFrameHtmlResponse } from '@coinbase/onchainkit/frame';  
import { NextRequest, NextResponse } from 'next/server';  
import { NEXT\_PUBLIC\_URL } from '../../../config';  
import { generateWarpcastURL } from 'src/utils';  
  
async function getResponse(req: NextRequest): Promise<NextResponse> {  
 const body: FrameRequest = await req.json();  
 console.log('body', body);  
  
 const text = body.untrustedData.inputText;  
  
 if (!text) {  
 return new NextResponse('No text', { status: 500 });  
 }  
  
 const state = {  
 page: 0,  
 };  
  
 return new NextResponse(  
 getFrameHtmlResponse({  
 isOpenFrame: true,  
 accepts: { anonymous: '1' },  
 buttons: [  
 {  
 label: 'Check another name',  
 action: 'post',  
 target: `${NEXT\_PUBLIC\_URL}/api/openframe`,  
 },  
 {  
 label: `Share`,  
 action: 'link',  
 target: generateWarpcastURL(  
 'When does your Basename expire?',  
 `${NEXT\_PUBLIC\_URL}/expiration-frame-open`,  
 ),  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/api/og?tokenId=${text}`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: 'Enter a Basename TokenId',  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/openframe`,  
 state: {  
 page: state?.page + 1,  
 time: new Date().toISOString(),  
 },  
 }),  
 );  
}  
  
export async function POST(req: NextRequest): Promise<Response> {  
 return getResponse(req);  
}  
  
export const dynamic = 'force-dynamic';  
Repeat this process for any of the additional routes you may want to convert from Farcaster Frames to Open Frames.  
Redploy your frame  
Before redeploying your Frame check how the routes differ using the Frame Debugger. The frame debugger should be used to validate that your frame follows the Openframe protocol spec.  
USE Anonymous  
Frame Dugger allows you to select the protocol you wish to debug. For Open Frame, use the anonymous (openframes) option  
Before:  
After:  
Your converted Farcaster Frame should pass all checks like the below image:  
Your frame should still have functionality w/o the farcaster dependency. Use the Farcaster Frame Validator to confirm.  
Conclusion  
By following these steps, you've successfully converted your Farcaster Frame to an Open Frame using Onchain Kit. Your frame is now more versatile and can be used across various platforms that support Open Frames.  
Remember to test your converted frame thoroughly to ensure it works as expected in different environments. Happy framing!  
Resources  
Open Frames Specification  
Onchain Kit Documentation  
Farcaster Documentation  
Frame Debugger  
Tags:frontend  
ON THIS PAGE  
Update the Frame Metadata  
Modifying the Frame Server  
Before:  
After:  
Adjust the Button Configurations  
Before:  
After:  
Removing Farcaster-specific Validations and Dependencies  
Before:  
After:  
Redploy your frame  
Before:  
After:  
Conclusion  
Before:  
After:  
Resources  
Resources  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/cross-chain-with-ccip  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Sending messages and tokens from Base to other chains using Chainlink CCIP  
This tutorial will guide you through the process of sending messages and tokens from a Base smart contract to another smart contract on a different chain using Chainlink's Cross-chain Interoperability Protocol (CCIP).  
Objectives  
By the end of this tutorial you should be able to do the following:  
Set up a smart contract project for Base using Foundry  
Install Chainlink CCIP as a dependency  
Use Chainlink CCIP within your smart contract to send messages and/or tokens to contracts on other different chains  
Deploy and test your smart contracts on Base testnet  
INFO  
Chainlink CCIP is in an “Early Access” development stage, meaning some of the functionality described within this tutorial is under development and may change in later versions.  
Prerequisites  
Foundry  
This tutorial requires you to have Foundry installed.  
From the command-line (terminal), run: curl -L https://foundry.paradigm.xyz | bash  
Then run foundryup, to install the latest (nightly) build of Foundry  
For more information, see the Foundry Book installation guide.  
Coinbase Wallet  
In order to deploy a smart contract, you will first need a wallet. You can create a wallet by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
Wallet funds  
For this tutorial you will need to fund your wallet with both ETH and LINK on Base Goerli and Optimism Goerli.  
The ETH is required for covering gas fees associated with deploying smart contracts to the blockchain, and the LINK token is required to pay for associated fees when using CCIP.  
To fund your wallet with ETH on Base Goerli, visit a faucet listed on the Base Faucets page.  
To fund your wallet with ETH on Optimism Goerli, visit a faucet listed on the Optimism Faucets page.  
To fund your wallet with LINK, visit the Chainlink Faucet.  
INFO  
If you are interested in building on Mainnet, you will need to apply for Chainlink CCIP mainnet access.  
What is Chainlink CCIP?  
Chainlink CCIP (Cross-chain Interoperability Protocol) provides a solution for sending message data and transferring tokens across different chains.  
The primary way for users to interface with Chainlink CCIP is through smart contracts known as Routers. A Router contract is responsible for initiating cross-chain interactions.  
Users can interact with Routers to perform the following cross-chain capabilities:  
Capability Description Supported receivers  
Arbitrary messaging Send arbitrary (encoded) data from one chain to another. Smart contracts only  
Token transfers Send tokens from one chain to another. Smart contracts or EOAs  
Programmable token transfers Send tokens and arbitrary (encoded) data from one chain to another, in a single transaction. Smart contracts only  
DANGER  
Externally owned accounts (EOAs) on EVM blockchains are unable to receive message data, because of this, only smart contracts are supported as receivers when sending arbitrary messages or programmable token transfers. Any attempt to send a programmable token transfer (data and tokens) to an EOA, will result in only the tokens being received.  
High-level concepts  
Although Routers are the primary interface users will interact with when using CCIP, this section will cover what happens after instructions for a cross-chain interaction are sent to a Router.  
OnRamps  
Once a Router receives an instruction for a cross-chain interaction, it passes it on to another contract known as an OnRamp. OnRamps are responsible for a variety of tasks, including: verifying message size and gas limits, preserving the sequencing of messages, managing any fee payments, and interacting with the token pool to lock or burn tokens if a token transfer is being made.  
OffRamps  
The destination chain will have a contract known as an OffRamp. OffRamps are responsible for a variety of tasks, including: ensuring the authenticity of a message, making sure each transaction is only executed once, and transmitting received messages to the Router contract on the destination chain.  
Token pools  
A token pool is an abstraction layer over ERC-20 tokens that facilitates OnRamp and OffRamp token-related operations. They are configured to use either a Lock and Unlock or Burn and Mint mechanism, depending on the type of token.  
For example, because blockchain-native gas tokens (i.e. ETH, MATIC, AVAX) can only be minted on their native chains, a Lock and Mint mechanism must be used. This mechanism locks the token at the source chain, and mints a synthetic asset on the destination chain.  
In contrast, tokens that can be minted on multiple chains (i.e. USDC, USDT, FRAX, etc.), token pools can use a Burn and Mint mechanism, where the token is burnt on the source chain and minted on the destination chain.  
Risk Management Network  
Between instructions for a cross-chain interaction making its way from an OnRamp on the source chain to an OffRamp on the destination chain, it will pass through the Risk Management Network.  
The Risk Management Network is a secondary validation service built using a variety of offchain and onchain components, with the responsibilities of monitoring all chains against abnormal activities.  
INFO  
A deep-dive on the technical details of each of these components is too much to cover in this tutorial, but if interested you can learn more by visiting the Chainlink documentation.  
Creating a project  
Before you begin, you need to set up your smart contract development environment. You can setup a development environment using tools like Hardhat or Foundry. For this tutorial you will use Foundry.  
To create a new Foundry project, first create a new directory:  
mkdir myproject  
Then run:  
cd myproject  
forge init  
This will create a Foundry project with the following basic layout:  
.  
├── foundry.toml  
├── script  
├── src  
└── test  
INFO  
You can delete the src/Counter.sol, test/Counter.t.sol, and script/Counter.s.sol boilerplate files that were generated with the project, as you will not be needing them.  
Installing Chainlink smart contracts  
To use Chainlink CCIP within your Foundry project, you need to install Chainlink CCIP smart contracts as a project dependency using forge install.  
To install Chainlink CCIP smart contracts, run:  
forge install smartcontractkit/ccip --no-commit  
Once installed, update your foundry.toml file by appending the following line:  
remappings = ['@chainlink/contracts-ccip/=lib/ccip/contracts']  
Writing the smart contracts  
The most basic use case for Chainlink CCIP is to send data and/or tokens between smart contracts on different blockchains.  
To accomplish this, in this tutorial, you will need to create two separate smart contracts:  
Sender contract: A smart contract that interacts with CCIP to send data and tokens.  
Receiver contract: A smart contract that interacts with CCIP to receive data and tokens.  
Creating a Sender contract  
The code snippet below is for a basic smart contract that uses CCIP to send data:  
pragma solidity ^0.8.0;  
  
import {IRouterClient} from "@chainlink/contracts-ccip/src/v0.8/ccip/interfaces/IRouterClient.sol";  
import {OwnerIsCreator} from "@chainlink/contracts-ccip/src/v0.8/shared/access/OwnerIsCreator.sol";  
import {Client} from "@chainlink/contracts-ccip/src/v0.8/ccip/libraries/Client.sol";  
import {IERC20} from "@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/token/ERC20/IERC20.sol";  
  
contract Sender is OwnerIsCreator {  
  
 IRouterClient private router;  
 IERC20 private linkToken;  
  
 /// @notice Initializes the contract with the router and LINK token address.  
 /// @param \_router The address of the router contract.  
 /// @param \_link The address of the link contract.  
 constructor(address \_router, address \_link) {  
 router = IRouterClient(\_router);  
 linkToken = IERC20(\_link);  
 }  
  
 /// @notice Sends data to receiver on the destination chain.  
 /// @param destinationChainSelector The identifier (aka selector) for the destination blockchain.  
 /// @param receiver The address of the recipient on the destination blockchain.  
 /// @param text The string text to be sent.  
 /// @return messageId The ID of the message that was sent.  
 function sendMessage(  
 uint64 destinationChainSelector,  
 address receiver,  
 string calldata text  
 ) external onlyOwner returns (bytes32 messageId) {  
 Client.EVM2AnyMessage memory message = Client.EVM2AnyMessage({  
 receiver: abi.encode(receiver), // Encode receiver address  
 data: abi.encode(text), // Encode text to send  
 tokenAmounts: new Client.EVMTokenAmount[](0), // Empty array indicating no tokens are being sent  
 extraArgs: Client.\_argsToBytes(  
 Client.EVMExtraArgsV1({gasLimit: 200\_000}) // Set gas limit  
 ),  
 feeToken: address(linkToken) // Set the LINK as the feeToken address  
 });  
  
 // Get the fee required to send the message  
 uint256 fees = router.getFee(  
 destinationChainSelector,  
 message  
 );  
  
 // Revert if contract does not have enough LINK tokens for sending a message  
 require(linkToken.balanceOf(address(this)) > fees, "Not enough LINK balance");  
  
 // Approve the Router to transfer LINK tokens on contract's behalf in order to pay for fees in LINK  
 linkToken.approve(address(router), fees);  
  
 // Send the message through the router  
 messageId = router.ccipSend(destinationChainSelector, message);  
  
 // Return the messageId  
 return messageId;  
 }  
}  
Create a new file under your project's src/ directory named Sender.sol and copy the code above into the file.  
Code walkthrough  
The sections below provide a detailed explanation for the code for the Sender contract provided above.  
Initializing the contract  
In order to send data using CCIP, the Sender contract will need access to the following dependencies:  
The Router contract: This contract serves as the primary interface when using CCIP to send and receive messages and tokens.  
The fee token contract: This contract serves as the contract for the token that will be used to pay fees when sending messages and tokens. For this tutorial, the contract address for LINK token is used.  
The Router contract address and LINK token address are passed in as parameters to the contract's constructor and stored as member variables for later for sending messages and paying any associated fees.  
contract Sender is OwnerIsCreator {  
  
 IRouterClient private router;  
 IERC20 private linkToken;  
  
 /// @notice Initializes the contract with the router and LINK token address.  
 /// @param \_router The address of the router contract.  
 /// @param \_link The address of the link contract.  
 constructor(address \_router, address \_link) {  
 router = IRouterClient(\_router);  
 linkToken = IERC20(\_link);  
 }  
The Router contract provides two important methods that can be used when sending messages using CCIP:  
getFee: Given a chain selector and message, returns the fee amount required to send the message.  
ccipSend: Given a chain selector and message, sends the message through the router and returns an associated message ID.  
The next section describes how these methods are utilized to send a message to another chain.  
Sending a message  
The Sender contract defines a custom method named sendMessage that utilizes the methods described above in order to:  
Construct a message using the EVM2AnyMessage method provided by the Client CCIP library, using the following data:  
receiver: The receiver contract address (encoded).  
data: The text data to send with the message (encoded).  
tokenAmounts: The amount of tokens to send with the message. For sending just an arbitrary message this field is defined as an empty array (new Client.EVMTokenAmount[](0)), indicating that no tokens will be sent.  
extraArgs: Extra arguments associated with the message, such as gasLimit.  
feeToken: The address of the token to be used for paying fees.  
Get the fees required to send the message using the getFee method provided by the Router contract.  
Check that the contract holds an adequate amount of tokens to cover the fee. If not, revert the transaction.  
Approve the Router contract to transfer tokens on the Sender contracts behalf in order to cover the fees.  
Send the message to a specified chain using the Router contract's ccipSend method.  
Return a unique ID associated with the sent message.  
/// @param receiver The address of the recipient on the destination blockchain.  
/// @param text The string text to be sent.  
/// @return messageId The ID of the message that was sent.  
function sendMessage(  
 uint64 destinationChainSelector,  
 address receiver,  
 string calldata text  
) external onlyOwner returns (bytes32 messageId) {  
 Client.EVM2AnyMessage memory message = Client.EVM2AnyMessage({  
 receiver: abi.encode(receiver), // Encode receiver address  
 data: abi.encode(text), // Encode text to send  
 tokenAmounts: new Client.EVMTokenAmount[](0), // Empty array indicating no tokens are being sent  
 extraArgs: Client.\_argsToBytes(  
 Client.EVMExtraArgsV1({gasLimit: 200\_000}) // Set gas limit  
 ),  
 feeToken: address(linkToken) // Set the LINK as the feeToken address  
 });  
  
 // Get the fee required to send the message  
 uint256 fees = router.getFee(  
 destinationChainSelector,  
 message  
 );  
  
 // Revert if contract does not have enough LINK tokens for sending a message  
 require(linkToken.balanceOf(address(this)) > fees, "Not enough LINK balance");  
  
 // Approve the Router to transfer LINK tokens on contract's behalf in order to pay for fees in LINK  
 linkToken.approve(address(router), fees);  
 // Send the message through the router  
 messageId = router.ccipSend(destinationChainSelector, message);  
  
 // Return the messageId  
 return messageId;  
}  
Creating a Receiver contract  
The code snippet below is for a basic smart contract that uses CCIP to receive data:  
pragma solidity ^0.8.0;  
  
import {Client} from "@chainlink/contracts-ccip/src/v0.8/ccip/libraries/Client.sol";  
import {CCIPReceiver} from "@chainlink/contracts-ccip/src/v0.8/ccip/applications/CCIPReceiver.sol";  
  
contract Receiver is CCIPReceiver {  
  
 bytes32 private \_messageId;  
 string private \_text;  
  
 /// @notice Constructor - Initializes the contract with the router address.  
 /// @param router The address of the router contract.  
 constructor(address router) CCIPReceiver(router) {}  
  
 /// @notice Handle a received message  
 /// @param message The cross-chain message being received.  
 function \_ccipReceive(  
 Client.Any2EVMMessage memory message  
 ) internal override {  
 \_messageId = message.messageId; // Store the messageId  
 \_text = abi.decode(message.data, (string)); // Decode and store the message text  
 }  
  
 /// @notice Gets the last received message.  
 /// @return messageId The ID of the last received message.  
 /// @return text The last received text.  
 function getMessage()  
 external  
 view  
 returns (bytes32 messageId, string memory text)  
 {  
 return (\_messageId, \_text);  
 }  
}  
Create a new file under your project’s src/ directory named Receiver.sol and copy the code above into the file.  
Code walkthrough  
The sections below provide a detailed explanation for the code for the Receiver contract provided above.  
Initializing the contract  
In order to receive data using CCIP, the Receiver contract will need to extend to theCCIPReceiver interface. Extending this interface allows the Receiver contract to initialize the contract with the router address from the constructor, as seen below:  
import {CCIPReceiver} from "@chainlink/contracts-ccip/src/v0.8/ccip/applications/CCIPReceiver.sol";  
  
contract Receiver is CCIPReceiver {  
  
 /// @notice Constructor - Initializes the contract with the router address.  
 /// @param router The address of the router contract.  
 constructor(address router) CCIPReceiver(router) {}  
}  
Receiving a message  
Extending the CCIPReceiver interface also allows the Receiver contract to override the \_ccipReceive handler method for when a message is received and define custom logic.  
/// @notice Handle a received message  
/// @param message The cross-chain message being received.  
function \_ccipReceive(  
 Client.Any2EVMMessage memory message  
) internal override {  
 // Add custom logic here  
}  
The Receiver contract in this tutorial provides custom logic that stores the messageId and text (decoded) as member variables.  
contract Receiver is CCIPReceiver {  
  
 bytes32 private \_messageId;  
 string private \_text;  
  
 /// @notice Constructor - Initializes the contract with the router address.  
 /// @param router The address of the router contract.  
 constructor(address router) CCIPReceiver(router) {}  
  
 /// @notice Handle a received message  
 /// @param message The cross-chain message being received.  
 function \_ccipReceive(  
 Client.Any2EVMMessage memory message  
 ) internal override {  
 \_messageId = message.messageId; // Store the messageId  
 \_text = abi.decode(message.data, (string)); // Decode and store the message text  
 }  
}  
Retrieving a message  
The Receiver contract defines a custom method named getMessage that returns the details of the last received message \_messagId and \_text. This method can be called to fetch the message data details after the \_ccipReceive receives a new message.  
/// @notice Gets the last received message.  
/// @return messageId The ID of the last received message.  
/// @return text The last received text.  
function getMessage()  
 external  
 view  
 returns (bytes32 messageId, string memory text)  
{  
 return (\_messageId, \_text);  
}  
Compiling the smart contracts  
To compile your smart contracts, run:  
forge build  
Deploying the smart contract  
Setting up your wallet as the deployer  
Before you can deploy your smart contract to the Base network, you will need to set up a wallet to be used as the deployer.  
To do so, you can use the cast wallet import command to import the private key of the wallet into Foundry's securely encrypted keystore:  
cast wallet import deployer --interactive  
After running the command above, you will be prompted to enter your private key, as well as a password for signing transactions.  
CAUTION  
For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation. It is critical that you do NOT commit this to a public repo.  
To confirm that the wallet was imported as the deployer account in your Foundry project, run:  
cast wallet list  
Setting up environment variables  
To setup your environment, create an .env file in the home directory of your project, and add the RPC URLs, CCIP chain selectors, CCIP router addresses, and LINK token addresses for both Base Goerli and Optimism Goerli testnets:  
BASE\_GOERLI\_RPC="https://goerli.base.org"  
OPTIMISM\_GOERLI\_RPC="https://goerli.optimism.io"  
  
BASE\_GOERLI\_CHAIN\_SELECTOR=5790810961207155433  
OPTIMISM\_GOERLI\_CHAIN\_SELECTOR=2664363617261496610  
  
BASE\_GOERLI\_ROUTER\_ADDRESS="0x80AF2F44ed0469018922c9F483dc5A909862fdc2"  
OPTIMISM\_GOERLI\_ROUTER\_ADDRESS="0xcc5a0B910D9E9504A7561934bed294c51285a78D"  
  
BASE\_GOERLI\_LINK\_ADDRESS="0x6D0F8D488B669aa9BA2D0f0b7B75a88bf5051CD3"  
OPTIMISM\_GOERLI\_LINK\_ADDRESS="0xdc2CC710e42857672E7907CF474a69B63B93089f"  
Once the .env file has been created, run the following command to load the environment variables in the current command line session:  
source .env  
Deploying the smart contracts  
With your contracts compiled and environment setup, you are ready to deploy the smart contracts.  
To deploy a smart contract using Foundry, you can use the forge create command. The command requires you to specify the smart contract you want to deploy, an RPC URL of the network you want to deploy to, and the account you want to deploy with.  
INFO  
Your wallet must be funded with ETH on the Base Goerli and Optimism Goerli to cover the gas fees associated with the smart contract deployment. Otherwise, the deployment will fail.  
To get testnet ETH for Base Goerli and Optimism Goerli, see the prerequisites.  
Deploying the Sender contract to Base Goerli  
To deploy the Sender smart contract to the Base Goerli testnet, run the following command:  
forge create ./src/Sender.sol:Sender --rpc-url $BASE\_GOERLI\_RPC --constructor-args $BASE\_GOERLI\_ROUTER\_ADDRESS $BASE\_GOERLI\_LINK\_ADDRESS --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet's private key.  
After running the command above, the contract will be deployed on the Base Goerli test network. You can view the deployment status and contract by using a block explorer.  
Deploying the Receiver contract to Optimism Goerli  
To deploy the Receiver smart contract to the Optimism Goerli testnet, run the following command:  
forge create ./src/Receiver.sol:Receiver --rpc-url $OPTIMISM\_GOERLI\_RPC --constructor-args $OPTIMISM\_GOERLI\_ROUTER\_ADDRESS --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet's private key.  
After running the command above, the contract will be deployed on the Optimism Goerli test network. You can view the deployment status and contract by using the OP Goerli block explorer.  
Funding your smart contracts  
In order to pay for the fees associated with sending messages, the Sender contract will need to hold a balance of LINK tokens.  
Fund your contract directly from your wallet, or by running the following cast command:  
cast send $BASE\_GOERLI\_LINK\_ADDRESS --rpc-url $BASE\_GOERLI\_RPC "transfer(address,uint256)" <SENDER\_CONTRACT\_ADDRESS> 5 --account deployer  
The above command sends 5 LINK tokens on Base Goerli testnet to the Sender contract.  
INFO  
Replace <SENDER\_CONTRACT\_ADDRESS> with the contract address of your deployed Sender contract before running the provided cast command.  
Interacting with the smart contract  
Foundry provides the cast command-line tool that can be used to interact with deployed smart contracts and call their functions.  
Sending data  
The cast command can be used to call the sendMessage(uint64, address, string) function on the Sender contract deployed to Base Goerli in order to send message data to the Receiver contract on Optimism Goerli.  
To call the sendMessage(uint64, address, string) function of the Sender smart contract, run:  
cast send <SENDER\_CONTRACT\_ADDRESS> --rpc-url $BASE\_GOERLI\_RPC "sendMessage(uint64, address, string)" $OPTIMISM\_GOERLI\_CHAIN\_SELECTOR <RECEIVER\_CONTRACT\_ADDRESS> "Based" --account deployer  
The command above calls the sendMessage(uint64, address, string) to send a message. The parameters passed in to the method include: The chain selector to the destination chain (Optimism Goerli), the Receiver contract address, and the text data to be included in the message (Based).  
INFO  
Replace <SENDER\_CONTRACT\_ADDRESS> and <RECEIVER\_CONTRACT\_ADDRESS> with the contract addresses of your deployed Sender and Receiver contracts respectively before running the provided cast command.  
After running the command, a unique messageId should be returned.  
Once the transaction has been finalized, it will take a few minutes for CCIP to deliver the data to Optimism Goerli and call the ccipReceive function on the Receiver contract.  
INFO  
You can use the CCIP explorer to see the status of the CCIP transaction.  
Receiving data  
The cast command can also be used to call the getMessage() function on the Receiver contract deployed to Optimism Goerli in order to read the received message data.  
To call the getMessage() function of the Receiver smart contract, run:  
cast send <RECEIVER\_CONTRACT\_ADDRESS> --rpc-url $OPTIMISM\_GOERLI\_RPC "getMessage()" --account deployer  
INFO  
Replace <RECEIVER\_CONTRACT\_ADDRESS> with the contract addresses of your deployed Receiver contract before running the provided cast command.  
After running the command, the messageId and text of the last received message should be returned.  
If the transaction fails, ensure the status of your ccipSend transaction has been finalized. You can using the CCIP explorer.  
Conclusion  
Congratulations! You have successfully learned how to perform cross-chain messaging on Base using Chainlink CCIP.  
To learn more about cross-chain messaging and Chainlink CCIP, check out the following resources:  
Cross-chain  
Chainlink CCIP  
Tags:cross-chain  
ON THIS PAGE  
Prerequisites  
What is Chainlink CCIP?  
High-level concepts  
Coinbase Wallet  
Wallet funds  
Creating a project  
High-level concepts  
Installing Chainlink smart contracts  
Writing the smart contracts  
Compiling the smart contracts  
Creating a Sender contract  
Creating a Receiver contract  
Deploying the smart contract  
Interacting with the smart contract  
Sending data  
Receiving data  
Deploying the smart contracts  
Funding your smart contracts  
Conclusion  
Sending data  
Receiving data  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/cross-chain-with-layerzero  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Sending messages from Base to other chains using LayerZero V2  
This tutorial will guide you through the process of sending cross-chain message data from a Base smart contract to another smart contract on a different chain using LayerZero V2.  
Objectives  
By the end of this tutorial you should be able to do the following:  
Set up a smart contract project for Base using Foundry  
Install the LayerZero smart contracts as a dependency  
Use LayerZero to send messages and from smart contracts on Base to smart contracts on different chains  
Deploy and test your smart contracts on Base testnet  
Prerequisites  
Foundry  
This tutorial requires you to have Foundry installed.  
From the command-line (terminal), run: curl -L https://foundry.paradigm.xyz | bash  
Then run foundryup, to install the latest (nightly) build of Foundry  
For more information, see the Foundry Book installation guide.  
Coinbase Wallet  
In order to deploy a smart contract, you will first need a wallet. You can create a wallet by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
Wallet funds  
To complete this tutorial, you will need to fund a wallet with ETH on Base Goerli and Optimism Goerli.  
The ETH is required for covering gas fees associated with deploying smart contracts to each network.  
To fund your wallet with ETH on Base Goerli, visit a faucet listed on the Base Faucets page.  
To fund your wallet with ETH on Optimism Goerli, visit a faucet listed on the Optimism Faucets page.  
What is LayerZero?  
LayerZero is an interoperability protocol that allows developers to build applications (and tokens) that can connect to multiple blockchains. LayerZero defines these types of applications as "omnichain" applications.  
The LayerZero protocol is made up of immutable on-chain Endpoints, a configurable Security Stack, and a permissionless set of Executors that transfer messages between chains.  
High-level concepts  
Endpoints  
Endpoints are immutable LayerZero smart contracts that implement a standardized interface for your own smart contracts to use and in order to manage security configurations and send and receive messages.  
Security Stack (DVNs)  
The Security Stack is a configurable set of required and optional Decentralized Verifier Networks (DVNs). The DVNs are used to verify message payloads to ensure integrity of your application's messages.  
Executors  
Executors are responsible for initiating message delivery. They will automatically execute the lzReceive function of the endpoint on the destination chain once a message has been verified by the Security Stack.  
Creating a project  
Before you begin, you need to set up your smart contract development environment by creating a Foundry project.  
To create a new Foundry project, first create a new directory:  
mkdir myproject  
Then run:  
cd myproject  
forge init  
This will create a Foundry project with the following basic layout:  
.  
├── foundry.toml  
├── script  
├── src  
└── test  
INFO  
You can delete the src/Counter.sol, test/Counter.t.sol, and script/Counter.s.sol boilerplate files that were generated with the project, as you will not be needing them.  
Installing the LayerZero smart contracts  
To use LayerZero within your Foundry project, you need to install the LayerZero smart contracts and their dependencies using forge install.  
To install LayerZero smart contracts and their dependencies, run the following commands:  
forge install GNSPS/solidity-bytes-utils --no-commit  
forge install OpenZeppelin/openzeppelin-contracts@v4.9.4 --no-commit  
forge install LayerZero-Labs/LayerZero-v2 --no-commit  
Once installed, update your foundry.toml file by appending the following lines:  
remappings = [  
 '@openzeppelin/contracts/=lib/openzeppelin-contracts/contracts',  
 'solidity-bytes-utils/=lib/solidity-bytes-utils',  
 '@layerzerolabs/lz-evm-oapp-v2/=lib/LayerZero-v2/oapp',  
 '@layerzerolabs/lz-evm-protocol-v2/=lib/LayerZero-v2/protocol',  
 '@layerzerolabs/lz-evm-messagelib-v2/=lib/LayerZero-v2/messagelib',  
]  
  
Getting started with LayerZero  
LayerZero provides a smart contract standard called OApp that is intended for omnichain messaging and configuration.  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
import { OAppSender } from "./OAppSender.sol";  
import { OAppReceiver, Origin } from "./OAppReceiver.sol";  
import { OAppCore } from "./OAppCore.sol";  
  
abstract contract OApp is OAppSender, OAppReceiver {  
 constructor(address \_endpoint) OAppCore(\_endpoint, msg.sender) {}  
  
 function oAppVersion() public pure virtual returns (uint64 senderVersion, uint64 receiverVersion) {  
 senderVersion = SENDER\_VERSION;  
 receiverVersion = RECEIVER\_VERSION;  
 }  
}  
INFO  
You can view the source code for this contract on GitHub.  
To get started using LayerZero, developers simply need to inherit from the OApp contract, and implement the following two inherited functions:  
\_lzSend: A function used to send an omnichain message  
\_lzReceive: A function used to receive an omnichain message  
In this tutorial, you will be implementing the OApp standard into your own project to add the capability to send messages from a smart contract on Base to a smart contract on Optimism.  
INFO  
An extension of the OApp contract standard known as OFT is also available for supporting omnichain fungible token transfers.  
INFO  
For more information on transferring tokens across chains using LayerZero, visit the LayerZero documentation.  
Writing the smart contract  
To get started, create a new Solidity smart contract file in your project's src/ directory named ExampleContract.sol, and add the following content:  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
import { OApp, Origin, MessagingFee } from "@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/OApp.sol";  
  
contract ExampleContract is OApp {  
 constructor(address \_endpoint, address \_owner) OApp(\_endpoint, \_owner) {}  
}  
The code snippet above defines a new smart contract named ExampleContract that extends the OApp contract standard.  
The contract's constructor expects two arguments:  
\_endpoint: The LayerZero Endpoint address for the chain the smart contract is deployed to.  
\_owner: The address of the owner of the smart contract.  
INFO  
LayerZero Endpoints are smart contracts that expose an interface for OApp contracts to manage security configurations and send and receive messages via the LayerZero protocol.  
Implementing message sending (\_lzSend)  
To send messages to another chain, your smart contract must call the \_lzSend function inherited from the OApp contract.  
Add a new custom function named sendMessage to your smart contract that has the following content:  
/// @notice Sends a message from the source chain to the destination chain.  
/// @param \_dstEid The endpoint ID of the destination chain.  
/// @param \_message The message to be sent.  
/// @param \_options The message execution options (e.g. gas to use on destination).  
function sendMessage(uint32 \_dstEid, string memory \_message, bytes calldata \_options) external payable {  
 bytes memory \_payload = abi.encode(\_message); // Encode the message as bytes  
 \_lzSend(  
 \_dstEid,  
 \_payload,  
 \_options,  
 MessagingFee(msg.value, 0), // Fee for the message (nativeFee, lzTokenFee)  
 payable(msg.sender) // The refund address in case the send call reverts  
 );  
}  
The sendMessage function above calls the inherited \_lzSend function, while passing in the following expected data:  
Name Type Description  
\_dstEid uint32 The endpoint ID of the destination chain to send the message to.  
\_payload bytes The message (encoded) to send.  
\_options bytes Additional options when sending the message, such as how much gas should be used when receiving the message.  
\_fee MessagingFee The calculated fee for sending the message.  
\_refundAddress address The address that will receive any excess fee values sent to the endpoint in case the \_lzSend execution reverts.  
Implementing gas fee estimation (\_quote)  
As shown in the table provided in the last section, the \_lzSend function expects an estimated gas fee to be provided when sending a message (\_fee).  
Therefore, sending a message using the sendMessage function of your contract, you first need to estimate the associated gas fees.  
There are multiple fees incurred when sending a message across chains using LayerZero, including: paying for gas on the source chain, fees paid to DVNs validating the message, and gas on the destination chain. Luckily, LayerZero bundles all of these fees together into a single fee to be paid by the msg.sender, and LayerZero Endpoints expose a \_quote function to estimate this fee.  
Add a new function to your ExampleContract contract called estimateFee that calls the \_quote function, as shown below:  
/// @notice Estimates the gas associated with sending a message.  
/// @param \_dstEid The endpoint ID of the destination chain.  
/// @param \_message The message to be sent.  
/// @param \_options The message execution options (e.g. gas to use on destination).  
/// @return nativeFee Estimated gas fee in native gas.  
/// @return lzTokenFee Estimated gas fee in ZRO token.  
function estimateFee(  
 uint32 \_dstEid,  
 string memory \_message,  
 bytes calldata \_options  
) public view returns (uint256 nativeFee, uint256 lzTokenFee) {  
 bytes memory \_payload = abi.encode(\_message);  
 MessagingFee memory fee = \_quote(\_dstEid, \_payload, \_options, false);  
 return (fee.nativeFee, fee.lzTokenFee);  
}  
The estimateFee function above calls the inherited \_quote function, while passing in the following expected data:  
Name Type Description  
\_dstEid uint32 The endpoint ID of the destination chain the message will be sent to.  
\_payload bytes The message (encoded) that will be sent.  
\_options bytes Additional options when sending the message, such as how much gas should be used when receiving the message.  
\_payInLzToken bool Boolean flag for which token to use when returning the fee (native or ZRO token).  
INFO  
Your contract’s estimateFee function should always be called immediately before calling sendMessage to accurately estimate associated gas fees.  
Implementing message receiving (\_lzReceive)  
To receive messages on the destination chain, your smart contract must override the \_lzReceive function inherited from the OApp contract.  
Add the following code snippet to your ExampleContract contract to override the \_lzReceive function:  
/// @notice Entry point for receiving messages.  
/// @param \_origin The origin information containing the source endpoint and sender address.  
/// - srcEid: The source chain endpoint ID.  
/// - sender: The sender address on the src chain.  
/// - nonce: The nonce of the message.  
/// @param \_guid The unique identifier for the received LayerZero message.  
/// @param \_message The payload of the received message.  
/// @param \_executor The address of the executor for the received message.  
/// @param \_extraData Additional arbitrary data provided by the corresponding executor.  
function \_lzReceive(  
 Origin calldata \_origin,  
 bytes32 \_guid,  
 bytes calldata payload,  
 address \_executor,  
 bytes calldata \_extraData  
 ) internal override {  
 data = abi.decode(payload, (string));  
 // other logic  
}  
The overridden \_lzReceive function receives the following arguments when receiving a message:  
Name Type Description  
\_origin Origin The origin information containing the source endpoint and sender address.  
\_guid bytes32 The unique identifier for the received LayerZero message.  
payload bytes The payload of the received message (encoded).  
\_executor address The address of the Executor for the received message.  
\_extraData bytes Additional arbitrary data provided by the corresponding Executor.  
Note that the overridden method decodes the message payload, and stores the string into a variable named data that you can read from later to fetch the latest message.  
Add the data field as a member variable to your contract:  
contract ExampleContract is OApp {  
  
 string public data;  
  
 constructor(address \_endpoint) OApp(\_endpoint, msg.sender) {}  
}  
INFO  
Overriding the \_lzReceive function allows you to provide any custom logic you wish when receiving messages, including making a call back to the source chain by invoking \_lzSend. Visit the LayerZero Message Design Patterns for common messaging flows.  
Final code  
Once you complete all of the steps above, your contract should look like this:  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
import { OApp, Origin, MessagingFee } from "@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/OApp.sol";  
  
contract ExampleContract is OApp {  
  
 string public data;  
  
 constructor(address \_endpoint) OApp(\_endpoint, msg.sender) {}  
  
 /// @notice Sends a message from the source chain to the destination chain.  
 /// @param \_dstEid The endpoint ID of the destination chain.  
 /// @param \_message The message to be sent.  
 /// @param \_options The message execution options (e.g. gas to use on destination).  
 function sendMessage(uint32 \_dstEid, string memory \_message, bytes calldata \_options) external payable {  
 bytes memory \_payload = abi.encode(\_message); // Encode the message as bytes  
 \_lzSend(  
 \_dstEid,  
 \_payload,  
 \_options,  
 MessagingFee(msg.value, 0), // Fee for the message (nativeFee, lzTokenFee)  
 payable(msg.sender) // The refund address in case the send call reverts  
 );  
 }  
  
 /// @notice Estimates the gas associated with sending a message.  
 /// @param \_dstEid The endpoint ID of the destination chain.  
 /// @param \_message The message to be sent.  
 /// @param \_options The message execution options (e.g. gas to use on destination).  
 /// @return nativeFee Estimated gas fee in native gas.  
 /// @return lzTokenFee Estimated gas fee in ZRO token.  
 function estimateFee(  
 uint32 \_dstEid,  
 string memory \_message,  
 bytes calldata \_options  
 ) public view returns (uint256 nativeFee, uint256 lzTokenFee) {  
 bytes memory \_payload = abi.encode(\_message);  
 MessagingFee memory fee = \_quote(\_dstEid, \_payload, \_options, false);  
 return (fee.nativeFee, fee.lzTokenFee);  
 }  
  
 /// @notice Entry point for receiving messages.  
 /// @param \_origin The origin information containing the source endpoint and sender address.  
 /// - srcEid: The source chain endpoint ID.  
 /// - sender: The sender address on the src chain.  
 /// - nonce: The nonce of the message.  
 /// @param \_guid The unique identifier for the received LayerZero message.  
 /// @param \_message The payload of the received message.  
 /// @param \_executor The address of the executor for the received message.  
 /// @param \_extraData Additional arbitrary data provided by the corresponding executor.  
 function \_lzReceive(  
 Origin calldata \_origin,  
 bytes32 \_guid,  
 bytes calldata payload,  
 address \_executor,  
 bytes calldata \_extraData  
 ) internal override {  
 data = abi.decode(payload, (string));  
 }  
}  
Compiling the smart contract  
Compile the smart contract to ensure it builds without any errors.  
To compile your smart contract, run:  
forge build  
Deploying the smart contract  
Setting up your wallet as the deployer  
Before you can deploy your smart contract to various chains you will need to set up a wallet to be used as the deployer.  
To do so, you can use the cast wallet import command to import the private key of the wallet into Foundry's securely encrypted keystore:  
cast wallet import deployer --interactive  
After running the command above, you will be prompted to enter your private key, as well as a password for signing transactions.  
CAUTION  
For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation. It is critical that you do NOT commit this to a public repo.  
To confirm that the wallet was imported as the deployer account in your Foundry project, run:  
cast wallet list  
Setting up environment variables  
To setup your environment, create an .env file in the home directory of your project, and add the RPC URLs and LayerZero Endpoint information for both Base Goerli and Optimism Goerli testnets:  
BASE\_GOERLI\_RPC="https://goerli.base.org"  
BASE\_GOERLI\_LZ\_ENDPOINT=0x464570adA09869d8741132183721B4f0769a0287  
BASE\_GOERLI\_LZ\_ENDPOINT\_ID=40184  
  
OPTIMISM\_GOERLI\_RPC="https://goerli.optimism.io"  
OPTIMISM\_GOERLI\_LZ\_ENDPOINT=0x464570adA09869d8741132183721B4f0769a0287  
OPTIMISM\_GOERLI\_LZ\_ENDPOINT\_ID=40132  
Once the .env file has been created, run the following command to load the environment variables in the current command line session:  
source .env  
With your contract compiled and environment setup, you are now ready to deploy the smart contract to different networks.  
Deploying the smart contract to Base Goerli  
To deploy a smart contract using Foundry, you can use the forge create command. The command requires you to specify the smart contract you want to deploy, an RPC URL of the network you want to deploy to, and the account you want to deploy with.  
INFO  
Your wallet must be funded with ETH on the Base Goerli and Optimism Goerli to cover the gas fees associated with the smart contract deployment. Otherwise, the deployment will fail.  
To get testnet ETH, see the prerequisites.  
To deploy the ExampleContract smart contract to the Base Goerli testnet, run the following command:  
forge create ./src/ExampleContract.sol:ExampleContract --rpc-url $BASE\_GOERLI\_RPC --constructor-args $BASE\_GOERLI\_LZ\_ENDPOINT --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet's private key.  
After running the command above, the contract will be deployed on the Base Goerli test network. You can view the deployment status and contract by using a block explorer.  
Deploying the smart contract to Optimism Goerli  
To deploy the ExampleContract smart contract to the Optimism Goerli testnet, run the following command:  
forge create ./src/ExampleContract.sol:ExampleContract --rpc-url $OPTIMISM\_GOERLI\_RPC --constructor-args $OPTIMISM\_GOERLI\_LZ\_ENDPOINT --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet's private key.  
After running the command above, the contract will be deployed on the Optimism Goerli test network. You can view the deployment status and contract by using the OP Goerli block explorer.  
Opening the messaging channels  
Once your contract has been deployed to Base Goerli and Optimism Goerli, you will need to open the messaging channels between the two contracts so that they can send and receive messages from one another. This is done by calling the setPeer function on the contract.  
The setPeer function expects the following arguments:  
Name Type Description  
\_eid uint32 The endpoint ID of the destination chain.  
\_peer bytes32 The contract address of the OApp contract on the destination chain.  
Setting the peers  
Foundry provides the cast command-line tool that can be used to interact with deployed smart contracts and call their functions.  
To set the peer of your ExampleContract contracts, you can use cast to call the setPeer function while providing the endpoint ID and address (in bytes) of the deployed contract on the respective destination chain.  
To set the peer of the Base Goerli contract to the Optimism Goerli contract, run the following command:  
cast send <BASE\_GOERLI\_CONTRACT\_ADDRESS> --rpc-url $BASE\_GOERLI\_RPC "setPeer(uint32, bytes32)" $OPTIMISM\_GOERLI\_LZ\_ENDPOINT\_ID <OPTIMISM\_GOERLI\_CONTRACT\_ADDRESS> --account deployer  
INFO  
Replace <BASE\_GOERLI\_CONTRACT\_ADDRESS> with the contract address of your deployed ExampleContract contract on Base Goerli, and<OPTIMISM\_GOERLI\_CONTRACT\_ADDRESS> with the contract address (as bytes) of your deployed ExampleContract contract on Optimism Goerli before running the provided cast command.  
To set the peer of the Optimism Goerli contract to the Base Goerli contract, run the following command:  
cast send <OPTIMISM\_GOERLI\_CONTRACT\_ADDRESS> --rpc-url $OPTIMISM\_GOERLI\_RPC "setPeer(uint32, bytes32)" $BASE\_GOERLI\_LZ\_ENDPOINT\_ID <BASE\_GOERLI\_CONTRACT\_ADDRESS> --account deployer  
INFO  
Replace <OPTIMISM\_GOERLI\_CONTRACT\_ADDRESS> with the contract address of your deployed ExampleContract contract on Optimism Goerli, and<BASE\_GOERLI\_CONTRACT\_ADDRESS> with the contract address (as bytes) of your deployed ExampleContract contract on Base Goerli before running the provided cast command.  
Sending messages  
Once peers have been set on each contract, they are now able to send and receive messages from one another.  
Sending a message using the newly created ExampleContract contract can be done in three steps:  
Build message options to specify logic associated with the message transaction  
Call the estimateFee function to estimate the gas fee for sending a message  
Call the sendMessage function to send a message  
Building message options  
The estimateFee and sendMessage custom functions of the ExampleContract contract both require a message options (\_options) argument to be provided.  
Message options allow you to specify arbitrary logic as part of the message transaction, such as the gas amount the Executor pays for message delivery, the order of message execution, or dropping an amount of gas to a destination address.  
LayerZero provides a Solidity library and TypeScript SDK for building these message options.  
As an example, below is a Foundry script that uses OptionsBuilder from the Solidity library to generate message options (as bytes) that set the gas amount that the Executor will pay upon message delivery to 200000 wei:  
pragma solidity ^0.8.0;  
  
import {Script, console2} from "forge-std/Script.sol";  
import { OptionsBuilder } from "@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/libs/OptionsBuilder.sol";  
  
contract OptionsScript is Script {  
 using OptionsBuilder for bytes;  
  
 function setUp() public {}  
  
 function run() public {  
 bytes memory options = OptionsBuilder.newOptions().addExecutorLzReceiveOption(200000, 0);  
 console2.logBytes(options);  
 }  
}  
The output of this script results in:  
0x00030100110100000000000000000000000000030d40  
For this tutorial, rather than building and generating your own message options, you can use the bytes output provided above.  
INFO  
Covering all of the different message options in detail is out of scope for this tutorial. If you are interested in learning more about the different message options and how to build them, visit the LayerZero developer documentation.  
Estimating the gas fee  
Before you can send a message from your contract on Base Goerli, you need to estimate the fee associated with sending the message. You can use the cast command to call the estimateFee() function of the ExampleContract contract.  
To estimate the gas fee for sending a message from Base Goerli to Optimism Goerli, run the following command:  
cast send <BASE\_GOERLI\_CONTRACT\_ADDRESS> --rpc-url $BASE\_GOERLI\_RPC "estimateFee(uint32, string, bytes)" $OPTIMISM\_GOERLI\_LZ\_ENDPOINT\_ID "Hello World" 0x00030100110100000000000000000000000000030d40 --account deployer  
INFO  
Replace <BASE\_GOERLI\_CONTRACT\_ADDRESS> with the contract address of your deployed ExampleContract contract on Base Goerli before running the provided cast command.  
The command above calls estimateFee(uint32, string, bytes, bool), while providing the required arguments, including: the endpoint ID of the destination chain, the text to send, and the message options (generated in the last section).  
Sending the message  
Once you have fetched the estimated gas for sending your message, you can now call sendMessage and provide the value returned as the msg.value.  
For example, to send a message from Base Goerli to Optimism Goerli with an estimated gas fee, run the following command:  
cast send <BASE\_GOERLI\_CONTRACT\_ADDRESS> --rpc-url $BASE\_GOERLI\_RPC --value <GAS\_ESTIMATE\_IN\_WEI> "sendMessage(uint32, string, bytes)" $OPTIMISM\_GOERLI\_LZ\_ENDPOINT\_ID "Hello World" 0x00030100110100000000000000000000000000030d40 --account deployer  
INFO  
Replace <BASE\_GOERLI\_CONTRACT\_ADDRESS> with the contract address of your deployed ExampleContract contract on Base Goerli, and <GAS\_ESTIMATE\_IN\_WEI> with the gas estimate (in wei) returned by the call to estimateFee, before running the provided cast command.  
You can view the status of your cross-chain transaction on LayerZero Scan.  
Receiving the message  
Once the message has been sent and received on the destination chain, the \_Receive function will be called on the ExampleContract and the message payload will be stored in the contract's public data variable.  
You can use the cast command to read the latest message received by the ExampleContract stored in the data variable.  
To read the latest received message data that was sent to Optimism Goerli from Base Goerli, run the following command:  
cast send <OPTIMISM\_GOERLI\_CONTRACT\_ADDRESS> --rpc-url $OPTIMISM\_GOERLI\_RPC "data" --account deployer  
The returned data should match the message text payload you sent in your message.  
You can view the status of your cross-chain transaction on LayerZero Scan.  
Conclusion  
Congratulations! You have successfully learned how to perform cross-chain messaging between Base and other chains (i.e. Optimism) using LayerZero V2.  
To learn more about cross-chain messaging and LayerZero V2, check out the following resources:  
Cross-chain  
LayerZero V2  
Tags:cross-chain  
ON THIS PAGE  
Prerequisites  
What is LayerZero?  
High-level concepts  
Coinbase Wallet  
Wallet funds  
Creating a project  
High-level concepts  
Installing the LayerZero smart contracts  
Getting started with LayerZero  
Writing the smart contract  
Compiling the smart contract  
Implementing message sending (\_lzSend)  
Implementing gas fee estimation (\_quote)  
Implementing message receiving (\_lzReceive)  
Final code  
Deploying the smart contract  
Opening the messaging channels  
Setting the peers  
Setting up environment variables  
Deploying the smart contract to Base Goerli  
Deploying the smart contract to Optimism Goerli  
Sending messages  
Building message options  
Conclusion  
Building message options  
Estimating the gas fee  
Sending the message  
Receiving the message  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/deploy-with-fleek  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Deploy an Onchain App with Fleek  
One of the "secrets" of onchain apps is that they almost always have a very large web2 component that they're dependent on. Most onchain apps rely on traditional infrastructure for their frontends, APIs, and other parts of the architecture.  
Fleek's goal is to address this issue with the Fleek Network, a fast and trustless Content Delivery Network (CDN).  
In this tutorial, you'll use Fleek to deploy a site built with the Onchain App Template.  
Objectives  
By the end of this tutorial you should be able to:  
Deploy a Next.js using the Coinbase Smart Wallet on Fleek  
Integrate with [Github] for CI/CD  
Prerequisites  
Next.js  
You should be familiar with Next.js, but do not need to be an expert. If you are comfortable with other React libraries, the pattern should be relatively easy to follow.  
Onchain Apps  
The tutorial assumes you're comfortable with the basics of deploying an app and connecting it to a smart contract. If you're still learning this part, check out our tutorials in Base Learn for [Building an Onchain App].  
Setting up the Template  
You can skip this section if you've already built an app based off the template, such as our tutorial for How to Mint on Zora with an App.  
Open Onchain App Template, click the green Use this template button, and create a new repository from the template. Clone your repo and open it in an editor.  
Install bun if you need to, and install dependencies.  
# Install bun in case you don't have it  
curl -fsSL https://bun.sh/install | bash  
  
# Install packages  
bun i  
  
# Run Next app  
bun run dev  
Navigate to localhost:3000 and make sure that it's working, then shut down the server. For this tutorial, you do not need to set any environment variables.  
Installing and Configuring Fleek  
Fleek requires static pages, so you'll need to ensure that your build process produces them. In your editor, open next.config.js and update the nextConfig.  
/\*\* @type {import('next').NextConfig} \*/  
const nextConfig = {  
 output: 'export',  
 reactStrictMode: true,  
 images: {  
 unoptimized: true,  
 },  
 trailingSlash: true,  
};  
  
module.exports = nextConfig;  
Save and close the file.  
Run bun run build and confirm that a directory called out is created.  
Navigate to Fleek's website and create an account, or log in if you already have one.  
Click into First Project. You can rename it if you want in the Settings tab.  
The best way to start is to link Fleek to your repo from the beginning. Click Add New from the upper right corner, then select Deploy My Site. Select your code location, log into your Git provider, and accept installing the Fleek app.  
You can either give it permissions for all repos, or add them one at a time.  
Select your repo, and click the Deploy button. The Configure Site window should automatically populate with the appropriate information, but just in case:  
Site Name: Your site name  
Framework: Next.js  
Branch: main  
Publish Directory: out  
Build Command: npm install && npm run build  
Click Deploy Site. Your deploy will probably fail, but this is expected!  
Return to your code editor.  
Open a terminal and install the Fleek CLI with:  
npm install -g @fleek-platform/cli  
Then, in the root of your project run:  
fleek login  
Click the link in your terminal, then click Confirm in the web page that opens up. Once your are connected, click the Visit Dashboard button. The site automatically creates a project called First Project. If you'd like, you can rename it, or add a new one.  
Each project can include more than one site.  
Return to your terminal in the app folder, and run:  
fleek sites init  
Select First Project from the list  
⚠️ Warning! To proceed, please select a project...  
  
✔ Select a project from the list: › First Project  
  
✅ Success! You have switched to project "First Project".  
For We've found existing sites. Would you like to link to one of them?, pick: Y  
Find the site you just added and select it.  
CAUTION  
You're using TypeScript, but do not select TypeScript (fleek.config.ts) in the final prompt. Select JSON (fleek.config.json).  
You'll get a few more prompts:  
? Please specify the directory containing the site files to be uploaded  
Enter out  
? Would you like to include the optional "build" command?  
Pick Y  
? Specify build command:  
Enter npm install && npm run build  
Select JSON (fleek.config.json)  
Deployment  
You can deploy the site from the CLI as the docs describe, but you do not need to. There is a better way!  
# Don't use, better method below!  
fleek sites deploy  
Instead, trigger an automatic deploy by making a change to the text at src/app/page.tsx, committing your changes, and pushing to your repo.  
Dashboard Overview and Confirming Deployment  
Return to your dashboard and click on the Sites tab. Click on the card for your new site to open it. Here, you can see information about your site in a similar presentation to other deployment providers.  
Click on the <-> Deploys tab and you'll see the automatic deploy you triggered by pushing the commit! Open your site by clicking on the build once it shifts from Pending to Live. You can then click on the link to view your site.  
Click on Settings. If you'd like, you can change the slug for your site to a name that's more related to your project.  
Conclusion  
In this tutorial, you learned how to use Fleek to deploy a Next.js site based on Onchain App Template. You also learned how to link Fleek to your Git provider to enable CI/CD.  
Tags:frontend  
ON THIS PAGE  
Prerequisites  
Setting up the Template  
Next.js  
Onchain Apps  
Installing and Configuring Fleek  
Dashboard Overview and Confirming Deployment  
Deployment  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/deploy-with-foundry  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Deploying a smart contract using Foundry  
This article will provide an overview of the Foundry development toolchain, and show you how to deploy a contract to Base Sepolia testnet.  
Foundry is a powerful suite of tools to develop, test, and debug your smart contracts. It comprises several individual tools:  
forge: the main workhorse of Foundry — for developing, testing, compiling, and deploying smart contracts  
cast: a command-line tool for performing Ethereum RPC calls (e.g., interacting with contracts, sending transactions, and getting onchain data)  
anvil: a local testnet node, for testing contract behavior from a frontend or over RPC  
chisel: a Solidity REPL, for trying out Solidity snippets on a local or forked network  
Foundry offers extremely fast feedback loops (due to the under-the-hood Rust implementation) and less context switching — because you'll be writing your contracts, tests, and deployment scripts All in Solidity!  
INFO  
For production / mainnet deployments the steps below in this tutorial will be almost identical, however, you'll want to ensure that you've configured Base (mainnet) as the network rather than Base Sepolia (testnet).  
Objectives  
By the end of this tutorial, you should be able to do the following:  
Setup Foundry for Base  
Create an NFT smart contract for Base  
Compile a smart contract for Base (using forge)  
Deploy a smart contract to Base (also with forge)  
Interact with a smart contract deployed on Base (using cast)  
Prerequisites  
Foundry  
This tutorial requires you have Foundry installed.  
From the command-line (terminal), run: curl -L https://foundry.paradigm.xyz | bash  
Then run foundryup, to install the latest (nightly) build of Foundry  
For more information, see the Foundry Book installation guide.  
Coinbase Wallet  
In order to deploy a smart contract, you will first need a web3 wallet. You can create a wallet by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
Wallet funds  
Deploying contracts to the blockchain requires a gas fee. Therefore, you will need to fund your wallet with ETH to cover those gas fees.  
For this tutorial, you will be deploying a contract to the Base Sepolia test network. You can fund your wallet with Base Sepolia ETH using one of the faucets listed on the Base Network Faucets page.  
Creating a project  
Before you can begin deploying smart contracts to Base, you need to set up your development environment by creating a Foundry project.  
To create a new Foundry project, first create a new directory:  
mkdir myproject  
Then run:  
cd myproject  
forge init  
This will create a Foundry project, which has the following basic layout:  
.  
├── foundry.toml  
├── script  
 │ └── Counter.s.sol  
├── src  
 │ └── Counter.sol  
└── test  
 └── Counter.t.sol  
Compiling the smart contract  
Below is a simple NFT smart contract (ERC-721) written in the Solidity programming language:  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.23;  
  
import "openzeppelin-contracts/contracts/token/ERC721/ERC721.sol";  
  
contract NFT is ERC721 {  
 uint256 public currentTokenId;  
  
 constructor() ERC721("NFT Name", "NFT") {}  
  
 function mint(address recipient) public payable returns (uint256) {  
 uint256 newItemId = ++currentTokenId;  
 \_safeMint(recipient, newItemId);  
 return newItemId;  
 }  
}  
The Solidity code above defines a smart contract named NFT. The code uses the ERC721 interface provided by the OpenZeppelin Contracts library to create an NFT smart contract. OpenZeppelin allows developers to leverage battle-tested smart contract implementations that adhere to official ERC standards.  
To add the OpenZeppelin Contracts library to your project, run:  
forge install openzeppelin/openzeppelin-contracts  
In your project, delete the src/Counter.sol contract that was generated with the project and add the above code in a new file called src/NFT.sol. (You can also delete the test/Counter.t.sol and script/Counter.s.sol files, but you should add your own tests ASAP!).  
To compile our basic NFT contract using Foundry, run:  
forge build  
Configuring Foundry with Base  
Next, you will configure your Foundry project to deploy smart contracts to the Base network. First you'll store your private key in an encrypted keystore, then you'll add Base as a network.  
Storing your private key  
The following command will import your private key to Foundry's secure keystore. You will be prompted to enter your private key, as well as a password for signing transactions:  
cast wallet import deployer --interactive  
CAUTION  
For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation. It is critical that you do NOT commit this to a public repo.  
Run this command to confirm that the 'deployer' account is setup in Foundry:  
cast wallet list  
Adding Base as a network  
When verifying a contract with BaseScan, you need an API key. You can get your BaseScan API key from here after you sign up for an account.  
CAUTION  
Although they're made by the same folks, Etherscan API keys will not work on BaseScan!  
Now create a .env file in the home directory of your project to add the Base network and an API key for verifying your contract on BaseScan:  
BASE\_MAINNET\_RPC="https://mainnet.base.org"  
BASE\_SEPOLIA\_RPC="https://sepolia.base.org"  
ETHERSCAN\_API\_KEY="<YOUR API KEY>"  
Note that even though you're using BaseScan as your block explorer, Foundry expects the API key to be defined as ETHERSCAN\_API\_KEY.  
Loading environment variables  
Now that you've created the above .env file, run the following command to load the environment variables in the current command line session:  
source .env  
Deploying the smart contract  
With your contract compiled and your environment configured, you are ready to deploy to the Base Sepolia test network!  
Today, you'll use the forge create command, which is a straightforward way to deploy a single contract at a time. In the future, you may want to look into forge script, which enables scripting onchain transactions and deploying more complex smart contract projects.  
You'll need testnet ETH in your wallet. See the prerequisites if you haven't done that yet. Otherwise, the deployment attempt will fail.  
To deploy the contract to the Base Sepolia test network, run the following command. You will be prompted to enter the password that you set earlier, when you imported your private key:  
forge create ./src/NFT.sol:NFT --rpc-url $BASE\_SEPOLIA\_RPC --account deployer  
The contract will be deployed on the Base Sepolia test network. You can view the deployment status and contract by using a block explorer and searching for the address returned by your deploy script. If you've deployed an exact copy of the NFT contract above, it will already be verified and you'll be able to read and write to the contract using the web interface.  
INFO  
If you'd like to deploy to mainnet, you'll modify the command like so:  
forge create ./src/NFT.sol:NFT --rpc-url $BASE\_MAINNET\_RPC --account deployer  
Regardless of the network you're deploying to, if you're deploying a new or modified contract, you'll need to verify it.  
Verifying the Smart Contract  
In web3, it's considered best practice to verify your contracts so that users and other developers can inspect the source code, and be sure that it matches the deployed bytecode on the blockchain.  
Further, if you want to allow others to interact with your contract using the block explorer, it first needs to be verified. The above contract has already been verified, so you should be able to view your version on a block explorer already, but we'll still walk through how to verify a contract on Base Sepolia testnet.  
INFO  
Remember, you need an API key from BaseScan to verify your contracts. You can get your API key from the BaseScan site after you sign up for an account.  
Grab the deployed address and run:  
forge verify-contract <DEPLOYED\_ADDRESS> ./src/NFT.sol:NFT --chain 84532 --watch  
You should see an output similar to:  
Start verifying contract `0x71bfCe1172A66c1c25A50b49156FAe45EB56E009` deployed on base-sepolia  
  
Submitting verification for [src/NFT.sol:NFT] 0x71bfCe1172A66c1c25A50b49156FAe45EB56E009.  
Submitted contract for verification:  
 Response: `OK`  
 GUID: `3i9rmtmtyyzkqpfvy7pcxj1wtgqyuybvscnq8d7ywfuskss1s7`  
 URL:  
 https://sepolia.basescan.org/address/0x71bfce1172a66c1c25a50b49156fae45eb56e009  
Contract verification status:  
Response: `NOTOK`  
Details: `Pending in queue`  
Contract verification status:  
Response: `OK`  
Details: `Pass - Verified`  
Contract successfully verified  
Search for your contract on BaseScan to confirm it is verified.  
INFO  
You can't re-verify a contract identical to one that has already been verified. If you attempt to do so, such as verifying the above contract, you'll get an error similar to:  
Start verifying contract `0x71bfCe1172A66c1c25A50b49156FAe45EB56E009` deployed on base-sepolia  
  
Contract [src/NFT.sol:NFT] "0x71bfCe1172A66c1c25A50b49156FAe45EB56E009" is already verified. Skipping verification.  
Interacting with the Smart Contract  
If you verified on BaseScan, you can use the Read Contract and Write Contract sections under the Contract tab to interact with the deployed contract. To use Write Contract, you'll need to connect your wallet first, by clicking the Connect to Web3 button (sometimes this can be a little finicky, and you'll need to click Connect twice before it shows your wallet is successfully connected).  
To practice using the cast command-line tool which Foundry provides, you'll perform a call without publishing a transaction (a read), then sign and publish a transaction (a write).  
Performing a call  
A key component of the Foundry toolkit, cast enables us to interact with contracts, send transactions, and get onchain data using Ethereum RPC calls. First you will perform a call from your account, without publishing a transaction.  
From the command-line, run:  
cast call <DEPLOYED\_ADDRESS> --rpc-url $BASE\_SEPOLIA\_RPC "balanceOf(address)" <YOUR\_ADDRESS\_HERE>  
You should receive 0x0000000000000000000000000000000000000000000000000000000000000000 in response, which equals 0 in hexadecimal. And that makes sense — while you've deployed the NFT contract, no NFTs have been minted yet and therefore your account's balance is zero.  
Signing and publishing a transaction  
Now, sign and publish a transaction, calling the mint(address) function on the NFT contract you just deployed.  
Run the following command:  
cast send <DEPLOYED\_ADDRESS> --rpc-url=$BASE\_SEPOLIA\_RPC "mint(address)" <YOUR\_ADDRESS\_HERE> --account deployer  
INFO  
Note that in this cast send command, you had to include your private key, but this is not required for cast call, because that's for calling view-only contract functions and therefore you don't need to sign anything.  
If successful, Foundry will respond with information about the transaction, including the blockNumber, gasUsed, and transactionHash.  
Finally, let's confirm that you did indeed mint yourself one NFT. If you run the first cast call command again, you should see that your balance increased from 0 to 1:  
cast call <DEPLOYED\_ADDRESS> --rpc-url $BASE\_SEPOLIA\_RPC "balanceOf(address)" <YOUR\_ADDRESS\_HERE>  
And the response: 0x0000000000000000000000000000000000000000000000000000000000000001 (1 in hex) — congratulations, you deployed a contract and minted an NFT with Foundry!  
Conclusion  
Phew, that was a lot! You learned how to setup a project, deploy to Base, and interact with our smart contract using Foundry. The process is the same for real networks, just more expensive — and of course, you'll want to invest time and effort testing your contracts, to reduce the likelihood of user-impacting bugs before deploying.  
For all things Foundry, check out the Foundry book, or head to the official Telegram dev chat or support chat.  
Tags:smart contracts  
ON THIS PAGE  
Prerequisites  
Creating a project  
Foundry  
Coinbase Wallet  
Wallet funds  
Compiling the smart contract  
Configuring Foundry with Base  
Deploying the smart contract  
Storing your private key  
Adding Base as a network  
Loading environment variables  
Verifying the Smart Contract  
Interacting with the Smart Contract  
Conclusion  
Performing a call  
Signing and publishing a transaction  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/deploy-with-hardhat  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Deploying a smart contract using Hardhat  
This section will guide you through deploying an NFT smart contract (ERC-721) on the Base test network using Hardhat.  
Hardhat is a developer tool that provides a simple way to deploy, test, and debug smart contracts.  
Objectives  
By the end of this tutorial, you should be able to do the following:  
Setup Hardhat for Base  
Create an NFT smart contract for Base  
Compile a smart contract for Base  
Deploy a smart contract to Base  
Interact with a smart contract deployed on Base  
Prerequisites  
Node v18+  
This tutorial requires you have Node version 18+ installed.  
Download Node v18+  
If you are using nvm to manage your node versions, you can just run nvm install 18.  
Coinbase Wallet  
In order to deploy a smart contract, you will first need a web3 wallet. You can create a wallet by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
Wallet funds  
Deploying contracts to the blockchain requires a gas fee. Therefore, you will need to fund your wallet with ETH to cover those gas fees.  
For this tutorial, you will be deploying a contract to the Base Sepolia test network. You can fund your wallet with Base Sepolia ETH using one of the faucets listed on the Base Network Faucets page.  
Creating a project  
Before you can begin deploying smart contracts to Base, you need to set up your development environment by creating a Node.js project.  
To create a new Node.js project, run:  
npm init --y  
Next, you will need to install Hardhat and create a new Hardhat project  
To install Hardhat, run:  
npm install --save-dev hardhat  
To create a new Hardhat project, run:  
npx hardhat init  
Select Create a TypeScript project then press enter to confirm the project root.  
Select y for both adding a .gitignore and loading the sample project. It will take a moment for the project setup process to complete.  
Configuring Hardhat with Base  
In order to deploy smart contracts to the Base network, you will need to configure your Hardhat project and add the Base network.  
To configure Hardhat to use Base, add Base as a network to your project's hardhat.config.ts file:  
import { HardhatUserConfig } from 'hardhat/config';  
import '@nomicfoundation/hardhat-toolbox';  
  
require('dotenv').config();  
  
const config: HardhatUserConfig = {  
 solidity: {  
 version: '0.8.23',  
 },  
 networks: {  
 // for mainnet  
 'base-mainnet': {  
 url: 'https://mainnet.base.org',  
 accounts: [process.env.WALLET\_KEY as string],  
 gasPrice: 1000000000,  
 },  
 // for testnet  
 'base-sepolia': {  
 url: 'https://sepolia.base.org',  
 accounts: [process.env.WALLET\_KEY as string],  
 gasPrice: 1000000000,  
 },  
 // for local dev environment  
 'base-local': {  
 url: 'http://localhost:8545',  
 accounts: [process.env.WALLET\_KEY as string],  
 gasPrice: 1000000000,  
 },  
 },  
 defaultNetwork: 'hardhat',  
};  
  
export default config;  
Install Hardhat toolbox  
The above configuration uses the @nomicfoundation/hardhat-toolbox plugin to bundle all the commonly used packages and Hardhat plugins recommended to start developing with Hardhat.  
To install @nomicfoundation/hardhat-toolbox, run:  
npm install --save-dev @nomicfoundation/hardhat-toolbox  
Loading environment variables  
The above configuration also uses dotenv to load the WALLET\_KEY environment variable from a .env file to process.env.WALLET\_KEY. You should use a similar method to avoid hardcoding your private keys within your source code.  
To install dotenv, run:  
npm install --save-dev dotenv  
Once you have dotenv installed, you can create a .env file with the following content:  
WALLET\_KEY="<YOUR\_PRIVATE\_KEY>"  
Substituting <YOUR\_PRIVATE\_KEY> with the private key for your wallet.  
CAUTION  
WALLET\_KEY is the private key of the wallet to use when deploying a contract. For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation. It is critical that you do NOT commit this to a public repo  
Local Networks  
You can run the Base network locally, and deploy using it. If this is what you are looking to do, see the repo containing the relevant Docker builds.  
It will take a very long time for your node to sync with the network. If you get errors that the nonce has already been used when trying to deploy, you aren't synced yet.  
For quick testing, such as if you want to add unit tests to the below NFT contract, you may wish to leave the defaultNetwork as 'hardhat'.  
Compiling the smart contract  
Below is a simple NFT smart contract (ERC-721) written in the Solidity programming language:  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.23;  
  
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";  
  
contract NFT is ERC721 {  
 uint256 public currentTokenId;  
  
 constructor() ERC721("NFT Name", "NFT") {}  
  
 function mint(address recipient) public payable returns (uint256) {  
 uint256 newItemId = ++currentTokenId;  
 \_safeMint(recipient, newItemId);  
 return newItemId;  
 }  
}  
The Solidity code above defines a smart contract named NFT. The code uses the ERC721 interface provided by the OpenZeppelin Contracts library to create an NFT smart contract. OpenZeppelin allows developers to leverage battle-tested smart contract implementations that adhere to official ERC standards.  
To add the OpenZeppelin Contracts library to your project, run:  
npm install --save @openzeppelin/contracts  
In your project, delete the contracts/Lock.sol contract that was generated with the project and add the above code in a new file called contracts/NFT.sol. (You can also delete the test/Lock.ts test file, but you should add your own tests ASAP!).  
To compile the contract using Hardhat, run:  
npx hardhat compile  
Deploying the smart contract  
Once your contract has been successfully compiled, you can deploy the contract to the Base Sepolia test network.  
To deploy the contract to the Base Sepolia test network, you'll need to modify the scripts/deploy.ts in your project:  
import { ethers } from 'hardhat';  
  
async function main() {  
 const nft = await ethers.deployContract('NFT');  
  
 await nft.waitForDeployment();  
  
 console.log('NFT Contract Deployed at ' + nft.target);  
}  
  
// We recommend this pattern to be able to use async/await everywhere  
// and properly handle errors.  
main().catch((error) => {  
 console.error(error);  
 process.exitCode = 1;  
});  
You'll also need testnet ETH in your wallet. See the prerequisites if you haven't done that yet. Otherwise, the deployment attempt will fail.  
Finally, run:  
npx hardhat run scripts/deploy.ts --network base-sepolia  
The contract will be deployed on the Base Sepolia test network. You can view the deployment status and contract by using a block explorer and searching for the address returned by your deploy script. If you've deployed an exact copy of the NFT contract above, it will already be verified and you'll be able to read and write to the contract using the web interface.  
INFO  
If you'd like to deploy to mainnet, you'll modify the command like so:  
npx hardhat run scripts/deploy.ts --network base-mainnet  
Regardless of the network you're deploying to, if you're deploying a new or modified contract, you'll need to verify it first.  
Verifying the Smart Contract  
If you want to interact with your contract on the block explorer, you, or someone, needs to verify it first. The above contract has already been verified, so you should be able to view your version on a block explorer already. For the remainder of this tutorial, we'll walk through how to verify your contract on Base Sepolia testnet.  
In hardhat.config.ts, configure Base Sepolia as a custom network. Add the following to your HardhatUserConfig:  
Basescan  
Blockscout  
etherscan: {  
 apiKey: {  
 "base-sepolia": "PLACEHOLDER\_STRING"  
 },  
 customChains: [  
 {  
 network: "base-sepolia",  
 chainId: 84532,  
 urls: {  
 apiURL: "https://api-sepolia.basescan.org/api",  
 browserURL: "https://sepolia.basescan.org"  
 }  
 }  
 ]  
 },  
INFO  
When verifying a contract with Basescan on testnet (Sepolia), an API key is not required. You can leave the value as PLACEHOLDER\_STRING. On mainnet, you can get your Basescan API key from here after you sign up for an account.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Now, you can verify your contract. Grab the deployed address and run:  
npx hardhat verify --network base-sepolia <deployed address>  
You should see an output similar to:  
Basescan  
Blockscout  
Nothing to compile  
No need to generate any newer typings.  
Successfully submitted source code for contract  
contracts/NFT.sol:NFT at 0x6527E5052de5521fE370AE5ec0aFCC6cD5a221de  
for verification on the block explorer. Waiting for verification result...  
  
Successfully verified contract NFT on Etherscan.  
  
  
  
  
  
  
  
INFO  
You can't re-verify a contract identical to one that has already been verified. If you attempt to do so, such as verifying the above contract, you'll get an error similar to:  
Error in plugin @nomiclabs/hardhat-etherscan: The API responded with an unexpected message.  
Contract verification may have succeeded and should be checked manually.  
Message: Already Verified  
Search for your contract on Blockscout or Basescan to confirm it is verified.  
Interacting with the Smart Contract  
If you verified on Basescan, you can use the Read Contract and Write Contract tabs to interact with the deployed contract. You'll need to connect your wallet first, by clicking the Connect button.  
Tags:smart contracts  
ON THIS PAGE  
Prerequisites  
Creating a project  
Node v18+  
Coinbase Wallet  
Wallet funds  
Configuring Hardhat with Base  
Compiling the smart contract  
Install Hardhat toolbox  
Loading environment variables  
Local Networks  
Deploying the smart contract  
Verifying the Smart Contract  
Interacting with the Smart Contract  
Interacting with the Smart Contract  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/deploy-with-remix  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Deploying a smart contract using Remix  
Remix is an online IDE that you can use to rapidly develop and deploy smart contracts. If you're new to smart contracts, it's a great tool that lets you jump right in without needing to configure a local editor or struggle through environment configuration issues before getting started.  
Remix contains a simulation of a blockchain that you can use to rapidly deploy and test your contracts. This simulation only exists within your browser, so you can't share it with others or use external tools or a front end to interact with it. However, you can also deploy to a variety of testnets from within Remix. Doing so will allow you to share your contract with others, at the cost of making it public.  
In this article, we'll give you an overview of Remix, and show you how to deploy a contract to Base Sepolia testnet.  
INFO  
For production / mainnet deployments the steps below in this tutorial will be almost identical, however, you'll want to ensure that you've selected Base (mainnet) as the network rather than Base Sepolia (testnet).  
If you're already familiar with Remix, you probably want to jump down to here.  
Objectives  
By the end of this tutorial, you should be able to:  
List the features, pros, and cons of using Remix as an IDE  
Deploy and test the Storage.sol demo contract in Remix  
Use Remix to deploy a contract to the Base Sepolia testnet and interact with it in BaseScan  
Remix Window Overview  
Begin by opening a browser window and navigating to [remix.ethereum.org]. Click through the introductory tips, then review the editor. It is divided into three areas, which should look familiar.  
Editor Pane  
The editor pane loads with the Remix home screen, which contains news, helpful links, and warnings about common scams. You can close the home tab if you'd like, then open 1\_Storage.sol, located inside the contracts folder of the default\_workspace.  
You'll edit your code in the editor pane. It also has most of the features you're expecting, such as syntax and error highlighting. Note that in Remix, errors are not underlined. Instead, you'll see an❗to the left of the line number where the error is present.  
At the top, you'll see a big green arrow similar to the Run button in other editors. In Solidity, this compiles your code, but it does not run it because you must first deploy your code to the simulated blockchain.  
Terminal/Output  
Below the editor pane, you'll find the terminal.  
You'll primarily use this panel to observe transaction logs from your smart contracts. It's also one way to access Remix's powerful debugging tools.  
Left Panel  
As with many other editors, the left panel in Remix has a number of vertical tabs that allow you to switch between different tools and functions. You can explore the files in your current workspace, create and switch between workspaces, search your code, and access a number of plugins.  
Plugins  
Most of the features in Remix are plugins, and the ones you'll use the most are active by default. You can view and manage plugins by clicking the plug button in the lower-left corner, right above the settings gear. You can turn them off and on by clicking activate/deactivate, and some, such as the Debug plugin will be automatically activated through other parts of the editor.  
Solidity Compiler  
The first default plugin (after the search function) is the Solidity Compiler. Be sure to check the Auto compile option. Smart contracts are almost always in very small files, so this shouldn't ever cause a performance problem while editing code.  
The Compile and Run script button in this plugin is a little misleading. This is not how you will usually run your contract through testing. You can click the I button for more information on this feature.  
Finally, if you have errors in your contracts, the complete text for each error will appear at the bottom of the page. Try it out by introducing some typos to 1\_Storage.sol.  
Deploy & Run Transactions  
The Deploy & Run Transactions plugin is what you'll use to deploy your contracts and then interact with them. At the top are controls to select which virtual machine to use, mock user wallets with test Ether, and a drop down menu to select the contract you wish to deploy and test.  
Fix any errors you introduced to 1\_Storage.sol and click the orange Deploy button. You'll see your contract appear below as STORAGE AT \<address>.  
CAUTION  
There are a couple gotchas that can be very confusing with deploying contracts in Remix.  
First, every time you hit the Deploy button, a new copy of your contract is deployed, but the previous deployments remain. Unless you are comparing or debugging between different versions of a contract, or deploying multiple contracts at once, you should click the Trash button to erase old deployments before deploying again.  
Second, if your code will not compile, clicking the deploy button will not generate an error! Instead, the last compiled version will be deployed. Visually check and confirm that there are no errors indicated by a number in a red circle on top of the Compiler plugin.  
Prepare for Deployment  
Testnets operate in a similar, but not exactly the same manner as the main networks they shadow. You need a wallet with the appropriate token to interact with them by deploying a new contract or calling functions in a deployed contract.  
Set Up a Wallet  
If you already have a wallet set up exclusively for development, you can skip to the next section. Otherwise, now is the time to jump in!  
DANGER  
It is very dangerous to use a wallet with valuable assets for development. You could easily write code with a bug that transfers the wrong amount of the wrong token to the wrong address. Transactions cannot be reversed once sent!  
Be safe and use separate wallets for separate purposes.  
First, add the Coinbase or MetaMask wallet to your browser, and set up a new wallet. As a developer, you need to be doubly careful about the security of your wallet! Many apps grant special powers to the wallet address that is the owner of the contract, such as allowing the withdrawal of all the Ether that customers have paid to the contract, or changing critical settings.  
Once you've completed wallet setup, enable developer settings and turn on testnets (Coinbase Settings, [MetaMask Settings]).  
Add the Base Sepolia Network to your Wallet  
Most wallets will already have the Base Sepolia network as one of the testnet networks. You may need to turn on developer mode to see them.  
For this tutorial, you will be deploying a contract to the Base Sepolia test network. You can fund your wallet with Base Sepolia ETH using one of the faucets listed on the Base Network Faucets page.  
Get Testnet Ether  
Testnet tokens have no real value, but the supply is not unlimited. You can use a faucet to get a small amount of Sepolia Ether to pay gas fees for testing. Most faucets allow you to ask for a small amount each day, and some won't send you more if your balance is too high.  
You can find many faucets by searching, and it's good to keep a few bookmarked because they have a tendency to go down from time to time. Faucet providers are constantly combating bad actors and sometimes need to disable their faucets while doing so.  
The Coinbase Wallet has faucets built in. Find them by clicking Settings -> Developer settings -> Testnet faucets.  
You can also access the faucets on the web.  
Once you have testnet Base Sepolia Ether, you can view your balance under the Testnets tab in the Coinbase wallet, or by selecting the testnet from the network dropdown in MetaMask. Sadly, it's not actually worth the amount listed!  
Deploying to Testnet  
Once you have testnet Ether, you can deploy the Storage contract!  
Selecting the Environment  
Open the Deploy & Run Transactions tab. Under Environment, select Injected Provider. It will list Coinbase, MetaMask, or any other wallet you have activated here.  
If that option is not available, you can add it by choosing Customize this list...  
The first time you do this, your wallet will ask you to confirm that you want to connect this app (Remix) to your wallet.  
Once you are connected, you'll see the name of the network below the Environment dropdown.  
For Base Sepolia, you should see Custom (84532) network (note: if you're deploying to mainnet rather than testnet, you should see Custom (8453) network instead).  
If you don't see the correct network, change the active network in your wallet.  
Deploy the Contract  
Click the orange Deploy button. Because it costs gas to deploy a contract, you'll be asked to review and confirm a transaction.  
DANGER  
Always carefully review all transactions, confirming the transaction cost, assets transferred, and network. As a developer, you'll get used to approving transactions regularly. Do the best you can to avoid getting into the habit of clicking Confirm without reviewing the transaction carefully. If you feel pressured to Confirm before you run out of time, it is almost certainly a scam.  
After you click the Confirm button, return to Remix and wait for the transaction to deploy. Copy its address and navigate to sepolia.basescan.org. Note: If you deployed to mainnet, you'll navigate to basescan.org instead.  
Verify the Contract  
INFO  
You don't need to verify the contract if you've deployed one identical to a contract that has already been verified.  
You can interact with your deployed contract using Remix, the same as before, but it's also possible to interact with it through BaseScan. Paste your address in the search field to find it.  
On this page, you can review the balance, information about, and all the transactions that have ever occurred with your contract.  
Click the Contract tab in the main panel. If you've deployed a unique contract, at the top is a message asking you to Verify and Publish your contract source code.  
Verifying your contract maps the names of your functions and variables to the compiled byte code, which makes it possible to interact with the contract using a human-readable interface.  
Click the link. Your contract address is already entered.  
Under Please select Compiler Type choose \_Solidity (Single file)  
For Please Select Compiler Version select the version matching the pragma at the top of your source file. Our examples are currently using v0.8.17+commit.8df45f5f.  
For Please select Open Source License Type pick the license that matches what you selected for your contract as the SPDX-License-Identifier. Pick None if you followed the Solidity-recommended practice of using UNLICENSED.  
On the next page, copy and paste your source code in the window. Verify that you are not a robot, and click Verify and Publish. You should see a success message.  
Click the linked address to your contract to return to the contract page. You'll now see your code!  
Interact with the Contract  
You can now interact with your contract using BaseScan. Click the Read Contract button. Both of your functions will be listed here and can be tested using the web interface.  
You won't have anything under Write Contract because this contract doesn't have any functions that save data to state.  
Conclusion  
You now have the power to put smart contracts on the blockchain! You've only deployed to a test network, but the process for real networks is exactly the same - just more expensive!  
Tags:smart contracts  
ON THIS PAGE  
Remix Window Overview  
Plugins  
Solidity Compiler  
Deploy & Run Transactions  
Left Panel  
Prepare for Deployment  
Set Up a Wallet  
Add the Base Sepolia Network to your Wallet  
Deploying to Testnet  
Selecting the Environment  
Deploy the Contract  
Verify the Contract  
Conclusion  
Selecting the Environment  
Deploy the Contract  
Verify the Contract  
Interact with the Contract  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/deploy-with-tenderly  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Deploying a smart contract using Tenderly  
This tutorial shows you how to set up a private environment for smart contract development on Base Network using Tenderly DevNets.  
Learn how to configure a DevNet, use built-in tools for smart contract debugging, run transaction simulations to validate your fixes, and introduce a DevNet into your continuous integration.  
Tenderly DevNets are a zero-setup environment for developing and testing smart contracts against production data. As a private replica of 30+ EVM networks, a DevNet environment allows you to instantly deploy, execute, and debug smart contracts. With an unlimited faucet, unlocked public accounts, and built-in debugging tools, you get full control over your environment while getting the latest network states. Additionally, you can integrate a DevNet RPC URL into your existing development and CI flows, automating setups with reusable and customizable YAML templates.  
Objectives  
By the end of this tutorial, you should be able to:  
Create a DevNet with the most recent Base network states.  
Deploy, verify, and execute your contracts on a DevNet.  
Debug smart contracts instantly and optimize their gas usage.  
Use transaction simulations to validate fixes prior to deployment.  
Integrate a DevNet into your Continuous Integration (CI) to automate your testing.  
Prerequisites  
1. Set up a Tenderly account  
To begin developing on Tenderly, you must first set up an account. You can easily create one by visiting the registration page.  
2. Install the Tenderly CLI  
After setting up your account, you should install the Tenderly Command Line Interface (CLI). This will allow you to interact with your DevNet environments directly from your terminal.  
Follow these instructions to install the Tenderly CLI.  
3. Create a DevNet template  
Before developing a smart contract, you must create a DevNet template via Tenderly.  
Follow these steps to set up your DevNet template for Base Network:  
Visit the Tenderly Dashboard.  
From the lefthand menu, navigate to DevNets and click Create Template.  
When prompted:  
Choose Base for your Network.  
Give your Base DevNet a unique Name.  
Click Create.  
INFO  
If needed, check out the Tenderly documentation for more information on setting up a Devnet template.  
Click Spawn DevNet, and that's about it - you've got your own private replica of Base Network.  
INFO  
If needed, check out the Tenderly documentation for more alternative methods to spawn a DevNet.  
4. Customize your DevNet environment  
You can also use the YAML editor in DevNets to create reusable templates that allow you to configure your environment. As a "blueprint" for your DevNets, the reusable YAML templates enable you to quickly spawn your DevNets with preset configurations. You can use them to override on-chain data for wallets and contracts of interest with custom values, including balances, erc20, storage, wallets, and contracts.  
YAML Template Configuration  
# Learn how to configure DevNet templates using YAML here: https://docs.tenderly.co/devnets/yaml-template  
  
version: v0  
template:  
 name: greeter  
 block-number: latest  
 visibility: TEAM  
 network-id: 8453  
 execution:  
 chain-config:  
 chain-id: 8453  
 block-gas-limit: 10000000  
 base-fee-per-gas: 1000000000  
  
 # contracts:  
 # - address: 0x395eD9ffd32b255dBD128092ABa40200159d664b  
 # bytecode: 0x123412341234123124123143  
 # balance: 100  
 # slots:  
 # - 0x3459054d09ae8631455b798b2b5d106e17bb4e68a39d2d2a935f5f1b7253988c: 0x0000000000000000000000000000000000000000000000000000000000000000  
 # - 0x3459054d09ae8631455b798b2b5d106e17bb4e68a39d2d2a935f5f1b7253988d: 0x0000000000000000000000000000000000000000000000000000000000000001  
 # wallets:  
 # - address: 0x395eD9ffd32b255dBD128092ABa40200159d664b  
 # balance: 100  
 # - private-key: 0x3459054d09ae8631455b798b2b5d106e17bb4e68a39d2d2a935f5f1b7253988c  
 # balance: 100  
 # storage:  
 # - address: 0x395eD9ffd32b255dBD128092ABa40200159d664b  
 # slots:  
 # - 0x1459054d09ae8631455b798b2b5d106e17bb4e68a39d2d2a935f5f1b7253988c: 0x0000000000000000000000000000000000000000000000000000000000000001  
 # - 0x2459054d09ae8631455b798b2b5d106e17bb4e68a39d2d2a935f5f1b7253988c: 0x0000000000000000000000000000000000000000000000000000000000000002  
 # - address: 0xabcde9ffd32b255dBD128092ABa40200159d665a  
 # slots:  
 # - 0x3459054d09ae8631455b798b2b5d106e17bb4e68a39d2d2a935f5f1b7253988c: 0x0000000000000000000000000000000000000000000000000000000000000003  
 # balances:  
 # - address: 0x395eD9ffd32b255dBD128092ABa40200159d664b  
 # amount: 1000  
 # - address: 0x395eD9ffd32b255dBD128092ABa40200159d664b  
 # amount: 1000  
 # erc20:  
 # - contract: 0x6B175474E89094C44Da98b954EedeAC495271d0F # DAI  
 # balances:  
 # - address: 0x0000000000000000000000000000000000000000  
 # amount: 100  
 display-name: Greeter  
5. Set up Hardhat and add a DevNet RPC  
Next, let's extend your HardHat environment by connecting it to a Tenderly DevNet. All transactions from your scripts and tests will be recorded for later viewing in the Tenderly DevNet Dashboard.  
First, you must modify a code example to use the new DevNet you created.  
The rest of this tutorial assumes you're either using the devnet-examples repository or that you have a Hardhat project set up with these dependencies included:  
"@nomicfoundation/hardhat-toolbox": "^2.0.2"  
"@nomiclabs/hardhat-ethers": "^2.2.3"  
"@tenderly/hardhat-tenderly": "^1.7.7"  
"ethers": "^5.7.2"  
"hardhat": "^2.17.1"  
First, download the example code from the Tenderly Github repo:  
git clone https://github.com/Tenderly/devnet-examples.git  
Navigate to the local-development directory (make sure @tenderly/hardhat-tenderly is set to: "^1.7.7") and execute the following:  
cd local-development  
yarn install  
Modify your hardhat.config.ts file by adding the items below:  
import {HardhatUserConfig} from "hardhat/config";  
import "@nomicfoundation/hardhat-toolbox";  
  
import \* as tdly from "@tenderly/hardhat-tenderly";  
import \* as dotenv from "dotenv";  
dotenv.config();  
  
tdly.setup({  
 \*\*// TODO: Prefer manual over automatic verification\*\*  
 automaticVerifications: false,  
});  
  
const config: HardhatUserConfig = {  
 solidity: "0.8.18",  
 defaultNetwork: "tenderly",  
 networks: {  
 \*\*// TODO: Make sure this is named "tenderly" as it is here.\*\*  
 tenderly: {  
 \*\*// TODO: Add your Base DevNet RPC URL here (created during the spawn step)\*\*  
 url: 'PASTE RPC LINK HERE',  
 chainId: 8453 // (Base ChainID)  
 }  
 },  
 tenderly: {  
 \*\*// TODO: Add your tenderly username (from the Dashboard)\*\*  
 username: "",  
 \*\*// TODO: Add your project name (from the Dashboard)\*\*  
 project: "",  
 privateVerification: false  
 }  
};  
  
export default config;  
Verify your smart contract  
You can verify your smart contract on Tenderly using several methods. In this tutorial, we use the Tenderly Hardhat plugin method, but feel free to choose a different one if it better suits your project.  
In deployGreeter.ts, add the following code after the greeter.setGreeting().  
await greeter.deployed();  
await greeter.setGreeting('hello');  
  
// Add this  
await tenderly.verify({  
 name: 'Greeter',  
 address: greeter.address,  
});  
Also, make sure to update the Tenderly import at the top of the file as well.  
import { ethers, \*\*tenderly\*\* } from "hardhat"; // Add tenderly  
Deploy your smart contract:  
npx hardhat run scripts/deployGreeter.ts --network tenderly  
If successful, you should get an output similar to:  
You can see the result of the deployment in this example on a publicly shared DevNet.  
Test your smart contract to make sure there are no errors (It is always best practice to run the included tests).  
Use Tenderly Debugger  
Tenderly Debugger allows you to pinpoint the exact line of code causing an issue and speed up your debugging process. With Debugger, you can step through code and inspect stack traces, filter internal and external calls, see decoded events and logs, examine state changes, and more.  
To use it, follow these steps:  
Open the Transaction tab in your DevNet.  
Click on the setGreeting transaction.  
Click Debugger.  
Click the Debugger button.  
Notice the current line (line 14 in this example) is highlighted. Use Debugger to examine the setGreeting function.  
Next, click the Evaluate button to evaluate complex expressions, global and local variables, functions, and other relevant parameters in a human-readable format.  
To try it out, follow these steps:  
Click the Evaluate button.  
In the modal that appears, type \_greeting and see what is being passed into the setGreeting function call.  
You should see the output {"result": "hello"}!  
Simulate transactions with updated contract source code  
After detecting the exact cause of a bug, you can immediately try out a potential solution and make sure it works using Transaction Simulator. This feature allows you to simulate transaction execution against real-time and historical Base data, as well as any other supported network, without actually deploying it on-chain.  
You can simulate transactions with updated contract source code and modified transaction parameters within your DevNet to see what would happen if the transaction got executed on Base Network. Running a transaction simulation with modified contract source code allows you to change compilation parameters, such as compiler version, optimization settings, or EVM version, to test your bug fixes.  
Follow these steps to try it out:  
Click the New Simulation button.  
Then, from the Simulation screen, find the Select address drop-down menu and click Greeter.  
Click Edit Source.  
Modify the console.log to say You changed the to the console.log within SetGreeting in the Greeter.sol smart contract and click the Apply button.  
From the Select Option drop-down menu, select "Set Greeting."  
Change the greeting to something like, Hello from Base!  
Click Simulate Transaction.  
Notice the smart contract edit you made is now reflected in the console.log. In addition, you simulated transaction execution under this new condition of a modified smart contract.  
Override the contract state  
In addition, we can change the state of the contract before simulating a transaction. This is important because it allows you to mock contract states and execute highly specific simulation scenarios. Note that state overrides are valid for the simulation you're running. Subsequent simulations will be unaware of the state override.  
To change the execution state, click on Simulator from the DevNet menu.  
Click New Simulation.  
Head over to State Overrides, select your contract, and assign a KEY and VALUE pair.  
For KEY, enter 0x0000000000000000000000000000000000000000000000000000000000000000  
For VALUE, enter 0x48656c6c6f2066726f6d2054656e6465726c79  
Note: The VALUE is hex for Hello from Tenderly. You can get a generated hex value using a string-to-hex converter.  
Click Add.  
Click Simulate.  
INFO  
Note that this has overridden the existing state with a new default greeting. This functionality allows you to run transaction simulations under custom conditions.  
Use DevNets in Continuous Integration (CI)  
Production-level code requires constant testing and monitoring. You can integrate a Tenderly DevNet into your CI pipeline to generate a staging environment for precisely that purpose.  
Follow along using the DevNet example project to set up a CI using GitHub Actions.  
Clone the devnets-example project if you haven't already:  
git clone https://github.com/Tenderly/devnet-examples.git  
Install the Tenderly CLI if you haven't already. Follow the provided instructions to install the CLI.  
Adapt the .github/workflows/smart-contract-ci.yml with the following configuration, and replace ??? with project slug, username, and a DevNet template slug.  
name: Smart Contracts CI  
  
on:  
 push:  
 branches:  
 - main  
 pull\_request:  
 branches:  
 - main  
  
jobs:  
 build-and-test:  
 runs-on: ubuntu-latest  
  
 steps:  
 - name: Checkout code  
 uses: actions/checkout@v3  
  
 - name: Set up Node.js  
 uses: actions/setup-node@v3  
 with:  
 node-version: 16  
  
 - name: Install dependencies  
 run: yarn install  
 working-directory: ./CI-project # hardhat location  
  
 - name: Install Tenderly CLI  
 run: curl https://raw.githubusercontent.com/Tenderly/tenderly-cli/master/scripts/install-linux.sh | sudo sh  
  
 - name: Run tests  
 run: yarn run test:devnet  
 working-directory: ./CI-project # hardhat location  
 env:  
 TENDERLY\_ACCESS\_KEY: ${{ secrets.TENDERLY\_ACCESS\_KEY }}  
 TENDERLY\_PROJECT\_SLUG: '???' # your project slug  
 TENDERLY\_DEVNET\_TEMPLATE: '???' # your devnet template slug  
 TENDERLY\_ACCOUNT\_ID: '???' # your username or organization name  
You can find the DevNet template slug (needed for TENDERLY\_DEVNET\_TEMPLATE) in the DevNets UI. Here is an example:  
Showing how to find TENDERLY\_DEVNET\_TEMPLATE  
From your terminal, run the tests locally. The test:devnet script will spawn a new DevNet automatically and run tests against it.  
npx hardhat test:devnet  
To test, change any file, commit, and push the changes. Any time you push a change, your Tenderly DevNet tests will execute.  
Head over to your project repository on GitHub.  
Click the Actions tab.  
Note that you should now have a workflow run in progress. After the build finishes, you can see all the transactions in a DevNet run.  
Conclusion  
Congrats! In this tutorial, you've learned how to set up your own private replica of Base with Tenderly DevNets. You can now use it to instantly deploy, execute, and debug your smart contracts.  
Plus, you can validate your fixes using transaction simulations before deploying them on-chain. Finally, you can integrate a DevNet environment into your CI flow to speed up and automate your testing process.  
Further guidance  
For more information on the Tenderly full-stack infrastructure, check out the following resources:  
Documentation  
Blog  
Twitter  
Tags:smart contracts  
ON THIS PAGE  
Prerequisites  
Verify your smart contract  
1. Set up a Tenderly account  
2. Install the Tenderly CLI  
3. Create a DevNet template  
4. Customize your DevNet environment  
5. Set up Hardhat and add a DevNet RPC  
Use Tenderly Debugger  
Simulate transactions with updated contract source code  
Override the contract state  
Use DevNets in Continuous Integration (CI)  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/deploy-with-thirdweb  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Deploying a smart contract using thirdweb  
thirdweb is a development framework that allows you to build web3 functionality into your applications.  
In this tutorial, we'll give you an overview of using the thirdweb CLI to deploy a contract to the Base Sepolia test network.  
Objectives  
By the end of this lesson, you should be able to:  
Create a project with a smart contract using thirdweb  
Deploy smart contracts using thirdweb  
Interact with deployed smart contracts using thirdweb  
Prerequisites  
The interactive thirdweb command line interface has everything you need to create, build and deploy smart contracts and apps to Base.  
We recommend using npx to always get the latest version. Alternatively, you can install the CLI as a global command on your machine:  
npm i -g @thirdweb-dev/cli  
Creating a project  
You can use the thirdweb CLI to create a new project that contains a smart contract, alternatively, you can deploy a prebuilt contract for NFTs, Tokens or Marketplace directly from the thirdweb Explore page.  
To create a new project using the CLI, run:  
npx thirdweb create contract  
This will kick off an interactive series of questions to help you get started:  
Give your project a name  
Select Hardhat as the framework  
Select ERC721 as the base contract  
Select None for optional extensions  
Exploring the project  
The create command generates a new directory with your project name. Open this directory in your text editor.  
Inside the contracts folder, you'll find a Contract.sol file; this is our smart contract written in Solidity!  
If we take a look at the code, you can see that our contract is inheriting the functionality of ERC721Base, by:  
Importing the contract  
Inheriting the contract; by declaring that our contract is ERC721Base  
Implementing any required methods such as the constructor.  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
import "@thirdweb-dev/contracts/base/ERC721Base.sol";  
  
contract Contract is ERC721Base {  
 constructor(  
 string memory \_name,  
 string memory \_symbol,  
 address \_royaltyRecipient,  
 uint128 \_royaltyBps  
 ) ERC721Base(\_name, \_symbol, \_royaltyRecipient, \_royaltyBps) {}  
}  
This inheritance pattern lets us use functionality from other contracts inside of ours, modify it, and add custom logic.  
For example, our contract currently implements all of the logic inside the ERC721Base.sol contract; which implements the ERC721A standard with several useful extensions.  
Deploying the contract  
You can use the thirdweb CLI to deploy a smart contract to Base.  
To deploy your smart contracts, from the root directory of your project, run:  
npx thirdweb deploy  
Running this command will:  
Compile all the contracts in the current directory.  
Allow you to select which contract(s) you want to deploy.  
Uploads your contract source code (ABI) to IPFS  
Open the deploy flow in the dashboard  
From the dashboard, you will need to first enter the values for our contract's constructor:  
\_name: The name of our contract  
\_symbol: The symbol or "ticker" given to our contracts tokens  
\_royaltyRecipient: The wallet address that will receive the royalties from secondary sales  
\_royaltyBps: The basis points (bps) that will be given to the royalty recipient for each secondary sale, e.g. 500 = 5%  
Finally, select the Base Sepolia test network as the network you want to deploy to, and click Deploy Now.  
INFO  
For production / mainnet deployments select Base (mainnet) as the network rather than Base Sepolia.  
Once your contract is deployed, you'll be redirected to a dashboard for managing your contract.  
Interacting with your contract  
Thirdweb provides SDKs for various programming languages, including React, React Native, TypeScript, Python, Go, and Unity.  
To interact with your smart contract, you can use the thirdweb CLI to create a web application that is pre-configured with the thirdweb React SDK.  
To create a web application preconfigured with the thirdweb SDK, run:  
npx thirdweb create app –evm  
This will kick off an interactive series of questions to help you get started:  
Give your project a name  
Select Create React App as the framework  
Select TypeScript as the language  
Exploring the project  
The create command generates a new directory with your project name. Open this directory in your text editor.  
Inside the index.tsx file, you'll find the ThirdwebProvider wrapping the entire application.  
This wrapper allows us to use all of the React SDK's hooks and UI Components throughout the application, as well as configure an activeChain; which declares which chain our smart contracts are deployed to.  
Since we deployed our smart contract to the Base network, we'll set the activeChain to BaseSepoliaTestnet:  
...  
import { BaseSepoliaTestnet } from "@thirdweb-dev/chains";  
import { ThirdwebProvider } from "@thirdweb-dev/react";  
  
const container = document.getElementById("root");  
const root = createRoot(container!);  
root.render(  
 <React.StrictMode>  
 <ThirdwebProvider activeChain={BaseSepoliaTestnet}>  
 <App />  
 </ThirdwebProvider>  
 </React.StrictMode>  
);  
  
Interacting with the contract  
To connect to your smart contract in the application, provide your smart contract address (which you can get from the dashboard) to the useContract hook like so:  
import { useContract } from '@thirdweb-dev/react';  
  
export default function Home() {  
 const { contract } = useContract('<CONTRACT\_ADDRESS>');  
  
 // Now you can use the contract in the rest of the component!  
}  
You can now call any function on your smart contract with useContractRead and useContractWrite hooks.  
For example, you can call useContractRead to get the name of the contract:  
const { data, isLoading } = useContractRead(contract, 'name');  
The thirdweb SDK also provides hooks for various interfaces and extensions that make reading and writing data easier. For example, we could use the ERC721 hooks to fetch the metadata for our NFT contract.  
For more information on interacting with smart contracts using the thirdweb SDK, visit the thirdweb developer documentation.  
Deploying the project  
To host your application on IPFS, run the following command:  
yarn deploy  
This command uses Storage to:  
Create a production build of your application  
Upload the build to IPFS  
Generate a URL where your app is permanently hosted.  
That's it! You now have a web application that interacts with smart contracts deployed to Base!  
Tags:smart contracts  
ON THIS PAGE  
Prerequisites  
Creating a project  
Deploying the contract  
Exploring the project  
Interacting with your contract  
Interacting with your contract  
Exploring the project  
Interacting with the contract  
Deploying the project  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/dynamic-nfts  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Build a Dynamic NFT on Base with Irys  
In this tutorial, you will create a dynamic NFT using Irys's mutability features.  
Dynamic NFTs are NFTs whose metadata evolves over time. They are commonly used in:  
Gaming projects where in-game assets evolve as players progress  
Loyalty programs where NFTs evolve as users accumulate points  
Objectives  
By the end of this tutorial, you should be able to:  
Permanently upload data onchain using Irys  
Create NFT metadata and use it to mint an NFT  
"Mutate" (change) the NFT metadata using Irys's mutability features  
Prerequisites  
Setup a Coinbase Wallet: You'll need a web3 wallet; you'll use it to deploy the NFT contract and to sign and pay for uploads to Irys. We recommend using the Coinbase Wallet.  
Wallet Funding: You'll need to fund your wallet with some Base Sepolia tokens. You can do this for free using the Base Faucet.  
About Irys  
Irys is a datachain, a blockchain optimized for data storage. Data uploaded to Irys is:  
Permanent and immutable  
Onchain  
Censorship-resistant  
Blockchain agnostic  
Unconstrained (you can upload files of any size)  
When you store your NFT assets on Irys and mint them using a smart contract on Base, you can guarantee the NFT will be retrievable forever. Creators can rest assured that their works will endure indefinitely, while collectors can feel secure knowing their NFTs are permanently preserved.  
Irys + Base  
Irys has a pay-once-store-forever model and accepts payment for storage using multiple tokens, including ETH on Base.  
"Mutability"  
Data on Irys is permanent and immutable, but you use Irys's mutability features to simulate mutability and create dynamic NFTs that evolve based on onchain or offchain actions.  
Using Irys's mutability features, you create a single, static URL that is linked to a series of transactions. Then, you can add a new transaction to the series at any time, and the URL will always resolve to the most recent transaction.  
You'll mint your NFT using a mutable-style URL, and then push updates to that URL. The URL won't change, but the metadata it resolves to will.  
About  
This tutorial focuses on creating a SuperMon NFT, similar to one used in a web3 game that would evolve during gameplay. The NFT starts with a basic appearance and can be "upgraded" twice. You will use the Irys CLI to "mutate" the metadata, simulating the automatic changes that would occur through player interactions in an actual game.  
Smart contract  
You're building an NFT, which means you need a smart contract. Here's a simple one you can use to mint the NFT you'll create.  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
// Import OpenZeppelin's ERC721 and ERC721URIStorage contracts  
// These URLs are compatible with Remix IDE  
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";  
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";  
import "@openzeppelin/contracts/access/Ownable.sol";  
  
contract SuperMon is ERC721URIStorage {  
 uint256 private \_tokenIdCounter;  
  
 // No arguments in the constructor, the owner will be the contract deployer  
 constructor() ERC721("SuperMon", "SMON") {  
 \_tokenIdCounter = 0;  
 }  
  
 // Mint function to create a new NFT  
 function mint(address to, string memory uri) public {  
 uint256 tokenId = \_tokenIdCounter;  
 \_tokenIdCounter += 1;  
 \_safeMint(to, tokenId);  
 \_setTokenURI(tokenId, uri);  
 }  
}  
  
Deploy the smart contract using Remix.  
Irys CLI  
You'll use the Irys CLI to upload images and metadata.  
Installing the CLI  
Install the Irys CLI globally using the -g flag. Depending on your setup, you may need to use sudo.  
npm i -g @irys/cli  
Or:  
sudo npm i -g @irys/cli  
Using private keys  
When executing CLI commands involving funding and signing, you must provide a private key. Use the -w flag to specify a private key along with the -t flag to signal you're using ETH on Base.  
irys -w <base-private-key> -t base-eth  
Uploading the images  
Download a zip containing PNGs for each level, and save them on your local drive.  
Next, fund Irys with 0.1 [Base Sepolia ETH] to pay for your uploads.  
In all of these CLI examples, make sure to replace the value of the -w parameter with your own private key.  
irys fund 100000000000000000 \  
 -n devnet \  
 -t base-eth \  
 -w 6dd5e....54a120201cb6a \  
 --provider-url https://sepolia.base.org  
The fund command accepts a value in atomic units, 0.1 ETH is equal to 100000000000000000 in atomic units.  
Next, use the Irys CLI to upload each of the images to the Irys Devnet.  
Uploads to Irys's devnet are kept for ~60 days and are paid for using free tokens available from faucets.  
irys upload image-level-1.png \  
 -n devnet \  
 -t base-eth \  
 -w 6dd5e....54a120201cb6a \  
 --provider-url https://sepolia.base.org  
  
irys upload image-level-2.png \  
 -n devnet \  
 -t base-eth \  
 -w 6dd5e....54a120201cb6a \  
 --provider-url https://sepolia.base.org  
  
irys upload image-level-3.png \  
 -n devnet \  
 -t base-eth \  
 -w 6dd5e....54a120201cb6a \  
 --provider-url https://sepolia.base.org  
Uploading the metadata  
Create three metadata files similar to the ones below. Make sure to change the value of the image field to match the URLs generated in the previous step.  
{  
 "name": "SuperMon",  
 "symbol": "SMON",  
 "image": "https://gateway.irys.xyz/3JE8cucmpLkXK1t84QwqDRv25FTB2EJWCUgpWdtvuJZd",  
 "description": "Super dooper, changing shapes, changing power",  
 "attributes": [  
 {  
 "trait\_type": "supermon-level",  
 "value": "1"  
 }  
 ]  
}  
{  
 "name": "SuperMon",  
 "symbol": "SMON",  
 "image": "https://gateway.irys.xyz/3JE8cucmpLkXK1t84QwqDRv25FTB2EJWCUgpWdtvuJZd",  
 "description": "Super dooper, changing shapes, changing power",  
 "attributes": [  
 {  
 "trait\_type": "supermon-level",  
 "value": "2"  
 }  
 ]  
  
}  
{  
 "name": "SuperMon",  
 "symbol": "SMON",  
 "image": "https://gateway.irys.xyz/3JE8cucmpLkXK1t84QwqDRv25FTB2EJWCUgpWdtvuJZd",  
 "description": "Super dooper, changing shapes, changing power",  
 "attributes": [  
 {  
 "trait\_type": "supermon-level",  
 "value": "3"  
 }  
 ]  
  
}  
And upload just the first file using the Irys CLI.  
irys upload metadata-level-1.json \  
 -n devnet \  
 -t base-eth \  
 -w 6dd5e....54a120201cb6a \  
 --provider-url https://sepolia.base.org  
The CLI will return a URL similar to https://gateway.irys.xyz/94TNg3UUKyZ96Dj8eSo9DVkBiivAz9jT39jjMFeTFvm3. To convert that to a mutable-style URL, interpolate it by adding /mutable/ after the domain and before the transaction ID.  
Your final URL will be similar to https://gateway.irys.xyz/mutable/94TNg3UUKyZ96Dj8eSo9DVkBiivAz9jT39jjMFeTFvm3.  
Minting the NFT  
To mint your NFT in Remix:  
Return to Remix.  
Under "Deployed Contracts", locate your contract and expand it to see its functions.  
Under the Mint function, enter the wallet address you want to mint the NFT to and the metadata URL (e.g. https://gateway.irys.xyz/mutable/94TNg3UUKyZ96Dj8eSo9DVkBiivAz9jT39jjMFeTFvm3) from the previous step.  
Click Transact.  
You can now view the NFT on the Opensea Testnet.  
Mutating the metadata  
To now "mutate" the NFT, upload a new version of the metadata tagging it as having a Root-TX equal to the transaction ID of your first transaction. In my example, I pass the value of 94TNg3UUKyZ96Dj8eSo9DVkBiivAz9jT39jjMFeTFvm3, however make sure to change this to match your unique transaction ID.  
irys upload metadata-level-2.json \  
 -n devnet \  
 -t base-eth \  
 -w 6dd5e....54a120201cb6a \  
 --tags Root-TX 94TNg3UUKyZ96Dj8eSo9DVkBiivAz9jT39jjMFeTFvm3 \  
 --provider-url https://rpc.sepolia.org  
Return to Opensea and request that it refresh your metadata.  
Give it a few minutes and your updated NFT should be visible.  
Free metadata uploads  
Uploads of less than 100 KiB are free on Irys, which is more than enough for most metadata files. This means projects can let users "evolve" their NFTs without having to pay gas fees.  
Caching  
Wallets and NFT websites typically cache metadata to optimize performance, this can affect the visibility of updates to dynamic NFTs. While OpenSea offers a feature for users to manually request metadata refreshes, not all platforms provide this level of control. When building dynamic NFT projects, make sure to thoroughly test and understand the implications of caching on your platform.  
Irys SDK  
This tutorial used the Irys CLI to permanently upload data. Irys also has a JS-SDK that can be used on the server and in the browser.  
Conclusion  
In this tutorial, you learned how to permanently upload data to Irys using their CLI and how to create a dynamic NFT. Data on Irys is onchain, permanent and immutable. When your NFT images and metadata are on Irys, you can guarantee to your users that the NFT will be retrievable forever.  
Dynamic NFTs are commonly used with gaming projects, similar to the one we built in this tutorial. However, there are countless other applications too. For example:  
Points programs: Create an NFT representing a user's participation in a points program. As the user earns more points, the NFT evolves.  
Sports NFTs: Create an NFT from a sports team that changes whenever the team wins an important game.  
Holiday NFTs: Create an NFT that changes seasonally to represent different holidays.  
Activity tracking: Create a health and wellness NFT that changes based on data from an activity tracker.  
Tags:nft  
ON THIS PAGE  
Prerequisites  
About Irys  
"Mutability"  
Irys + Base  
About  
Smart contract  
Irys CLI  
Uploading the images  
Installing the CLI  
Using private keys  
Uploading the metadata  
Minting the NFT  
Mutating the metadata  
Free metadata uploads  
Caching  
Irys SDK  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/event-gate-with-nouns  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Gate IRL Events with Nouns  
You've probably seen people in onchain communities use ⌐◨-◨ in their profile names. These are called Nouns Goggles, or Noggles. They're an ASCII representation of the glasses found on every procedurally generated Nouns NFT avatar. The Nouns Auction makes one new Noun available for auction every single day - forever!  
In this tutorial, you'll learn how to use Nouns and the Coinbase Smart Wallet to create an app in which non-crypto-native participants at an IRL event can be onboarded and receive Nounish avatars.  
Objectives  
By the end of this tutorial you should be able to:  
Deploy a copy of the Nouns Protocol  
Construct a web app that scan a user's QR code and indicate whether or not the user possesses the appropriate NFT  
Prerequisites  
ERC-721 Tokens  
This tutorial assumes that you are familiar with the properties of ERC-721 tokens.  
Vercel  
You'll need to be comfortable deploying your app to [Vercel], or using another solution on your own. Check out our tutorial on [deploying with Vercel] if you need a refresher!  
Onchain Apps  
The tutorial assumes you're comfortable with the basics of deploying an app and connecting it to a smart contract. If you're still learning this part, check out our tutorials in Base Learn for [Building an Onchain App].  
The Nouns Protocol  
Ownership of a Noun gives a wallet address membership in the Nouns DAO. The Nouns Protocol is open-source and the art is public domain, so any builder can leverage the protocol to create their own community. For example, the Purple DAO supports Farcaster.  
Creating your Own DAO  
You can use the contracts in the Nouns Monorepo to deploy your own DAO, but there's an easier method! BuilderDAO is a DAO that maintains a set of tools making it easy for anyone to create their own DAO.  
Navigate to the Testnet Builder DAO site and connect your wallet.  
DANGER  
The name of this site is a little misleading. It will treat the DAOs you create here as test DAOs, but it deploys contracts on Base Mainnet, not Base Sepolia. Luckily, gas is inexpensive enough that this isn't a terrible price.  
Click the green circle in the upper right, and select Create a DAO  
General  
Give your DAO a name and symbol. Optionally, upload a cover avatar image and/or link to a website. Click Continue.  
Auction  
For testing, you'll want the Auction Reserve Price as low as possible, .0001 ETH, and the Auction Duration to be enough time to buy a few NFTs for testing with different wallet addresses, but not so long you get bored. About 2 minutes is a good middle ground.  
Be sure to change Days from 1 to 0!  
Veto and Allocation  
Set Veto Power and initial Token Allocation as you see fit. To speed up testing, you may wish to grant several addresses here.  
CAUTION  
The app gets confused if you change addresses from the browser extension wallet, copy them from another window, or another location.  
Artwork  
Add a description for your DAO. You must put something in this field.  
For the art, click the Download demo folder link and review the assets downloaded if you're not familiar with how these types of collections assemble NFT art out of random properties.  
When you're done, simply upload the folder you downloaded. Use the Generate Random Preview button a few times to see the variants of NFTs this protocol can create, then click Continue.  
Deploying the Contracts  
Review the linked documents for the ToS and Protocol Rewards, the check the boxes and click Deploy Contracts (1 of 2).  
Approve the transaction. Remember, this is using real funds!  
Click Deploy Contracts (2 of 2) and approve the transaction. After it processes, you should be taken to the auction page for your DAO. Click Start Auction and approve the transaction.  
If you chose to distribute tokens to founding addresses, those will be airdropped to those addresses. Otherwise, use the auction to purchase NFTs with a few different addresses.  
Finding the Contracts  
Near the bottom of the page, click the Contracts tab. You'll need the NFT contract address for the app.  
Building the App  
The app is designed to be used by the person at the door to a private event for DAO members. To gain entry, members must prove their ownership of at least 1 NFT from the DAO.  
The app will work by using a QR code scanner to read the member's "pay me" QR code, look up their NFT ownership status, then share that with the person using the app to scan entrants.  
Setting up the Template  
Start by using the Onchain App Template as the foundation of your app.  
Clone your copy of the repo, install dependencies, and check that it works.  
Open src/app/page.tsx. Clean out the demo content and set up for your app:  
'use client';  
  
export default function Page() {  
 return (  
 <div className="flex w-96 flex-col md:w-[600px]">  
 <section className="mb-6 flex w-full flex-col border-b border-sky-800 pb-6">  
 <aside className="mb-6 flex">  
 <h2 className="text-2xl">Welcome to our Event!</h2>  
 </aside>  
 <main className="flex flex-col space-x-4">  
 <p className="text-body mb-4 text-white">  
 Please scan the QR code for your wallet address that holds the NFT.  
 </p>  
 </main>  
 </section>  
 </div>  
 );  
}  
Reading QR Codes  
You'll need a library to read QR codes. A lightweight option is Html5-QRCode.  
Add it with:  
bun add html5-qrcode  
In page.tsx, import, and add the state variables needed by the scanner, as outlined in the Html5-QRCode Docs, and the variables you'll need to show whether or not to let the person into the event:  
import { Html5Qrcode } from 'html5-qrcode';  
const [scannedAddress, setScannedAddress] = useState<`0x${string}` | null>(null);  
const [authorized, setAuthorized] = useState(false);  
const [scanning, setScanning] = useState(false);  
scannedAddress will be populated by the QR Code scanner. You'll use it with useEffect to check for NFT ownership.  
Review the operations of the scanner from the Html5-QRCode Docs, and modify it to only look for addresses, and to handle the different formats used by popular wallets:  
async function startScanning() {  
 setScanning(true);  
 setScannedAddress(null);  
 setAuthorized(false);  
 let cameraId = '';  
  
 try {  
 // This method will trigger user permissions  
 const devices = await Html5Qrcode.getCameras();  
  
 if (!devices || devices.length === 0) {  
 return;  
 }  
  
 cameraId = devices[0].id;  
  
 const html5QrCode = new Html5Qrcode('reader', false);  
  
 try {  
 html5QrCode.start(  
 { facingMode: 'environment' } || cameraId,  
 {  
 fps: 10, // Optional, frame per seconds for qr code scanning  
 qrbox: { width: 500, height: 500 }, // Optional, if you want bounded box UI  
 },  
 (decodedText) => {  
 let scannedAddress = decodedText;  
 if (scannedAddress.includes('ethereum:')) {  
 scannedAddress = scannedAddress.replace('ethereum:', '');  
 }  
  
 // If there is an @ symbol, remove it and everything after it  
 if (scannedAddress.includes('@')) {  
 scannedAddress = scannedAddress.split('@')[0];  
 }  
  
 // If it's not the correct length, it's not a valid address  
 if (scannedAddress.length !== 42) {  
 return;  
 }  
  
 // handle the result  
 setScannedAddress(scannedAddress as `0x${string}`);  
 setScanning(false);  
 html5QrCode.stop();  
 },  
 (errorMessage) => {  
 // parse error, ignore it.  
 },  
 );  
 } catch (error) {}  
 } catch (error) {}  
}  
Update the returned HTML to display:  
The scanner object  
The last scanned address  
Whether or not the last scanned address is authorized to enter  
Any errors  
A button to start up the scanner  
return (  
 <div className="flex w-96 flex-col md:w-[600px]">  
 <section className="mb-6 flex w-full flex-col border-b border-sky-800 pb-6">  
 <aside className="mb-6 flex">  
 <h2 className="text-2xl">Welcome to our Event!</h2>  
 </aside>  
 <main className="flex flex-col space-x-4">  
 <p className="text-body mb-4 text-white">  
 Please scan the QR code for your wallet address that holds the NFT.  
 </p>  
 <div id="middle" className="flex h-3/4 items-center justify-center">  
 {!scanning && (  
 <button  
 onClick={() => startScanning()}  
 className="rounded bg-blue-500 px-4 py-2 font-bold text-white hover:bg-blue-700"  
 >  
 Scan QR Code  
 </button>  
 )}  
 {scanning && (  
 <div id="reader-container" className="h-full w-full">  
 <div id="reader"></div>  
 </div>  
 )}  
 </div>  
 <div id="bottom" className="h-1/8">  
 <br />  
 <p>Last scanned: {scannedAddress}</p>  
 <p>Authorized: {authorized ? 'Yes' : 'No'}</p>  
 </div>  
 </main>  
 </section>  
 </div>  
);  
Run the app, and press the button to enable your camera. You may have to grant permissions.  
On your phone, open your wallet app, display the QR code you use to receive payments and show it to the camera. Your wallet address will be displayed below the video. If you have another wallet app, try it as well.  
Authorized won't change because you aren't checking for NFT ownership yet.  
Verifying NFT Ownership  
To verify NFT ownership, you can use the ERC-721 balanceOf function and check that the balance returned for the scanned address is > 0. To call this, you'll need the address of the NFT contract, which you got above from the Nouns Builder page, and the ABI for the contract.  
You may be used to generating the ABI as a part of your deployment process, but that's not the only way to get it. They also don't need to be complete, and aren't unique for contracts following a specification.  
This means that you can use a simplified one here. Add a folder in app called constants, and a file called abi.ts. In it, put:  
export const abi = [  
 {  
 constant: true,  
 inputs: [  
 {  
 name: 'owner',  
 type: 'address',  
 },  
 ],  
 name: 'balanceOf',  
 outputs: [  
 {  
 name: '',  
 type: 'uint256',  
 },  
 ],  
 payable: false,  
 stateMutability: 'view',  
 type: 'function',  
 },  
] as const;  
Back in page.tsx, import your ABI and add a constant for your contract address:  
import { abi } from './constants/abi';  
const contractAddress = '<YOUR CONTRACT ADDRESS HERE>';  
Your contract is on Base Mainnet, so you need to modify src/wagmi.ts to use that. Go ahead and update your app name while you're there:  
import { http, createConfig } from 'wagmi';  
import { base } from 'wagmi/chains';  
import { coinbaseWallet } from 'wagmi/connectors';  
  
export const wagmiConfig = createConfig({  
 chains: [base],  
 // turn off injected provider discovery  
 multiInjectedProviderDiscovery: false,  
 connectors: [  
 coinbaseWallet({  
 appName: 'DAO Gate',  
 preference: 'all',  
 version: '4',  
 }),  
 ],  
 ssr: true,  
 transports: {  
 [base.id]: http(),  
 },  
});  
  
declare module 'wagmi' {  
 interface Register {  
 config: typeof wagmiConfig;  
 }  
}  
Return to page.tsx, and implement a useReadContract to read the balanceOf the scanned address. First, add needed dependencies:  
import { useReadContract } from 'wagmi';  
import { useQueryClient } from '@tanstack/react-query';  
And the queryClient:  
export default function Page() {  
 // Other state variables  
  
 const queryClient = useQueryClient();  
  
 // Other code...  
Then, add the useReadContract hook:  
const {  
 data: balanceData,  
 isError: balanceIsError,  
 isPending: balanceIsPending,  
 queryKey: balanceQueryKey,  
} = useReadContract({  
 address: contractAddress,  
 abi,  
 functionName: 'balanceOf',  
 args: scannedAddress ? [scannedAddress] : undefined,  
});  
And a useEffect to handle the balanceData being retrieved. You should also make use of the additional variables to handle errors, pending state, etc. on your own.  
useEffect(() => {  
 if (balanceData !== undefined) {  
 const balance = balanceData as bigint;  
 setAuthorized(balance > 0n);  
 }  
}, [balanceData]);  
You also need a useEffect to refetch the data from the blockchain when scannedAddress changes, accomplished by invalidating the query for the queryKey you extracted from useReadContract :  
useEffect(() => {  
 if (scannedAddress) {  
 queryClient.invalidateQueries({ queryKey: balanceQueryKey });  
 }  
}, [scannedAddress]);  
With these pieces in place, when the scanner scans a QR code, it will check if what it has read is a valid address and return it. That triggers an update to the scannedAddress, which in turn causes a read of the blockchain to see if that address owns any of the NFTS. If the address does own at least one NFT, it sets authorized to true, and displays that the user may enter.  
From here, you can add much better styling, or even pull the NFT art of the owner's NFT for a personalized welcome! Another great feature would be displaying a status level based on the number of NFTs owned by the entrant.  
Conclusion  
In this tutorial, you learned about Nouns and how the platform makes it easy to create avatar-based communities. You also learned how to use BuilderDAO to create your own DAO without writing any code. Finally, you learned how to create an app that can be used to allow or deny access to a private event based on ownership of an NFT in your DAO.  
Tags:nftsmart contractsfrontend  
ON THIS PAGE  
Prerequisites  
The Nouns Protocol  
ERC-721 Tokens  
Vercel  
Onchain Apps  
Creating your Own DAO  
Veto and Allocation  
Artwork  
Deploying the Contracts  
Building the App  
Setting up the Template  
Reading QR Codes  
Verifying NFT Ownership  
Conclusion  
Setting up the Template  
Reading QR Codes  
Verifying NFT Ownership  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/farcaster-cast-actions-simple  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Farcaster Cast Actions: Create a Simple Cast Action  
Cast Actions are a new feature of Farcaster that enable you to create actions that users can install on their Farcaster client and then click a button on any cast to trigger the action. In this tutorial, you'll learn how to build a simple cast action with OnchainKit that returns a message to the Farcaster client.  
CAUTION  
Cast Actions are brand new and tools for building them are evolving quickly. Check the Cast Actions docs and OnchainKit [changelog]!  
Objectives  
By the end of this tutorial you should be able to:  
Create a Farcaster cast action that returns a message when triggered by a user  
Identify the location to add additional functionality to the cast  
Enable users to install the cast action  
Prerequisites  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. If you need a refresher, check out our tutorial on deploying with Vercel!  
Farcaster  
You must have a Farcaster account with a connected wallet. Check out the Base channel to stay in the loop when we release tutorials like this!  
Setup  
Create a copy of a-frame-in-100-lines. It's a template, so you can easily add a new copy by clicking the green button at the top of the page. If you're working off of an existing copy, check to make sure you have the most up-to-date version of OnchainKit!  
Run yarn install.  
Cast actions need to be deployed to a server to work, so connect your repo to Vercel with CI/CD. Remember that while doing so does make it much easier to test and redeploy, it also makes it easier to break a live cast action!  
Open app/config.ts and update NEXT\_PUBLIC\_URL to your new deployment.  
Creating A Simple Cast Action  
Cast actions work similarly to the way Frames call your api endpoint. As a result, you can quickly adapt the code of an endpoint expecting a frame to instead manage a cast action.  
Add a new folder in api called action with a file called route.ts to create a new route in your app. Import dependencies from OnchainKit and Next.js:  
import { FrameRequest, getFrameMessage } from '@coinbase/onchainkit/frame';  
import { NextRequest, NextResponse } from 'next/server';  
You won't be returning a frame, so you don't need getFrameHtmlResponse.  
Stub out your POST handler and response function:  
async function getResponse(req: NextRequest): Promise<NextResponse> {  
 // TODO  
}  
  
export async function POST(req: NextRequest): Promise<Response> {  
 console.log('POST');  
 return getResponse(req);  
}  
  
export async function GET(req: NextRequest): Promise<Response> {  
 console.log('GET');  
 return getResponse(req);  
}  
  
export const dynamic = 'force-dynamic';  
Parse and validate the request exactly the same as you would a frame in the top of getResponse:  
const body: FrameRequest = await req.json();  
const { isValid, message } = await getFrameMessage(body, { neynarApiKey: 'NEYNAR\_ONCHAIN\_KIT' });  
  
if (!isValid) {  
 return new NextResponse('Message not valid', { status: 500 });  
}  
Instead of returning a frame, you'll return a json NextResponse with a string message and a 200 status:  
return NextResponse.json({ message: 'Hello from the frame route' }, { status: 200 });  
Testing the Cast Action  
Commit your work to trigger a deploy in Vercel. Once it's done, you can test your cast action with the Cast Actions Playground.  
The above link has an example embedded in it. Try it out, then replace the Post URL with your own. Click Submit Action and you will see the hello message from your route.  
Performing a Task in the Cast Action  
Before sending the response, you can do whatever you want to give your cast action functionality. Possibilities include:  
Extracting the cast hash or caster FID from the request and using it for something  
Reading or writing from a database or smart contract  
Anything else you'd use an API call to do on behalf of a user  
For now, you can add a simple addition to let the user know what time it is.  
Modify your return:  
// Get the current date and time  
const now = new Date();  
const date = now.toLocaleDateString();  
const time = now.toLocaleTimeString();  
  
// Format them as a string  
const dateTime = `${date} ${time} UTC`;  
  
return NextResponse.json({ message: dateTime }, { status: 200 });  
Installing the Cast Action  
To install a cast action, you need to pick an Octicon from the list at the bottom of the Cast Actions doc. You also need to decide on a name for your action. Using those, you create a deeplink to Warpcast that will prompt the user to install your action. Currently, this replaces their existing action, but the capacity for multiple will be added soon.  
For example:  
https://warpcast.com/~/add-cast-action?actionType=post&name=Current+Time&icon=clock&postUrl=https%3A%2F%2Ftest-cast-actions.vercel.app%2Fapi%2Faction  
The Cast Actions Playground has a nice tool to build this link for you!  
From a Web Page  
To enable your users to install your action from a web page, simply include the link. Update the root of your site to do so:  
export default function Page() {  
 return (  
 <>  
 <div>  
 <a href="https://warpcast.com/~/add-cast-action?actionType=post&name=Current+Time&icon=clock&postUrl=https%3A%2F%2Ftest-cast-actions.vercel.app%2Fapi%2Faction">  
 Click here to install the Current Time action on Warpcast.  
 </a>  
 </div>  
 </>  
 );  
}  
From a Frame  
Finally, update the frame on your root page to also have the install link:  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 action: 'link',  
 label: 'Add Current Time Action',  
 target:  
 'https://warpcast.com/~/add-cast-action?actionType=post&name=Current+Time&icon=clock&postUrl=https%3A%2F%2Ftest-cast-actions.vercel.app%2Fapi%2Faction',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/park-3.png`,  
 aspectRatio: '1:1',  
 },  
});  
Conclusion  
In this tutorial, you created a Farcaster that allows your users to get the current date and time from a cast. You also learned where and how you can add further functionality to your action. Finally, you created a link and a frame to allow your users to install your cast action.  
Watch the Base Channel to stay up-to-date on new developments!  
Tags:framesactionsOnchainKit  
ON THIS PAGE  
Prerequisites  
Setup  
Vercel  
Farcaster  
Creating A Simple Cast Action  
Performing a Task in the Cast Action  
Testing the Cast Action  
Installing the Cast Action  
Conclusion  
From a Web Page  
From a Frame  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/farcaster-frames-deploy-to-vercel  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Farcaster Frames: Deploying to Vercel  
To share your Frames on Farcaster, you must first deploy them to the web. Farcaster reads the metadata from your site to build the Frame as it initially appears to the user. In this tutorial, you'll learn how to deploy the frame in the OnchainKit example - a-frame-in-100-lines.  
Objectives  
By the end of this tutorial, you should be able to:  
Deploy a Farcaster Frame with Vercel  
Use CI/CD to automatically deploy to Vercel when you update your main or master branch  
Find and review logs and error messages for your deployed Frame  
Prerequisites  
Onchain App Development  
You'll need to be comfortable building onchain apps.  
GitHub  
You'll need a GitHub account and should be comfortable pushing code to it. If you're using another solution for managing your code, you should be able to adapt this tutorial to your use case.  
Farcaster  
You must have a Farcaster account with a connected wallet. Check out the Base channel to stay in the loop when we release tutorials like this!  
Setup and Testing the Template  
Start by creating a new repo using a-frame-in-100-lines as a template.  
Run yarn install, then yarn dev.  
In the browser, open http://localhost:3000/. All you'll see is a heading with Zizzamia.xyz. This is expected!  
Open app/page.tsx. Here, you'll find the initial setup of the metadata that Farcaster reads to create the frame, as well as the simple page you just viewed. Change the <h1> to be your name.  
export default function Page() {  
 return (  
 <>  
 <h1>YOUR NAME HERE</h1>  
 </>  
 );  
}  
Change the label of the first button in buttons to be something you'll recognize as well. This can be your name, your pet's, or anything you like!  
buttons: [  
 {  
 label: 'YOUR NAME HERE',  
 },  
 {  
 action: 'tx',  
 label: 'Send Base Sepolia',  
 target: `${NEXT\_PUBLIC\_URL}/api/tx`,  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/tx-success`,  
 },  
],  
Setting up Vercel  
Navigate to Vercel. Create an account if you need to, then sign in. You'll want to use your social login from GitHub, GitLab, or BitBucket.  
You should see something like this:  
Click the Install button to install the Vercel app in your Github organization. You'll need to select the appropriate choice for your organization between All repositories and Only selected repositories.  
All is more convenient, but gives a (well respected) third party more access than is required. For this tutorial, we're assuming that you've chosen to go with minimum necessary access. Click the Select repositories dropdown, and pick the repo for your Frame.  
Click Install.  
You should see the Import Git Repository screen:  
INFO  
If you've come back to Vercel after a few months, it may be unclear how to add more repos. To do so, use the Add New... dropdown on the projects page to get to the screen above, then click Adjust GitHub App Permissions ->. Scroll down in the popup window to find your list of repos to add.  
Click the button, then configure your project. The a-frame-in-100-lines example uses Next.js, which is made by the same people as Vercel, so the default settings are fine. Pick a new name if you'd like, then click Deploy.  
CAUTION  
You won't have been the first person to name a project a-frame-in-100-lines, so Vercel will adjust the name for you for the file path, if you gave your template copy the same name. Don't be confused when your changes don't show up at a-frame-in-100-lines.vercel, app, that one isn't yours!  
After the deploy completes, click Continue to Dashboard.  
On this screen, you'll see:  
A Deployment link to this specific deployment. These persist, so you can always go back to compare before and after if you need to  
A Domains link that goes to the main domain for your project. This will always link to your most recent deploy  
The Status of your deploy, indicating, success, failure, or in process for each build  
A preview of the front page of your site  
There are also a number of useful tabs. You should eventually check them all out, but for now inspect:  
Project: You are here  
Deployments: A list of the current, and all prior deployments. You'll use this tab to monitor new deployments as they build, or to revert to an old one if you need to  
Logs: Any console.log or console.error logs you put in your code will appear here. Note that there may be a few seconds delay  
Settings: Among other things, this is where you can set your environment variables. If it works locally but not deployed, this is a good place to review!  
CI/CD with Vercel  
By default, Vercel sets up CI/CD, so whenever you push a change to your main branch, it will automatically redeploy!  
Make a trivial change to your repo, commit it, and push it. When you do so, you'll see a new build in the Deployments tab:  
If it fails, you'll get a fairly comprehensive log explaining what happened. Otherwise, it will automatically deploy!  
Manual Deploys  
If you want, you can disable CI/CD and set up manual deploys. One convenient way to do this is to set up a one click deploy button.  
Testing the Cast  
Open the Frame Validator and paste in your link. Viewing a frame from the validator will re-cache it for future casts, but it will not change any existing casts of the frame.  
Click Load. You should see:  
Except it will have the text that you edited!  
Conclusion  
In this tutorial, you learned how to deploy a frame with Vercel and take advantage of CI/CD.  
Tags:framesOnchainKit  
ON THIS PAGE  
Prerequisites  
Setup and Testing the Template  
Onchain App Development  
GitHub  
Farcaster  
Setting up Vercel  
CI/CD with Vercel  
Manual Deploys  
Testing the Cast  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/farcaster-frames-gating-and-redirects  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Farcaster Frames: Gating content and creating redirects  
You can build complex and exciting Frames on Farcaster that meet the community's expectation for new and exciting things to do. OnchainKit, and our a-frame-in-100-lines demo give you a number of tools to build complex interactions within a frame.  
Objectives  
By the end of this tutorial, you should be able to:  
Build a Farcaster Frame that gates content behind a requirement that the user follows, recasts, or likes you or your frame  
Create frames that use the text field and extract it from the message  
Construct redirect buttons in Frames  
Create multiple paths within a Frame based on the button clicked by a user  
Prerequisites  
Onchain App Development  
You'll need to be comfortable building onchain apps.  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. Check out our tutorial on deploying with Vercel if you need a refresher!  
Farcaster  
You must have a Farcaster account with a connected wallet. Check out the Base channel to stay in the loop when we release tutorials like this!  
Frames  
You should be comfortable with the basics of creating Farcaster Frames. If you aren't, check out our tutorial on NFT Minting Frame.  
Setting Up a Copy of A Frame in 100 Lines  
Create a new repo using a-frame-in-100-lines as a template.  
Add your new repo to Vercel and deploy. If you need a refresher, check out deploying with Vercel. Open the Frame Validator and test the current version of the template.  
The demo has examples of most of the features available for your frames.  
Buttons and Images  
The core functionality of a frame is an image and at least 1 button. Buttons can do a number of things, including requesting a new frame to replace the current one, opening a transaction for the user to consider approving, or opening a link to a website.  
Text Entry  
Optionally, frames can include a text entry field. Try it out! When you click the Story Time! button, the text will be preserved, and will appear in the button on the next frame.  
The text input field is created by adding the input property to getFrameMetadata.  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 label: 'Story time',  
 },  
 {  
 action: 'tx',  
 label: 'Send Base Sepolia',  
 target: `${NEXT\_PUBLIC\_URL}/api/tx`,  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/tx-success`,  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/park-3.png`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: 'Tell me a story',  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
});  
When the user enters the text, it gets included in the frame message. You can see how it is retrieved in api/frame/route.ts. The getFrameMessageBody function extracts the frame message from the request, and validates it. The returned message contains a number of useful properties that can be seen where it is defined in OnchainKit, in src/frame/types.ts:  
export interface FrameValidationData {  
 address: string | null; // The connected wallet address of the interacting user.  
 button: number; // Number of the button clicked  
 following: boolean; // Indicates if the viewer clicking the frame follows the cast author  
 input: string; // Text input from the viewer typing in the frame  
 interactor: {  
 fid: number; // Viewer Farcaster ID  
 custody\_address: string; // Viewer custody address  
 verified\_accounts: string[]; // Viewer account addresses  
 verified\_addresses: {  
 eth\_addresses: string[] | null;  
 sol\_addresses: string[] | null;  
 };  
 };  
 liked: boolean; // Indicates if the viewer clicking the frame liked the cast  
 raw: NeynarFrameValidationInternalModel;  
 recasted: boolean; // Indicates if the viewer clicking the frame recasted the cast  
 state: {  
 serialized: string; // Serialized state (e.g. JSON) passed to the frame server  
 };  
 transaction: {  
 hash: string;  
 } | null;  
 valid: boolean; // Indicates if the frame is valid  
}  
The demo doesn't make use of the input property, but it does extract the text:  
//api/frames/route.ts  
  
// Extract the input:  
const text = message.input || '';  
Link Button  
You can now add outbound links to buttons. To do this with OnchainKit, simply create a button with an action property of link, and a target of the desired url:  
{  
 action: 'link',  
 label: 'Link to Google',  
 target: 'https://www.google.com',  
},  
Redirect Button  
You can also do a redirect with a button. The example has one on the second frame - Dog pictures ↗:  
{  
 action: 'post\_redirect',  
 label: 'Dog pictures',  
},  
The way it works is a little tricky.  
Clicking this button hits the postUrl, with the requirement that you return a redirect response with a status of 302, and a link. You can see that in route.ts:  
if (message?.button === 3) {  
 return NextResponse.redirect(  
 'https://www.google.com/search?q=cute+dog+pictures&tbm=isch&source=lnms',  
 { status: 302 },  
 );  
}  
Try changing it to your favorite link!  
Creating Gates  
The message, shown above, has properties that make it easy to design interactions based on whether or not the user interacting with the frame has liked or recast your post, or if they follow the original caster.  
!!!caution  
The community is evolving quickly and many people are fatigued with "Like, follow, recast" spam in their feeds. These are useful tools, but some consideration should be given for designing an experience for the current culture of the community you are trying to reach.  
!!!  
Like Gate  
To require the user to "like" the cast before seeing images of puppies, simply modify the conditional for the redirect to include that check:  
if (message?.button === 3 && message?.liked) {  
 return NextResponse.redirect(  
 'https://www.google.com/search?q=cute+dog+pictures&tbm=isch&source=lnms',  
 { status: 302 },  
 );  
}  
If they haven't, return a version of the original frame with a new message in the third button. In doing so, you've created a loop with a behavior condition for the user to exit:  
if (message?.button === 3 && message.liked) {  
 return NextResponse.redirect(  
 'https://www.google.com/search?q=cute+dog+pictures&tbm=isch&source=lnms',  
 { status: 302 },  
 );  
} else if (message?.button === 3) {  
 return new NextResponse(  
 getFrameHtmlResponse({  
 buttons: [  
 {  
 label: `State: ${state?.page || 0}`,  
 },  
 {  
 action: 'link',  
 label: 'OnchainKit',  
 target: 'https://onchainkit.xyz',  
 },  
 {  
 action: 'post\_redirect',  
 label: 'Like to see doggos!',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/park-1.png`,  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
 state: {  
 page: state?.page + 1,  
 time: new Date().toISOString(),  
 },  
 }),  
 );  
}  
Note that we're returning a getFrameHtmlResponse here, not a getFrameMetadata. That's only used for the initial frame created from a page!  
Follow and Recast Gates  
On your own, try changing the "like" gate to require the user to follow you, recast, or all three!  
INFO  
Hint: Take a close look at the FrameValidationData properties. OnchainKit makes this easy!  
Conclusion  
In this tutorial, you learned how to use the main features of Frames - text input, link buttons, and redirects. You also learned how to use new features in OnchainKit to require your users to perform certain actions to unlock features in your Frame. Finally, you learned how to create a loop in your Frame's behavior, which can be used to create very complicated Frames!  
Tags:framesOnchainKit  
ON THIS PAGE  
Prerequisites  
Setting Up a Copy of A Frame in 100 Lines  
Buttons and Images  
Text Entry  
Link Button  
Redirect Button  
Creating Gates  
Like Gate  
Follow and Recast Gates  
Link Button  
Redirect Button  
Conclusion  
Like Gate  
Follow and Recast Gates  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/farcaster-frames-hyperframes  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Farcaster Frames: Building HyperFrames  
Frames on Farcaster are getting more complex. Developers are now building interactions that require a handful, or even dozens of frames in response to various user states, inputs, and actions. HyperFrames are a system to organize navigation for large numbers of frames, using OnchainKit. In this tutorial, we'll use making the navigation for an old-school adventure game fully in a frame. You can use this same technique for many other intents with your Frames, such as games, stores, customized mints, etc.  
Objectives  
By the end of this tutorial, you should be able to:  
Build HyperFrames - connected Farcaster Frames that manage significant numbers of cross-linked frames in an organized fashion  
Add management of conditional states to the navigation system  
Use frame state to pass information from one frame to another  
Prerequisites  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. Check out our tutorial on deploying with Vercel if you need a refresher!  
Farcaster  
You must have a Farcaster account with a connected wallet. Check out the [Base channel] to stay in the loop when we release tutorials like this!  
Frames  
You should be comfortable with the basics of creating Farcaster Frames. If you aren't, check out our tutorial on NFT Minting Frame.  
Getting Started  
This tutorial assumes you're using a-frame-in-100-lines as the base for your project. It's a very lightweight template, so with minor modifications, you can adapt this technique to any project.  
You'll also need to create or find a handful of images to use for each frame. AI tools are wonderful for this type of prototyping, or you can right-click and save the images from the old-school adventure game.  
Creating the First Frame  
Open app/page.tsx. Modify the getFrameMetadata for the first frame to match the frame in the example.  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 label: 'Road',  
 },  
 {  
 label: 'Woods',  
 },  
 {  
 label: 'Cave',  
 },  
 {  
 action: 'link',  
 label: 'TODO',  
 target: 'https://www.google.com',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/frame-1-forest.webp`,  
 aspectRatio: '1:1',  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
});  
CAUTION  
Per the Frames specification, state is not an allowed property on the first frame.  
Configure the rest of the metadata as you see fit. Remember, this won't show up in your frame, but it will appear if someone links your site to another platform that uses the standard Open Graph metadata.  
export const metadata: Metadata = {  
 title: 'HyperFrames!',  
 description: 'Time is a flat circle.',  
 openGraph: {  
 title: 'HyperFrames!',  
 description: 'Time is a flat circle.',  
 images: [`${NEXT\_PUBLIC\_URL}/frame-1-forest.webp`],  
 },  
 other: {  
 ...frameMetadata,  
 },  
};  
Setting up the Route  
The route you'll construct is similar to the example in app/api/route.ts of the 100-lines example. It will use OnchainKit to retrieve and validate the message from the frame. In doing so, it will collect:  
The user's address  
The button clicked by the user to get here  
Text, if the previous frame had a text box  
Any state data included in the sending frame  
Stub out the route, but remove the conditionals and the existing response:  
import { FrameRequest, getFrameMessage, getFrameHtmlResponse } from '@coinbase/onchainkit';  
import { NextRequest, NextResponse } from 'next/server';  
  
async function getResponse(req: NextRequest): Promise<NextResponse> {  
 let accountAddress: string | undefined = '';  
 let text: string | undefined = '';  
  
 const body: FrameRequest = await req.json();  
 const { isValid, message } = await getFrameMessage(body, { neynarApiKey: 'NEYNAR\_ONCHAIN\_KIT' });  
  
 if (isValid) {  
 accountAddress = message.interactor.verified\_accounts[0];  
 } else {  
 return new NextResponse('Message not valid', { status: 500 });  
 }  
  
 return new NextResponse();  
 // TODO: Return a frame  
}  
  
export async function POST(req: NextRequest): Promise<Response> {  
 return getResponse(req);  
}  
  
export const dynamic = 'force-dynamic';  
Identifying the Sending Frame  
To identify which frame sent the request to the endpoint, you can extract the state sent with the frame data in the request. The first frame will not have this, so you'll have to make an assumption there.  
After the first frame, if the state property is present, it will appear in the message as state, but it will need to be decoded.  
let state = { frame: 'start' };  
  
try {  
 state = JSON.parse(decodeURIComponent(message.state?.serialized));  
} catch (e) {  
 // Note that this error will always be triggered by the first frame  
 console.error(e);  
}  
HyperFrames  
Add a file called hyperframes.ts to the app folder. Import:  
import { getFrameHtmlResponse } from '@coinbase/onchainkit';  
import { NEXT\_PUBLIC\_URL } from './config';  
Next, add an interface for the HyperFrame:  
export type HyperFrame = {  
 frame: string;  
 1: string | ((text: string) => string) | (() => string);  
 2?: string | ((text: string) => string) | (() => string);  
 3?: string | ((text: string) => string) | (() => string);  
 4?: string | ((text: string) => string) | (() => string);  
};  
The interface contains properties to track the frame itself, and the identifier for which frame each button is mapped to. These are all strings, or functions that return strings, because you'll use string identifiers for your frames, and because the frame returned by getFrameHtmlResponse is formatted as a string.  
For the buttons, you can now use:  
A string identifying the frame the button should lead to  
A function that returns a string that also takes a string argument. Use this to pass in the user's address, or the text from the frame  
Any function with no arguments that returns a frame name as a string  
If you need to, update the type here to handle more complex cases.  
Next, add a Record to store your collection of HyperFrames, and a function to add frames to the Record:  
const frames: Record<string, HyperFrame> = {};  
  
export function addHyperFrame(label: string, frame: HyperFrame) {  
 frames[label] = frame;  
}  
Finally, add a function to retrieve and return a HyperFrame by its label and handle the case of an error.  
export function getHyperFrame(frame: string, text: string, button: number) {  
 const currentFrame = frames[frame];  
 const nextFrameIdOrFunction = currentFrame[button as keyof HyperFrame];  
  
 let nextFrameId: string;  
 if (typeof nextFrameIdOrFunction === 'function') {  
 nextFrameId = nextFrameIdOrFunction(text);  
 } else {  
 nextFrameId = nextFrameIdOrFunction as string;  
 }  
  
 if (!frames[nextFrameId]) {  
 throw new Error(`Frame not found: ${nextFrameId}`);  
 }  
  
 return frames[nextFrameId].frame;  
}  
Adding HyperFrames  
You can put the HyperFrames wherever you want and import them into your route. For the sake of simplicity, this demo will simply include them at the top of the route file. Import  
import { addHyperFrame, getHyperFrame } from '../../hyperframes';  
To store the HyperFrames, add them to the Record type with addHyperFrame. To create the frames themselves, use getFrameHtmlResponse to build the frame, and add the names of the frames you have created, or will create, to the appropriate button.  
Add the name of each frame as that frame's state as well.  
addHyperFrame('start', {  
 frame: getFrameHtmlResponse({  
 buttons: [  
 {  
 label: 'Road',  
 },  
 {  
 label: 'Woods',  
 },  
 {  
 label: 'Cave',  
 },  
 {  
 action: 'link',  
 label: 'TODO',  
 target: 'https://www.google.com',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/frame-1-forest.webp`,  
 aspectRatio: '1:1',  
 },  
 state: { frame: 'start' },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
 }),  
 1: 'road',  
 2: 'woods-bear',  
 3: 'cave-1',  
});  
  
addHyperFrame('road', {  
 frame: getFrameHtmlResponse({  
 buttons: [  
 {  
 label: 'Go Back',  
 },  
 {  
 label: 'Shack',  
 },  
 {  
 label: 'Forward',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/road.png`,  
 aspectRatio: '1:1',  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
 }),  
 1: 'start',  
 2: 'shack',  
 3: 'desert-road',  
});  
Above, you've created two HyperFrames. The first, has three buttons, mapped to the frames named road, woods-bear, and cave-1. Only the first will work, because you haven't built the other HyperFrames, or error handling.  
The second, also has three buttons, mapped to frames as well. Only start is implemented as of yet.  
Calling getHyperFrame  
Return to route.ts. To avoid TypeScript errors, you'll need to implement at least a minimum of error handling for the event that the frame query parameter or button are not present:  
if (!frame) {  
 return new NextResponse('Frame not found', { status: 404 });  
}  
  
// There should always be a button number  
if (!message?.button) {  
 return new NextResponse('Button not found', { status: 404 });  
}  
Then, simply import and call getHyperFrame as the NextResponse:  
return new NextResponse(getHyperFrame(frame as string, text || '', message?.button));  
Deploy, test with the Frame Validator, and debug!  
Adding Conditionals  
It's not very interesting for everyone to be able to explore without restriction, so add a lock with a password! To do so, add HyperFrame that contains a function to see if the text contains the correct password.  
addHyperFrame('shack', {  
 frame: getFrameHtmlResponse({  
 buttons: [  
 {  
 label: 'Go Back',  
 },  
 {  
 label: 'Door',  
 },  
 {  
 label: 'Testing',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/shack.png`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: 'What is the password?',  
 },  
 state: { frame: 'shack' },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
 }),  
 1: 'road',  
 2: (text: string) => {  
 return text === 'All your Base are belong to you' ? 'key' : 'shack-bad-password';  
 },  
});  
For the event that the user enters the wrong password, simply add a nearly identical frame asking them to try again. If they enter the correct password, take them to a new room via a different HyperFrame:  
addHyperFrame('shack-bad-password', {  
 frame: getFrameHtmlResponse({  
 buttons: [  
 {  
 label: 'Go Back',  
 },  
 {  
 label: 'Door',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/shack.png`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: 'Try again. What is the password?',  
 },  
 state: { frame: 'shack-bad-password' },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
 }),  
 1: 'road',  
 2: (text: string) => {  
 return text === 'All your Base are belong to you' ? 'key' : 'shack-bad-password';  
 },  
});  
  
addHyperFrame('key', {  
 frame: getFrameHtmlResponse({  
 buttons: [  
 {  
 label: 'Go Back',  
 },  
 {  
 label: 'TODO',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/key.png`,  
 aspectRatio: '1:1',  
 },  
 state: { frame: 'key' },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
 }),  
 1: 'shack',  
});  
Conclusion  
In this tutorial, you learned how to implement a system of HyperFrames - frames that are easily cross-linkable. You also learned how to add variety and depth to this system by adding conditionals for the button linking one frame to another. Finally, you learned how to use the state property to pass information between frames.  
Tags:framesOnchainKit  
ON THIS PAGE  
Prerequisites  
Getting Started  
Vercel  
Farcaster  
Frames  
Creating the First Frame  
Setting up the Route  
HyperFrames  
Identifying the Sending Frame  
Adding HyperFrames  
Calling getHyperFrame  
Adding Conditionals  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/farcaster-frames-nft-minting  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Farcaster Frames: Building an NFT airdrop Frame  
Frames have taken the Farcaster world by storm. They allow you to place a small onchain app inside of a cast! In this tutorial, we'll show you how to build your first frame using OnchainKit, and our a-frame-in-100-lines template.  
Objectives  
By the end of this tutorial you should be able to:  
Build a Farcaster Frame with a mint and airdrop function  
Use the BaseScan API to find the owner of an NFT  
Pull the tokenUri from a contract and use it to show the real art for an NFT  
Prerequisites  
ERC-721 Tokens  
This tutorial assumes that you are able to write, test, and deploy your own ERC-721 tokens using the Solidity programming language. If you need to learn that first, check out our content in Base Learn or the sections specific to ERC-721 Tokens!  
Onchain App Development  
You'll need to be comfortable building onchain apps.  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own.  
Farcaster  
You must have a Farcaster account with a connected wallet. Check out the Base channel to stay in the loop when we release tutorials like this!  
Setup and Testing the Template  
Start by creating a new repo, using a-frame-in-100-lines as a template.  
Run yarn install, then yarn dev.  
In the browser, open http://localhost:3000/. All you'll see is a heading with Zizzamia.xyz. This is expected!  
Open page.tsx. Here, you'll find the initial setup of the metadata that Farcaster reads to create the frame, as well as the simple page you just viewed. Change the heading to use your name.  
Long term, you'd probably want to put something there, as the cast must have a link to this page. Currently, it's pretty common for the community to just skip that part. The cool stuff is in the Frame!  
Deploy your project with Vercel, then open it at the root. Again, you'll see the simple page, now with your name!  
Testing the Cast  
To cast a frame, you'd simply paste this URL into when you create a cast, and it will work automagically!  
But you probably want to use the Frame Validator first. Open that up, and paste in your link. Click Load. You should see:  
Click, and the image will change. You'll also see a new set of buttons, demonstrating some of the features available for frame development.  
Open up api/frame/route.ts. This file handles the route in post\_url that is called when you click the button.  
The route:  
Uses getFrameMessage from onchainkit to validate and interpret the request  
Pulls the user's wallet address, and other useful information, from the message  
Sends a NextResponse containing a new frame, which is then displayed to the user  
Building A Mint Frame  
You can put a transaction in a frame and require users to pay gas to mint your NFT. Another option is to airdrop NFTs to your users, with you paying the gas. This tutorial takes the second approach.  
The strategy uses the contract from the Onchain NFTs tutorial, modified so that a designated address can mint for a provided address:  
function mintTo(address \_recipient) public onlyOwner {  
 if(counter >= maxSupply) {  
 revert MaxSupplyReached();  
 }  
 if(minted[\_recipient]) {  
 revert OneMintPerAddress();  
 }  
 minted[\_recipient] = true;  
 counter++;  
 \_safeMint(\_recipient, counter);  
}  
INFO  
You should also use a mintTo function that accepts an address \_recipient if you are doing a mint transaction button. Doing so helps Farcaster more easily scan your transaction for safety.  
You'll also want to keep track of addresses that have already minted, to prevent a few spammers from claiming all the NFTs!  
DANGER  
Make sure you added .env.local to `.gitignore! If you don't do this you are going to expose your key and lose your wallet!  
Create .env.local and add:  
BASESCAN\_API\_KEY - https://basescan.org/myapikey  
NFT\_CONTRACT\_ADDRESS - As described above  
WALLET\_PRIVATE\_KEY - the private key of the wallet that owns the above NFT contract and can mint it. This key is the same as your 12-word phrase  
Install dotenv with yarn add dotenv, then open route.ts.  
Add a new image in the public folder. Per the Frames docs, images must have a 1.91 to 1, or a 1 to 1 aspect ratio. We used a grey-scale copy of the NFT, but some frame developers prefer to show the full image, or another call to action.  
For now, just use something you'll recognize as a placeholder.  
In app/config.ts, change NEXT\_PUBLIC\_URL to the address where you've deployed your site.  
Change the image link in both files to the new image you added in public. Next automatically makes these available at your-site.vercel.app/your-image.png.  
Redeploy and test the Frame again.  
You'll see the new image, but only in the validator, or for new casts. Any existing casts with the frame will permanently display the old image. These can not be updated.  
The Image Route  
If your NFTs are unique, you'll want to get the picture of the actual NFT once the user mints. If you're not using unique NFTs, skip this part. It's more efficient to keep a local copy of the art in public and save some API calls.  
Add a route to get the image. Right now, this can just return the existing image:  
import { NextResponse } from 'next/server';  
  
export async function GET() {  
 // Temporary, does NOT fetch the correct image for the actual NFT  
 const img = await fetch('https://land-sea-and-sky.vercel.app/lss-bw.png').then((res) =>  
 res.blob(),  
 );  
 return new NextResponse(img, {  
 status: 200,  
 headers: {  
 'Content-Type': 'image/png',  
 'Cache-Control': 'max-age=10',  
 },  
 });  
}  
Update both route.ts and page.tsx to use this route for the image:  
// page.tsx  
const imageUrl = `${NEXT\_PUBLIC\_URL}/api/images/nft`;  
  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 label: 'Story time',  
 },  
 {  
 action: 'tx',  
 label: 'Send Base Sepolia',  
 target: `${NEXT\_PUBLIC\_URL}/api/tx`,  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/tx-success`,  
 },  
 ],  
 image: {  
 src: imageUrl,  
 aspectRatio: '1.91:1',  
 },  
 input: {  
 text: 'Tell me a story',  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
});  
  
export const metadata: Metadata = {  
 title: 'zizzamia.xyz',  
 description: 'LFG',  
 openGraph: {  
 title: 'zizzamia.xyz',  
 description: 'LFG',  
 images: [imageUrl],  
 },  
 other: {  
 ...frameMetadata,  
 },  
};  
// route.ts  
return new NextResponse(  
 getFrameHtmlResponse({  
 buttons: [  
 {  
 label: `State: ${state?.page || 0}`,  
 },  
 {  
 action: 'link',  
 label: 'OnchainKit',  
 target: 'https://onchainkit.xyz',  
 },  
 {  
 action: 'post\_redirect',  
 label: 'Dog pictures',  
 },  
 ],  
 image: {  
 src: imgUrl,  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/frame`,  
 state: {  
 page: state?.page + 1,  
 time: new Date().toISOString(),  
 },  
 }),  
);  
Redeploy and test!  
Setting up the Contract  
Import your deployment artifact, if you're using Hardhat. If you're using Foundry, or another tool, adapt as appropriate to get the ABI and contract address.  
Add your wallet key, and a provider url to your `.env.local, then import them as well.  
import LandSeaSkyNFT from '../constants/LandSeaSkyNFT.json';  
  
require('dotenv').config();  
  
const WALLET\_PRIVATE\_KEY = process.env.WALLET\_PRIVATE\_KEY;  
const PROVIDER\_URL = process.env.PROVIDER\_URL;  
Install viem with yarn add viem.  
Import:  
import { privateKeyToAccount } from 'viem/accounts';  
import { baseSepolia } from 'viem/chains';  
import { createWalletClient, http } from 'viem';  
Create a walletClient from the account, created from the private key, and a publicClient to read from the blockchain.  
const nftOwnerAccount = privateKeyToAccount(WALLET\_PRIVATE\_KEY as `0x${string}`);  
const nftOwnerClient = createWalletClient({  
 account: nftOwnerAccount,  
 chain: baseSepolia,  
 transport: http(PROVIDER\_URL as string),  
});  
  
const publicClient = createPublicClient({  
 chain: baseSepolia,  
 transport: http(PROVIDER\_URL as string),  
});  
Use the address of the Farcaster user, conveniently provided by OnchainKit, to check whether or not the user has minted the NFT:  
let minted = false;  
  
try {  
 minted = !!(await publicClient.readContract({  
 address: LandSeaSkyNFT.address as `0x${string}`,  
 abi: LandSeaSkyNFT.abi,  
 functionName: 'minted',  
 args: [message.address],  
 }));  
} catch (err) {  
 console.error(err);  
}  
The double !! will force the response into a boolean.  
Updating the Frame  
If the address has already minted an NFT, update the button to share a message thanking them, otherwise, try to mint and airdrop an NFT:  
if (minted) {  
 return new NextResponse(  
 getFrameHtmlResponse({  
 buttons: [  
 {  
 label: 'You already minted, thanks!',  
 },  
 ],  
 image: imageUrl,  
 }),  
 );  
} else {  
 // Try to mint and airdrop the NFT  
 try {  
 const { request } = await publicClient.simulateContract({  
 account: nftOwnerAccount,  
 address: LandSeaSkyNFT.address as `0x${string}`,  
 abi: LandSeaSkyNFT.abi,  
 functionName: 'mintTo',  
 args: [message.address],  
 });  
 await nftOwnerClient.writeContract(request);  
 } catch (err) {  
 console.error(err);  
 }  
 if (minted) {  
 return new NextResponse(  
 getFrameHtmlResponse({  
 buttons: [  
 {  
 label: 'Thanks for minting!',  
 },  
 ],  
 image: imageUrl,  
 }),  
 );  
}  
Update your environment variables in Vercel then redeploy and test. You won't see a different image, but you'll be able to see the transactions on BaseScan.  
Showing the Real NFT Image  
Now, it's time to make the frame show the user's NFT, if they have one. Because Land, Sea, and Sky, the sample from [Complex Onchain NFTs] generates a unique NFT for each user, you'll need to use a third-party api to get the user's NFT id, then fetch the tokenURI for that NFT.  
First, change your button on the first page to represent the new use case: "Mint or View NFT".  
Open route.ts for the image endpoint.  
Update it to grab query parameters from your endpoint, and show the grayscale image if either is missing:  
const url = new URL(req.url);  
 const queryParams = url.searchParams;  
  
 const { minted, address } = Object.fromEntries(queryParams.entries());  
 console.log({ minted, address });  
  
  
 if (!minted || !address) {  
 const img = await fetch('https://land-sea-and-sky.vercel.app/lss-bw.png').then((res) => res.blob());  
 return new NextResponse(img, {  
 status: 200,  
 headers: {  
 'Content-Type': 'image/png',  
 'Cache-Control': 'max-age=10',  
 }  
 });  
 } else {  
 // Find out if the address still owns the NFT, and if so, what the token ID is  
  
 }  
}  
  
You'll use the BaseScan api to find the token information. Add your api key to your .env if you didn't already.  
CAUTION  
Note that BaseScan and Etherscan use different keys!  
There isn't an sdk to install.  
Add a folder inside app called types and a file called index.ts inside that. Add the types for etherscan api responses:  
// @dev values taken from sample response  
export type BaseScanEventResponse = {  
 blockNumber: string;  
 timeStamp: string;  
 hash: string;  
 nonce: string;  
 blockHash: string;  
 from: string;  
 contractAddress: string;  
 to: string;  
 tokenID: string;  
 tokenName: string;  
 tokenSymbol: string;  
 tokenDecimal: string;  
 transactionIndex: string;  
 gas: string;  
 gasPrice: string;  
 gasUsed: string;  
 cumulativeGasUsed: string;  
 input: string;  
 confirmations: string;  
};  
  
export type BaseScanResponse = {  
 status: '0' | '1';  
 message: string;  
 result: BaseScanEventResponse[];  
};  
Import these into the image route:  
import { BaseScanResponse } from '../../../types';  
Use the BaseScan API to get a list of ERC721 Transfer Events for the address for the contract, filter that list to remove any NFT ids that were given away.  
if (!minted || !address) {  
 // Send the grayscale image  
} else {  
 // Find out of the address still owns the NFT, and if so, what the token ID is  
 const API\_URL = `https://api-sepolia.basescan.org/api?module=account&action=tokennfttx&contractaddress=${LandSeaSkyNFT.address}&address=${address}&sort=asc&apikey=${BASESCAN\_API\_KEY}`;  
 const response = await fetch(API\_URL);  
 const json: BaseScanResponse = (await response.json()) as BaseScanResponse;  
  
 const result = json.result;  
  
 // Create a list of the token IDs where to matches the address  
 const tokenIds = result  
 .filter((tx) => tx.to === address.toLocaleLowerCase())  
 .map((tx) => tx.tokenID);  
  
 // Create a list of the token IDs where from matches the address  
 const tokenIdsFrom = result  
 .filter((tx) => tx.from === address.toLocaleLowerCase())  
 .map((tx) => tx.tokenID);  
  
 // Remove any token IDS that are in tokenIdsFrom from tokenIds  
 const tokenIdsTo = tokenIds.filter((tokenId) => !tokenIdsFrom.includes(tokenId));  
}  
Add another image, to handle users that have minted your NFT, but don't anymore. We used a sad whale:  
if (tokenIdsTo.length === 0) {  
 const img = await fetch('https://land-sea-and-sky.vercel.app/gave-me-away.png').then((res) =>  
 res.blob(),  
 );  
 return new NextResponse(img, {  
 status: 200,  
 headers: {  
 'Content-Type': 'image/png',  
 'Cache-Control': 'max-age=10',  
 },  
 });  
} else {  
 // Get the actual NFT image from the contract  
}  
Otherwise, create a viem public client and call tokenURI with the id of the first token on the list  
Note: This is not an elegant solution for if someone is gifted an NFT then mints their own!  
You did this already in api/frame. Use the same pattern, but instead call tokenURI with the token id:  
// Get the actual NFT image from the contract  
const publicClient = createPublicClient({  
 chain: baseSepolia,  
 transport: http(PROVIDER\_URL as string),  
});  
  
let tokenMetadata = '';  
try {  
 tokenMetadata = (await publicClient.readContract({  
 address: LandSeaSkyNFT.address as `0x${string}`,  
 abi: LandSeaSkyNFT.abi,  
 functionName: 'tokenURI',  
 args: [tokenIdsTo[0]],  
 })) as string;  
} catch (err) {  
 console.error(err);  
}  
  
console.log({ tokenMetadata });  
The tokenURI for the sample contract returns the actual json metadata, json encoded, with the actual svg inside, also base64 encoded. You'll need to decode it, then return it.  
// Decode the base64 encoded JSON  
const tokenMetadataJson = JSON.parse(atob(tokenMetadata.split(',')[1]));  
  
// Then decode the base64 encoded svg image  
const svg = atob(tokenMetadataJson.image.split(',')[1]);  
  
// Create a blob from the svg  
const img = new Blob([svg], { type: 'image/svg+xml' });  
  
// Return the blob  
return new NextResponse(img, {  
 status: 200,  
 headers: {  
 'Content-Type': 'image/svg+xml',  
 'Cache-Control': 'max-age=10',  
 },  
});  
This works!  
Adjusting the Image Ratio  
Farcaster Frames now allow a 1:1 image ratio, but they used to only accept a 1:91 to 1 ratio. You can skip this if you don't want the classic look.  
If you do want to go old-school, you'll need to adjust the svg, and clip the extra content, since the svg has pieces that are intended to be out of frame:  
function frameSvgStringToBlob(originalSvgString: string): Blob {  
 // Define the dimensions based on the 1.91:1 aspect ratio  
 const originalSize = 1024;  
 const newWidth = originalSize \* 1.91; // 1955.84  
 const centerX = newWidth / 2;  
 const clipStartX = centerX - 512; // Start of the clip rectangle  
  
 // Create the new SVG string and add a clipPath to clip the contents  
 const framedSvgString = `  
 <svg width="${newWidth}" height="${originalSize}" viewBox="0 0 ${newWidth} ${originalSize}" xmlns="http://www.w3.org/2000/svg">  
 <defs>  
 <clipPath id="clip">  
 <!-- Set the clipping rectangle to start at clipStartX and be 1024 units wide -->  
 <rect x="${clipStartX}" width="1024" height="${originalSize}" />  
 </clipPath>  
 </defs>  
 <g clip-path="url(#clip)">  
 ${originalSvgString}  
 </g>  
 </svg>  
 `;  
  
 // Convert the string to a blob  
 const blob = new Blob([framedSvgString], { type: 'image/svg+xml' });  
  
 return blob;  
}  
Call your new function, then return it:  
const img = new Blob([frameSvgStringToBlob(svg)], { type: 'image/svg+xml' });  
// Return the blob  
return new NextResponse(img, {  
 status: 200,  
 headers: {  
 'Content-Type': 'image/svg+xml',  
 'Cache-Control': 'max-age=10',  
 },  
});  
CAUTION  
SVG images aren't working in Frames on mobile clients. Stay tuned on the Base channel for a fix! Or add a library here to return the image as a .png instead of a .svg.  
Remember to update your envars in Vercel, redeploy, and test. You'll now see your NFT in the frame!  
Conclusion  
In this tutorial, you learned how to create Farcaster frames. You then updated it to airdrop an NFT to users who click the button in your frame. Finally, you learned how to pull real NFT images from the blockchain, and display them in your frames.  
Tags:framesnftOnchainKit  
ON THIS PAGE  
Prerequisites  
Setup and Testing the Template  
Testing the Cast  
Onchain App Development  
Vercel  
Farcaster  
Building A Mint Frame  
The Image Route  
Setting up the Contract  
The Image Route  
Updating the Frame  
Showing the Real NFT Image  
Conclusion  
Adjusting the Image Ratio  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/farcaster-frames-nocode-minting  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Farcaster Frames: Building a no-code minting Frame  
Frames on Farcaster allow you to add buttons that link to external sites in a new window. In this tutorial, you'll learn how to build a frame with OnchainKit that links to an NFT mint on mint.fun, a no-code platform for creating and sharing NFT collections. If you're new to frames and not ready to write a smart contract, this is an excellent starting point! If you've already made a few, this one does showcase the latest features, so check it out and make sure you haven't missed anything.  
Objectives  
By the end of this tutorial you should be able to:  
Build a Frame with an external link button  
Use mint.fun to set up a no-code NFT mint  
Prerequisites  
ERC-721 Tokens  
This tutorial assumes that you are familiar with ERC-721 tokens (NFTs). You do not need to be able to write a smart contract. You'll use an online tool for that part.  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. If you need a refresher, check out our tutorial on deploying with Vercel!  
Farcaster  
You must have a Farcaster account with a connected wallet. Check out the Base channel to stay in the loop when we release tutorials like this!  
Setup  
Create a copy of a-frame-in-100-lines If you're working off of an existing copy, check to make sure you have the most up-to-date version of OnchainKit!  
Run yarn install.  
Frames need to be deployed to a server to work, so connect your repo to Vercel with CI/CD. Remember that while doing so does make it much easier to test and redeploy, it also makes it easier to break a live frame!  
Open app/config.ts and update NEXT\_PUBLIC\_URL to your new deployment.  
Creating the Initial Frame  
As always, you'll set up the first frame in app/page.tsx. Open that up, and change the example to have a single button. You can keep the example that links to Google:  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 action: 'link',  
 label: 'Link to Google',  
 target: 'https://www.google.com',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/park-3.png`,  
 aspectRatio: '1:1',  
 },  
});  
Commit and push to deploy the frame. Wait for Vercel to build, then test with the Frame Validator. You should see a frame with a single button, that opens a new window once the warning is passed:  
Setting up the Mint  
Before you can link to your mint, you need to set it up! Navigate to mint.fun. Create an account or log in, then click the + button in the upper right corner by the wallet balance.  
Give your project a snappy name and upload a picture. Unless you're an experienced NFT artist, the default settings are probably best. Note that the mint will only run for a fixed time, so you may want to do the first pass with a test name and asset.  
It will cost a small amount of gas to create the project. Approve the transaction, then you'll be taken to your mint page.  
Copy the link, and update your button, and change the frame to use the image that you uploaded for your NFT:  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 action: 'link',  
 label: 'Mint',  
 target: 'https://mint.fun/base/<your contract address>',  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/2024-a-base-odyssey.png`,  
 aspectRatio: '1:1',  
 },  
});  
Deploy and test, now your button links to your NFT!  
Next Steps  
Your simple mint is ready to use on Farcaster! Be sure to check out our other tutorials if you want to add more advanced behavior, or mint with your own contract.  
Conclusion  
In this tutorial, you learned how to make a simple Frame on Farcaster that is tied to a mint.fun NFT mint!  
Tags:framesnftOnchainKit  
ON THIS PAGE  
Prerequisites  
Setup  
ERC-721 Tokens  
Vercel  
Farcaster  
Creating the Initial Frame  
Setting up the Mint  
Next Steps  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/farcaster-frames-transactions  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Farcaster Frames: Making transactions  
Frames on Farcaster support wallet transactions invoked directly from the buttons in a Frame! OnchainKit supports this feature. In this tutorial, you'll learn how to set up a frame that will allow your users to complete a simple transaction.  
Objectives  
By the end of this tutorial, you should be able to:  
Build a Farcaster Frame that invokes a smart contract transaction from a button  
Prerequisites  
Onchain App Development  
You'll need to be comfortable building onchain apps.  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. Check out our tutorial on deploying with Vercel if you need a refresher!  
Farcaster  
You must have a Farcaster account with a connected wallet. Check out the Base channel to stay in the loop when we release tutorials like this!  
Frames  
You should be comfortable with the basics of creating Farcaster Frames. If you aren't, check out our tutorials for how to build a no-code NFT mint Frame or NFT airdrop Frame.  
Smart Contracts  
You'll need a smart contract with at least one public function that is not pure or view. It needs to change state, so that it costs gas to execute. You can use your own, or the one that we've provided below.  
An Overview of the Transaction Process  
As outlined in the Public Draft V2, transactions in frames work through a multi-step process between the frame developer's endpoint, and the Warpcast (or other) app. A slightly more detailed breakdown of the process:  
Any frame can have a button with 'tx' assigned as the action. If this is the case, the target must be a URL for an endpoint that can process the request, and return calldata for an onchain transaction  
When the user clicks the button, a POST request is sent to the endpoint with the usual set of data you get from the frame  
The endpoint uses that data, such as the user's attached wallet address, to build a transaction and send it back to the Farcaster client app  
The app then redirects the user to their wallet with the provided calldata, which opens the normal flow for the user to review and approve or deny a transaction  
If successful, the wallet returns the user to the app with the transaction id  
The app then automatically makes another POST request to the same target, this time with the transaction id. Make sure you handle this second call appropriately!  
Setting up the Smart Contract  
Before you can build a frame that calls a smart contract transaction, you need to have a smart contract! You can use your own, deploy your own copy of our Click the Button smart contract, or use our deployment.  
The contract contains a simple game where players can "click the button" and pay a few cents worth of ether to get a point. That's it! No rewards, no complexity, just a tempting big red button, and a leaderboard that can be retrieved unsorted.  
function clickTheButton()  
public and payable  
Cost is 10000 gwei, or 0.00001 ether  
Adds 1 point to the score of msg.sender  
function getMyClicks()  
public  
Returns the number of clicks for msg.sender  
function getAllClicks()  
public  
Returns an unsorted list of each address and their score  
function withdraw()  
public and onlyOwner  
Withdraw all funds to the owner address  
We've deployed an instance of the contract on testnet, and on mainnet. You can use this if you like.  
Building the Frame  
Start a new project using a-frame-in-100-lines as a template or guide, or open an existing one. Make sure the version of OnchainKit is current.  
The First Frame and Page  
Add a new page to the Next.js App Router by adding a new folder in app called buttonclicker and a file called page.tsx inside the new folder. Doing so will automatically place the new page at yourappname.vercel.app/buttonclicker.  
Using the sample page as a guide, set up a new frame, and stub for a new page:  
import { getFrameMetadata } from '@coinbase/onchainkit/frame';  
import type { Metadata } from 'next';  
import { NEXT\_PUBLIC\_URL } from './config';  
  
const frameMetadata = getFrameMetadata({  
 buttons: [  
 {  
 action: 'tx',  
 label: 'Click the Button',  
 target: `${NEXT\_PUBLIC\_URL}/api/buttonclicker`,  
 },  
 {  
 action: 'link',  
 label: 'Leaderboard',  
 target: `${NEXT\_PUBLIC\_URL}/buttonclicker`,  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/button.webp`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: "Don't click the button!",  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/aftertx`,  
});  
  
export const metadata: Metadata = {  
 title: 'Click the Button',  
 description: "Don't click the button!",  
 openGraph: {  
 title: 'Click the Button',  
 description: "Don't click the button!",  
 images: [`${NEXT\_PUBLIC\_URL}/button.webp`],  
 },  
 other: {  
 ...frameMetadata,  
 },  
};  
  
export default function Page() {  
 return (  
 <>  
 <h1>Leaderboard</h1>  
 <p>TODO</p>  
 </>  
 );  
}  
A few notes:  
The naming conventions and organization of OnchainKit are evolving as frames evolve. Check the OnchainKit repo and docs if the imports don't work  
You'll make the api endpoint for the button clicker game next  
Feel free to adjust the text. We're just having fun by making it give conflicting instructions  
The button to show the leaderboard simply goes to your page. You could render an svg to png in the frame endpoint to show it in frames as well  
You'll have to do the leaderboard on your own. Check out the frontend content in Base Learn if you need a hand learning how to interact with your contracts!  
Adding the Transaction Endpoints  
The transaction endpoints must handle two scenarios:  
A POST request to the endpoint providing the user's information and expecting transaction calldata in return  
A POST to the postUrl endpoint after the transaction is successful, containing the transaction id. The second POST should return a valid frame  
These can be the same or different endpoints  
Setting up the Transaction Endpoint  
Add a new folder called buttonclicker containing a file called route.ts inside the api folder of your app router. This will automatically create a new route at https://yourapp.vercel.app/buttonclicker.  
You'll need to import the standard Frames functions you've been using, as well as some utilities from viem. You'll also need a new type from OnchainKit.  
import { FrameRequest, getFrameMessage, getFrameHtmlResponse } from '@coinbase/onchainkit/frame';  
import { NextRequest, NextResponse } from 'next/server';  
import { encodeFunctionData, formatEther, parseGwei } from 'viem';  
import { base } from 'viem/chains';  
import type { FrameTransactionResponse } from '@coinbase/onchainkit/frame';  
Finally, you'll need to import the ABI and address for your contract. If you're using a tool that exports the ABI as an object, you can add it as below after adding the folder and file for the ABI. Make sure you have const abi = before the array containing the ABI, and export default abi;.  
You also need to add the contract address to config.ts.  
If you're using Hardhat json artifacts, add those to your project and import from there.  
import ClickTheButtonABI from '../../\_contracts/ClickTheButtonAbi';  
import { CLICK\_THE\_BUTTON\_CONTRACT\_ADDR } from '../../config';  
The getResponse function works similar to other frames. Stub it out first:  
async function getResponse(req: NextRequest): Promise<NextResponse | Response> {  
 const body: FrameRequest = await req.json();  
 const { isValid } = await getFrameMessage(body, { neynarApiKey: 'NEYNAR\_ONCHAIN\_KIT' });  
  
 if (!isValid) {  
 return new NextResponse('Message not valid', { status: 500 });  
 }  
  
 // TODO  
}  
  
export async function POST(req: NextRequest): Promise<Response> {  
 return getResponse(req);  
}  
  
export const dynamic = 'force-dynamic';  
To begin, you'll need to build the data for the transaction. You'll use encodeFunctionData from viem to do this, the same as any other onchain app using viem:  
const data = encodeFunctionData({  
 abi: ClickTheButtonABI,  
 functionName: 'clickTheButton',  
});  
Next, use FrameTransactionResponse from OnchainKit to build the response the app expects:  
const txData: FrameTransactionResponse = {  
 chainId: `eip155:${base.id}`,  
 method: 'eth\_sendTransaction',  
 params: {  
 abi: [],  
 data,  
 to: CLICK\_THE\_BUTTON\_CONTRACT\_ADDR,  
 value: parseGwei('10000').toString(), // 0.00001 ETH  
 },  
};  
Finally, return the transaction as a NextResponse:  
return NextResponse.json(txData);  
INFO  
If you find Warpcast errors or spins forever after receiving your transaction data, it can be handy to simulate the transaction on your service first. It makes debugging much easier and will rule out any errors in forming the transaction arguments. To learn how, check out viem's documentation on Simulating Contract Interactions.  
Setting Up the After Transaction Endpoint  
You can use a different postUrl to separate concerns with generating the transaction, and returning a frame after. You've already named this endpoint aftertx in the first frame's postUrl. Add it now to the api folder, and open aftertx/route.ts.  
In this case, simply return the original frame with slightly updated buttons and text:  
import { FrameRequest, getFrameMessage, getFrameHtmlResponse } from '@coinbase/onchainkit/frame';  
import { NextRequest, NextResponse } from 'next/server';  
import { NEXT\_PUBLIC\_URL } from '../../config';  
  
async function getResponse(req: NextRequest): Promise<NextResponse> {  
 const body: FrameRequest = await req.json();  
 const { isValid, message } = await getFrameMessage(body, { neynarApiKey: 'NEYNAR\_ONCHAIN\_KIT' });  
  
 if (!isValid) {  
 return new NextResponse('Message not valid', { status: 500 });  
 }  
  
 return new NextResponse(  
 getFrameHtmlResponse({  
 buttons: [  
 {  
 action: 'tx',  
 label: 'Click Again!',  
 target: `${NEXT\_PUBLIC\_URL}/api/buttonclicker`,  
 },  
 {  
 action: 'link',  
 label: 'Leaderboard',  
 target: `${NEXT\_PUBLIC\_URL}/buttonclicker`,  
 },  
 ],  
 image: {  
 src: `${NEXT\_PUBLIC\_URL}/button.webp`,  
 aspectRatio: '1:1',  
 },  
 input: {  
 text: 'Noooo, why did you click!?',  
 },  
 postUrl: `${NEXT\_PUBLIC\_URL}/api/aftertx`,  
 }),  
 );  
}  
  
export async function POST(req: NextRequest): Promise<Response> {  
 return getResponse(req);  
}  
  
export const dynamic = 'force-dynamic';  
INFO  
In certain applications you might want to monitor the status of the transaction at this point. It's possible it fails, takes a while, and/or you may want to do another operation only after it has been confirmed. To do so you can make use of message.transaction.hash and build a frame flow that checks the status of the transaction by fetching the transaction receipt.  
Conclusion  
In this tutorial, you've learned how to invoke a smart contract transaction from a frame! You used it to implement a simple game.  
Click the Button Source Code  
This contract uses Remix-style imports. You'll need to update them for other toolchains!  
// SPDX-License-Identifier: MIT  
  
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/EnumerableMap.sol";  
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol";  
  
pragma solidity 0.8.20;  
  
error WrongValue();  
  
contract ClickTheButton is Ownable {  
 uint constant COST = 10000 gwei;  
 using EnumerableMap for EnumerableMap.AddressToUintMap;  
  
 EnumerableMap.AddressToUintMap private timesClicked;  
  
 constructor() Ownable(msg.sender) {}  
  
 struct Clicker {  
 address user;  
 uint clicks;  
 }  
  
 function clickTheButton() public payable {  
 if(msg.value != COST) {  
 revert WrongValue();  
 }  
 (, uint clicks) = timesClicked.tryGet(msg.sender);  
 timesClicked.set(msg.sender, clicks+1);  
 }  
  
 function getMyClicks() public view returns (uint) {  
 (, uint clicks) = timesClicked.tryGet(msg.sender);  
 return clicks;  
 }  
  
 function getAllClicks() public view returns (Clicker[] memory) {  
 address[] memory keys = timesClicked.keys();  
 Clicker[] memory allClicks = new Clicker[](keys.length);  
  
 for(uint i = 0; i < keys.length; i++) {  
 address user = address(keys[i]);  
 allClicks[i] = Clicker(user, timesClicked.get(user));  
 }  
  
 return allClicks;  
 }  
  
 function withdraw() public onlyOwner {  
 payable(owner()).transfer(address(this).balance);  
 }  
}  
Tags:framesOnchainKit  
ON THIS PAGE  
Prerequisites  
An Overview of the Transaction Process  
Onchain App Development  
Vercel  
Farcaster  
Frames  
Smart Contracts  
Setting up the Smart Contract  
Building the Frame  
Adding the Transaction Endpoints  
Setting up the Transaction Endpoint  
Conclusion  
Setting up the Transaction Endpoint  
Setting Up the After Transaction Endpoint  
Click the Button Source Code  
Click the Button Source Code  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/gasless-transaction-on-base-using-a-paymaster  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Gasless Transactions on Base using Base Paymaster  
Still trying to onboard users to your app? Want to break free from the worries of gas transactions and sponsor them for your users on Base? Look no further!  
Base transaction fees are less than a penny, but the concept of gas can be confusing for new users. Abstract this away and improve your UX by using the Base Paymaster. The Base Paymaster allows you to batch multi-step transactions and create custom gasless experiences. Sponsor up to $10k monthly on mainnet (unlimited on testnet). To request an increase in limit, reach out in Discord.  
Objectives  
Configure security measures to ensure safe and reliable transactions  
Manage and allocate resources for sponsored transactions  
Subsidize transaction fees for users, enhancing the user experience by making transactions free for them  
Set up and manage sponsored transactions on various schedules, including weekly, monthly, and daily cadences  
Prerequisites  
This tutorial assumes you have a Coinbase Cloud Developer Platform account. If not, sign up on the CDP site.  
Coinbase CDP account  
This is your access point to the Coinbase Cloud Developer Platform, where you can manage projects and utilize tools like the Paymaster.  
Familiarity with Smart Accounts and ERC 4337  
Understanding Smart Accounts and the ERC 4337 standard is crucial as they are the backbone of executing advanced transaction patterns and account abstractions on the Ethereum network.  
Foundry  
Foundry is a development environment, testing framework, and smart contract toolkit for Ethereum. It's essential for deploying and testing smart contracts.  
Set Up a Base Paymaster & Bundler  
In this section, you will configure a Paymaster to sponsor payments on behalf of a specific smart contract for a specified amount. First, navigate to the Coinbase Developer Platform, create or select your project, and click on the Paymaster tool from the left navigation. Then, go to the Configuration tab and save the RPC URL to your clipboard, which will be needed for later steps in your index.js file.  
Navigate to the Coinbase Developer Platform:  
Create or select your project of choice from the upper left corner of your screen.  
Click on the Paymaster tool on the left navigation:\*\* Paymaster Tool  
Click on Configuration at the top of the screen  
Save the RPC URL to your paymaster to your clipboard. You will need it in your index.js file in a later step.  
Allowlist a Sponsorable Contract  
Sponsoring transactions are beneficial to easing the onboarding and UX of a decentralized application. As a developer, you want to ensure this is done in the most secure and cost-effective manner. Start by allowlisting a contract of your choice.  
Select Base Mainnet  
From the configuration page, select Base Mainnet from the dropdown menu. Then, enable your paymaster by clicking on the toggle button to the right of the screen.  
Allowlist the NFT contract and the mintTo functions:  
Click the Add button to add a contract.  
Add the following contract: 0x83bd615eb93eE1336acA53e185b03B54fF4A17e8  
Put mintTo(address) as the function to allowlist then click Save at the bottom of the page.  
USE YOUR OWN CONTRACT  
You will be using this simple NFT contractdeployed on Base mainnet for our example. Feel free to use a contract of your choice.  
Global & Per User Limits:  
Scroll down to the “Per User Limit” section  
You can set a Paymaster policy that specifies either a specific dollar amount or a number of UserOperations. You can enable a 'limit cycle' that allows this policy to refresh every week. This feature enables applications to sponsor smart wallets on a weekly, daily, or monthly basis, either by amount or by number of UserOperations.  
Set the max USD to $0.05 and the max UserOperation to 1 to create a policy with a maximum sponsorship limit of $0.05 and a maximum of 1 UserOperation per user.  
LIMIT CYCLES  
Limit Cycles enables applications to sponsor smart wallets on a weekly, daily, or monthly basis, either by amount or by number of UserOperations. These limits reset based on the cadence selected in the dropdown menu.  
Set the Global limit:  
This setting allows you to define the maximum amount of gas or USD that can be sponsored globally across all users. It helps control the total expenditure on gas sponsorship, ensuring that the allocated budget is not exceeded. For example, setting a global limit of $1 means that the Paymaster will sponsor transactions until the total gas cost reaches $1.  
Set your global policy to be $.07 by entering the amount in the text field and clicking the Save button.  
Test Your Paymaster policy  
You will now test the policy that was created.  
THE USE OF FOUNDRY  
In this tutorial, You use foundry to create two key pairs that will allow us to create Smart Accounts that you will sponsor transactions on behalf of. You will not need to send any funds (ETH) to these wallets. You may use private keys of wallets you own as an alternative.  
Foundry depends on Rust to work. If you do not have rust install:  
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh  
For more information, see the Foundry Book installation guide.  
Open a terminal and install Foundry by running:  
curl -L https://foundry.paradigm.xyz | bash  
Install the Foundry toolchain installer:  
foundryup  
Create a directory named sponsored\_transactions:  
mkdir sponsored\_transactions  
Change into the new directory, initialize a node project, install two dependencies viem and permissionless and create a file titled index.js:  
cd sponsored\_transactions  
npm init es6  
npm install permissionless  
npm install viem  
touch index.js  
In the index.js file, import the dependencies from viem and permissionless. These dependencies will allow us to connect to Base, create our smart accounts, initialize a paymaster, and send our encoded data to the network.  
import { http, createPublicClient, encodeFunctionData } from 'viem';  
import { base } from 'viem/chains';  
import { createSmartAccountClient } from 'permissionless';  
import { privateKeyToSimpleSmartAccount } from 'permissionless/accounts';  
import { createPimlicoPaymasterClient } from 'permissionless/clients/pimlico';  
Set Constants for Your Paymaster & Bundler endpoint  
FIND YOUR PAYMASTER & BUNDLER ENDPOINT  
The Paymaster & Bundler endpoint is the URL for your Coinbase Developer Platform (CDP) Paymaster.  
This was saved in the previous section and follows this format: https://api.developer.coinbase.com/rpc/v1/base/<SPECIAL-KEY>  
Navigate to the Paymaster Tool and select the Configuration tab at the top of the screen to obtain your RPC URL.  
SECURE YOUR ENDPOINTS  
You will create a constant for our Paymaster & Bundler endpoint obtained from cdp.portal.coinbase.com. The most secure way to do this is by using a proxy. For the purposes of this demo, hardcode it into our index.js file. For product, we highly recommend using a proxy service.  
You will also need the address of the entrypoint contract for Base. A full list of entrypoint contracts and their addresses can be found here.  
Add the following to lines of code after your import statements:  
const rpcUrl = https://api.developer.coinbase.com/rpc/v1/base/<SPECIAL-KEY> //Paymaster & Bundler endpoint  
const baseEntryPoint = '0x5FF137D4b0FDCD49DcA30c7CF57E578a026d2789';  
const baseFactoryAddress = '0x15Ba39375ee2Ab563E8873C8390be6f2E2F50232';  
Now, create a public client. Public clients enable our application to interact with the Ethereum blockchain. They use JSON-RPC API methods to perform actions such as retrieving block numbers, transactions, and reading from smart contracts. Learn more about this on viem.  
Initialize a public client named publicClient and set the chain to base and the transport to our rpcURL variable:  
const publicClient = createPublicClient({  
 chain: base,  
 transport: http(rpcUrl),  
});  
Create Two Smart Accounts  
Now, you are going to create two SimpleAccounts for this demonstration. Collectively, these two accounts will help us test the security policies you set at both the global level and the per-account user operation level.  
To do so, you'll first need two private keys that will be used to create the Smart Accounts using Foundry. Open a terminal and run the following command twice:  
cast wallet new  
You will see something like this:  
> cast wallet new  
Successfully created new keypair.  
Address: 0xD440D74620542...D6F005cfD9  
Private key: 0x01c9720c1dfa3c9...634793138897  
  
  
> cast wallet new  
Successfully created new keypair.  
Address: 0x5f8e5bC8620542...D6F005cfD9  
Private key: 0xbcd6fbc1dfa3c9...634793138897  
Use the newly generated private keys as the parameter to the privateKeyToSimpleSmartAccount function and set the factory address to  
SECURE YOUR PRIVATE KEYS  
Be sure to store your private keys somewhere safe. Committing these to a public code repository will give anyone access to your smart accounts.  
Saving them to a .env file is safest.  
const simpleAccount = await privateKeyToSimpleSmartAccount(publicClient, {  
 privateKey: '<Your first key>',  
 factoryAddress: baseFactoryAddress, //Base v0.6  
 entryPoint: baseEntryPoint,  
});  
  
const simpleAccount2 = await privateKeyToSimpleSmartAccount(publicClient, {  
 privateKey: '<Your second key>',  
 factoryAddress: baseFactoryAddress, //Base v0.6  
 entryPoint: baseEntryPoint,  
});  
Let’s break down what’s happening here: Private key : the private key to the wallet that your created either using Foundry or a wallet that you own. Either is fine.  
Factory address is the address to the smart account factory deployed on base. Account factories are smart contracts that facilitate the creation of new wallet contracts. You make view a more comprehensive list of factory addresses on Alchemy.  
Entrypoint is the entrypoint contract for Base.  
FIND THE CORRECT ENTRYPOINT ADDRESS  
Make an JSON RPC request to a node using the `` method to get the correct entrpoint contract. Here's an example for Base using a Coinbase Base Node:  
curl --request POST \  
 --url https://api.developer.coinbase.com/rpc/v1/base/<Your-Key> \  
 --header 'Accept: application/json' \  
 --header 'Content-Type: application/json' \  
 --data '{"id": 1, "jsonrpc": "2.0", "method": "eth\_supportedEntryPoints", "params": []}'  
Initialize Paymaster and Create Accounts  
Initialize the paymaster and smart account client for both smart accounts:  
const cloudPaymaster = createPimlicoPaymasterClient({  
 chain: base,  
 transport: http(rpcUrl),  
 entryPoint: baseEntryPoint,  
});  
  
const smartAccountClient = createSmartAccountClient({  
 account: simpleAccount,  
 chain: base,  
 bundlerTransport: http(rpcUrl),  
 middleware: {  
 sponsorUserOperation: cloudPaymaster.sponsorUserOperation,  
 },  
});  
  
const smartAccountClient2 = createSmartAccountClient({  
 account: simpleAccount2,  
 chain: base,  
 bundlerTransport: http(rpcUrl),  
 middleware: {  
 sponsorUserOperation: cloudPaymaster.sponsorUserOperation,  
 },  
});  
To sponsor transactions you will need the ABI and the address of the contracts you want to support.  
SPONSOR ANY CONTRACT  
Feel free to use your own contract to interact with the Paymaster. For learning purposes, you will interact with an ERC-721 contract deployed on the Base mainnet. You will call the mintTo function, which takes an address as its only parameter.  
You will be interacting with the NFT + ABI from a simple NFT contract deployed at: 0x83bd615eb93eE1336acA53e185b03B54fF4A17e8  
Copy and paste the NFT's abi into index.js  
const abi = [  
 {  
 type: 'constructor',  
 inputs: [  
 { name: '\_name', type: 'string', internalType: 'string' },  
 { name: '\_symbol', type: 'string', internalType: 'string' },  
 ],  
 stateMutability: 'nonpayable',  
 },  
 {  
 type: 'function',  
 name: 'approve',  
 inputs: [  
 { name: 'spender', type: 'address', internalType: 'address' },  
 { name: 'id', type: 'uint256', internalType: 'uint256' },  
 ],  
 outputs: [],  
 stateMutability: 'nonpayable',  
 },  
 {  
 type: 'function',  
 name: 'balanceOf',  
 inputs: [{ name: 'owner', type: 'address', internalType: 'address' }],  
 outputs: [{ name: '', type: 'uint256', internalType: 'uint256' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'currentTokenId',  
 inputs: [],  
 outputs: [{ name: '', type: 'uint256', internalType: 'uint256' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'getApproved',  
 inputs: [{ name: '', type: 'uint256', internalType: 'uint256' }],  
 outputs: [{ name: '', type: 'address', internalType: 'address' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'isApprovedForAll',  
 inputs: [  
 { name: '', type: 'address', internalType: 'address' },  
 { name: '', type: 'address', internalType: 'address' },  
 ],  
 outputs: [{ name: '', type: 'bool', internalType: 'bool' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'mintTo',  
 inputs: [{ name: 'recipient', type: 'address', internalType: 'address' }],  
 outputs: [{ name: '', type: 'uint256', internalType: 'uint256' }],  
 stateMutability: 'payable',  
 },  
 {  
 type: 'function',  
 name: 'name',  
 inputs: [],  
 outputs: [{ name: '', type: 'string', internalType: 'string' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'ownerOf',  
 inputs: [{ name: 'id', type: 'uint256', internalType: 'uint256' }],  
 outputs: [{ name: 'owner', type: 'address', internalType: 'address' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'safeTransferFrom',  
 inputs: [  
 { name: 'from', type: 'address', internalType: 'address' },  
 { name: 'to', type: 'address', internalType: 'address' },  
 { name: 'id', type: 'uint256', internalType: 'uint256' },  
 ],  
 outputs: [],  
 stateMutability: 'nonpayable',  
 },  
 {  
 type: 'function',  
 name: 'safeTransferFrom',  
 inputs: [  
 { name: 'from', type: 'address', internalType: 'address' },  
 { name: 'to', type: 'address', internalType: 'address' },  
 { name: 'id', type: 'uint256', internalType: 'uint256' },  
 { name: 'data', type: 'bytes', internalType: 'bytes' },  
 ],  
 outputs: [],  
 stateMutability: 'nonpayable',  
 },  
 {  
 type: 'function',  
 name: 'setApprovalForAll',  
 inputs: [  
 { name: 'operator', type: 'address', internalType: 'address' },  
 { name: 'approved', type: 'bool', internalType: 'bool' },  
 ],  
 outputs: [],  
 stateMutability: 'nonpayable',  
 },  
 {  
 type: 'function',  
 name: 'supportsInterface',  
 inputs: [{ name: 'interfaceId', type: 'bytes4', internalType: 'bytes4' }],  
 outputs: [{ name: '', type: 'bool', internalType: 'bool' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'symbol',  
 inputs: [],  
 outputs: [{ name: '', type: 'string', internalType: 'string' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'tokenURI',  
 inputs: [{ name: 'id', type: 'uint256', internalType: 'uint256' }],  
 outputs: [{ name: '', type: 'string', internalType: 'string' }],  
 stateMutability: 'view',  
 },  
 {  
 type: 'function',  
 name: 'transferFrom',  
 inputs: [  
 { name: 'from', type: 'address', internalType: 'address' },  
 { name: 'to', type: 'address', internalType: 'address' },  
 { name: 'id', type: 'uint256', internalType: 'uint256' },  
 ],  
 outputs: [],  
 stateMutability: 'nonpayable',  
 },  
 {  
 type: 'event',  
 name: 'Approval',  
 inputs: [  
 {  
 name: 'owner',  
 type: 'address',  
 indexed: true,  
 internalType: 'address',  
 },  
 {  
 name: 'spender',  
 type: 'address',  
 indexed: true,  
 internalType: 'address',  
 },  
 { name: 'id', type: 'uint256', indexed: true, internalType: 'uint256' },  
 ],  
 anonymous: false,  
 },  
 {  
 type: 'event',  
 name: 'ApprovalForAll',  
 inputs: [  
 {  
 name: 'owner',  
 type: 'address',  
 indexed: true,  
 internalType: 'address',  
 },  
 {  
 name: 'operator',  
 type: 'address',  
 indexed: true,  
 internalType: 'address',  
 },  
 { name: 'approved', type: 'bool', indexed: false, internalType: 'bool' },  
 ],  
 anonymous: false,  
 },  
 {  
 type: 'event',  
 name: 'Transfer',  
 inputs: [  
 { name: 'from', type: 'address', indexed: true, internalType: 'address' },  
 { name: 'to', type: 'address', indexed: true, internalType: 'address' },  
 { name: 'id', type: 'uint256', indexed: true, internalType: 'uint256' },  
 ],  
 anonymous: false,  
 },  
];  
Encode the Function Call  
Encode the mintTo function call with the parameter being the address of the first smart wallet  
const callData = encodeFunctionData({  
 abi: abi,  
 functionName: 'mintTo',  
 args: [smartAccountClient.account.address],  
});  
Create a function called sendTransactionFromAccount1 in order to send the transaction request and store the response in a variable named txHash:  
async function sendTransactionFromAccount1() {  
 try {  
 const txHash = await smartAccountClient.sendTransaction({  
 account: smartAccountClient.account,  
 to: '0x83bd615eb93eE1336acA53e185b03B54fF4A17e8',  
 data: callData,  
 value: 0n,  
 });  
  
 console.log('✅ Transaction successfully sponsored for account 1!');  
 console.log(`🔍 View on Etherscan: https://basescan.org/tx/${txHash}`);  
 } catch (error) {  
 console.log('Transaction failed from account 1: ', error);  
 }  
}  
Call the function at the bottom of the index.js file:  
async function sendTransactionFromAccount1() {  
 // previous lines of code ...  
}  
  
sendTransactionFromAccount1();  
Run the following line of code from your terminal, ensuring you are in the root folder:  
node index.js  
The first attempt should be successful with the following output:  
✅ Transaction successfully sponsored!  
🔍 View on Etherscan: https://basescan.org/tx/0x1439431b0bf333f8106bc1815a532fdec18d62c3ee1c07a10d07711c6e5a2e56  
Run the script again but this time you should get an error that contains a code and message similar to this:  
{  
 "code": -32001,  
 "message": "request denied - rejected due to maximum per address transaction count reached"  
}  
Awesome! Errors, in this case, are good. This means our Paymaster policy works and prevents unwanted sponsorship.  
Navigate back to your CDP portal and update the number of UserOperations to 2.  
Give it about 10-15 minutes for the changes to take affect then run node index.js again to send a new transaction.  
It should return a successful prompt:  
✅ Transaction successfully sponsored for account 1!  
🔍 View on Etherscan: https://basescan.org/tx/0x1439431b0bf333f8106bc1815a532fdec18d62c3ee1c07a10d07711c6e5a2e56  
Looking back at our Paymaster configuration, you'll see that you've used about $0.03 of our $0.05 global limit, and for account 1, you've hit our UserOperation limit.  
Now, let's send another transaction, this time using the second account.  
Before you do that, let's update the per-wallet UserOperation limit to 10. Since gas is cheap on Base, you may need to do a few transactions before you hit our global limit of $0.07.  
INCREASE THE GLOBAL LIMIT  
You may get a warning to increase the global limit and that is okay for now...  
[Image-of-policy]  
Navigate back to the UI and update the policy to 10 and hit the Save button.  
[image of updated policy]  
Testing Global Limits  
Back in your code editor, open the index.js file and create a function called sendTransactionFromAccount2 and run place it at the bottom on the index.js file:  
// previous lines of code ...  
  
async function sendTransactionFromAccount2() {  
 try {  
 const txHash2 = await smartAccountClient2.sendTransaction({  
 account: smartAccountClient2.account,  
 to: '0x83bd615eb93eE1336acA53e185b03B54fF4A17e8',  
 data: callData,  
 value: 0n,  
 });  
  
 console.log('✅ Transaction successfully sponsored for account 2!');  
 console.log(`🔍 View on Etherscan: https://basescan.org/tx/${txHash2}`);  
 } catch (error) {  
 console.log('Transaction failed from account 2: ', error);  
 }  
}  
  
// sendTransactionFromAccount1();  
sendTransactionFromAccount2();  
Run the sendTransactionFromAccount2() function a few times to help reach your global limit.  
You should eventually encounter an error similar to this:  
{  
 "code": -32001,  
 "message": "request denied - rejected due to max global usd spend limit reached"  
}  
Congrats! Your global limit has been hit.  
Navigate back to the cloud portal and increase the global policy to your choosing. Run node index.js again and you should be able to successfully sponsor a transaction.  
Conclusion  
In this tutorial, you learned to set up and configure the Base Paymaster on the Coinbase Developer Platform, allowing you to sponsor gasless transactions for your users. This enhances the user experience by abstracting away gas fees and making transactions more accessible and cost-effective.  
Tags:account abstractionPaymaster  
ON THIS PAGE  
Prerequisites  
Set Up a Base Paymaster & Bundler  
Allowlist a Sponsorable Contract  
Global & Per User Limits:  
Foundry  
Test Your Paymaster policy  
Set Constants for Your Paymaster & Bundler endpoint  
Create Two Smart Accounts  
Conclusion  
Set Constants for Your Paymaster & Bundler endpoint  
Create Two Smart Accounts  
Initialize Paymaster and Create Accounts  
Encode the Function Call  
Testing Global Limits  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/hardhat-debugging  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Hardhat: Debugging smart contracts  
In this tutorial, you'll learn how to debug your smart contracts using the built-in debugging capabilities of Hardhat.  
Objectives  
By the end of this tutorial, you should be able to:  
Use console.log to write debugging logs  
List common errors and their resolutions  
Determine if an error is a contract error or an error in the test  
Overview  
Debugging smart contracts can be a challenging task, especially when dealing with decentralized applications and blockchain technology. Hardhat provides powerful tools to simplify the debugging process.  
In this tutorial, you will explore the essential debugging features offered by Hardhat and learn how to effectively identify and resolve common errors in your smart contracts.  
Your first console.log  
One of the key features of Hardhat is the ability to use console.log for writing debugging logs in your smart contracts. In order to use it, you must include hardhat/console.sol in the contract you wish to debug.  
In the following contract Lock.sol for example, you include hardhat/console.sol by importing it and adding a few console.logs in the constructor with the text "Creating" and the Ether balance of the contract. This can help you not only with tracking that the contract was created successfully but also, more importantly, with the ability to include additional logs such as the balance of the contract after it was created:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "hardhat/console.sol";  
  
contract Lock {  
 uint256 public unlockTime;  
 address payable public owner;  
  
 constructor(uint \_unlockTime) payable {  
 require(  
 block.timestamp < \_unlockTime,  
 "Unlock time should be in the future"  
 );  
  
 unlockTime = \_unlockTime;  
 owner = payable(msg.sender);  
  
 console.log("Creating");  
 console.log("Balance", address(this).balance);  
 }  
}  
In order to test it, you need to create a new file in the test folder called Lock.test.ts with the following content:  
import { expect } from "chai";  
import { ethers } from "hardhat";  
  
import { time } from "@nomicfoundation/hardhat-network-helpers";  
import { SignerWithAddress } from '@nomicfoundation/hardhat-ethers/signers'  
  
import { Lock\_\_factory, Lock } from '../typechain-types'  
  
describe("Lock Tests", function () {  
 const UNLOCK\_TIME = 10000;  
 const VALUE\_LOCKED = ethers.parseEther("0.01");  
  
 let lastBlockTimeStamp: number;  
 let lockInstance: Lock;  
 let ownerSigner: SignerWithAddress  
  
 before(async () => {  
 lastBlockTimeStamp = await time.latest()  
  
 const signers = await ethers.getSigners()  
 ownerSigner = signers[0]  
  
 lockInstance = await new Lock\_\_factory().connect(ownerSigner).deploy(lastBlockTimeStamp + UNLOCK\_TIME, {  
 value: VALUE\_LOCKED  
 })  
 })  
  
 it('should get the unlockTime value', async () => {  
 expect(await lockInstance.unlockTime()).to.equal(lastBlockTimeStamp + UNLOCK\_TIME)  
 })  
});  
Notice that a single test is included in order to get proper logs. However, you're only interested in the creation process that happens in the before hook. Then, you can run:  
npx hardhat test  
You should see the following in the terminal:  
 Lock  
Creating  
Balance 10000000000000000  
 ✔ should get the unlockTime value  
The terminal shows the text "Creating" and the balance (which is 0.01 Ether) because during the creation, you are depositing Ether in the smart contract via the value property.  
A note about console.log  
In the previous example, you used console.log to include some debugging logs. Be aware that the console.log version of Solidity is limited compared to the ones that are provided in other programming languages, where you can log almost anything.  
Console.log can be called with up to four parameters of the following types:  
uint  
string  
bool  
address  
Hardhat includes other console functions, such as:  
console.logInt(int i)  
console.logBytes(bytes memory b)  
console.logBytes1(bytes1 b)  
console.logBytes2(bytes2 b)  
...  
console.logBytes32(bytes32 b)  
These log functions are handy when the type you intend to log doesn't fall within the default accepted types of console.log. For further details, refer to the official console.log documentation.  
Identifying common errors  
While debugging your smart contracts, it's crucial to be familiar with common errors that can arise during development. Recognizing these errors and knowing how to resolve them is an important skill.  
In our Base Learn series of tutorials, we cover a few compile-time errors in Error Triage. Other errors, such as reverts or index out of bounds errors can be unexpected during the runtime of the smart contract.  
The following explores typical techniques to debug these types of errors.  
Revert errors  
When a transaction fails due to a require or revert statement, you'll need to diagnose why the condition isn't met and then resolve it. Typically, this involves verifying input parameters, state variables, or contract conditions.  
The following is the Lock.sol contract with a require statement that validates that the parameter you are passing (\_unlockTime) must be greater than the current block.timestamp.  
A simple solution to troubleshoot this error is to log the value of block.timestamp and \_unlockTime, which will help you compare these values and then ensure that you are passing the correct ones:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "hardhat/console.sol";  
  
contract Lock {  
 uint public unlockTime;  
 address payable public owner;  
  
 // event Withdrawal(uint amount, uint when);  
  
 constructor(uint \_unlockTime) payable {  
 console.log("\_unlockTime",\_unlockTime);  
 console.log("block.timestamp",block.timestamp);  
 require(  
 block.timestamp < \_unlockTime,  
 "Unlock time should be in the future"  
 );  
  
 unlockTime = \_unlockTime;  
 owner = payable(msg.sender);  
  
 console.log("Creating");  
 console.log("Balance", address(this).balance);  
 }  
}  
When you run the tests with npx hardhat test, you'll then see the following:  
Lock Tests  
\_unlockTime 1697493891  
block.timestamp 1697483892  
Creating  
Balance 10000000000000000  
 ✔ should get the unlockTime value  
You are now able to see the block.timestamp and the value you are passing, which makes it easier to detect the error.  
Unintended behavior errors  
Unintended behavior errors occur when you introduce unexpected behavior into the codebase due to a misunderstanding in the way Solidity works.  
In the following example, LockCreator is a contract that allows anybody to deploy a Lock.sol instance. However, the LockCreator contains an error: the createLock functions are able to accept Ether to be locked but the amount sent is not being transferred to the Lock contract:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "hardhat/console.sol";  
  
import {Lock} from "./Lock.sol";  
  
contract LockCreator {  
  
 Lock[] internal locks;  
  
 // Example of bad code, do not use  
 function createLock(uint256 \_unlockTime) external payable {  
 Lock newLock = new Lock(\_unlockTime);  
 locks.push(newLock);  
 }  
}  
You can create a test file LockCreator.test.ts that can identify the error and then solve it:  
import { ethers } from "hardhat";  
  
import { time } from "@nomicfoundation/hardhat-network-helpers";  
import { SignerWithAddress } from '@nomicfoundation/hardhat-ethers/signers'  
  
import { LockCreator, LockCreator\_\_factory } from '../typechain-types'  
  
describe("LockCreator Tests", function () {  
 const UNLOCK\_TIME = 10000;  
 const VALUE\_LOCKED = ethers.parseEther("0.01");  
  
 let lastBlockTimeStamp: number;  
 let lockInstance: LockCreator;  
 let ownerSigner: SignerWithAddress  
  
 before(async () => {  
 const signers = await ethers.getSigners()  
 ownerSigner = signers[0]  
  
 lockInstance = await new LockCreator\_\_factory().connect(ownerSigner).deploy()  
 })  
  
 it('should create a lock', async () => {  
 lastBlockTimeStamp = await time.latest()  
 await lockInstance.createLock(lastBlockTimeStamp + UNLOCK\_TIME, {  
 value: VALUE\_LOCKED  
 })  
 })  
});  
The following appears in the terminal where you can see the balance is 0:  
 LockCreator Tests  
Creating  
Balance 0  
 ✔ should create a lock (318ms)  
Although this issue can be avoided by adding more test cases with proper assertions, the re-transfer of Ether from the LockCreator was something you may have overlooked.  
The solution is to modify the createLock function with:  
function createLock(uint256 \_unlockTime) external payable {  
 Lock newLock = new Lock{ value: msg.value}(\_unlockTime);  
 locks.push(newLock);  
}  
Out-of-bounds errors  
Attempting to access arrays at an invalid position can also cause errors.  
If you wish to retrieve all the Lock contract instances being created in the previous example, you can make the locks array public. In order to illustrate this example, though, you can create a custom function called getAllLocks:  
contract LockCreator {  
 //  
 // rest of the code..  
 //  
 function getAllLocks() external view returns(Lock[] memory result) {  
 result = new Lock[](locks.length);  
 for(uint i = 0; i <= locks.length; i++){  
 result[i] = locks[i];  
 }  
 }  
}  
The function can be tested with the following test:  
describe("LockCreator Tests", function () {  
 const UNLOCK\_TIME = 10000;  
 const VALUE\_LOCKED = ethers.parseEther("0.01");  
  
 let lastBlockTimeStamp: number;  
 let lockInstance: LockCreator;  
 let ownerSigner: SignerWithAddress  
  
 before(async () => {  
 const signers = await ethers.getSigners()  
 ownerSigner = signers[0]  
  
 lockInstance = await new LockCreator\_\_factory().connect(ownerSigner).deploy()  
  
 lastBlockTimeStamp = await time.latest()  
  
 await lockInstance.createLock(lastBlockTimeStamp + UNLOCK\_TIME, {  
 value: VALUE\_LOCKED  
 })  
 })  
  
 it('should get all locks', async () => {  
 const allLocks = await lockInstance.getAllLocks()  
  
 console.log("all locks", allLocks)  
 })  
});  
Which will then throw an error:  
LockCreator Tests  
Creating  
Balance 10000000000000000  
 1) should get all locks  
  
 0 passing (3s)  
 1 failing  
  
 1) LockCreator Tests  
 should get all locks:  
 Error: VM Exception while processing transaction: reverted with panic code 0x32 (Array accessed at an out-of-bounds or negative index)  
You can include some debugging logs to identify the issue:  
 function getAllLocks() external view returns(Lock[] memory result) {  
 result = new Lock[](locks.length);  
  
 console.log("locks length %s", locks.length);  
  
 for(uint i = 0; i <= locks.length; i++){  
 console.log("Locks index %s", i);  
 result[i] = locks[i];  
 }  
}  
Then, you see the following in the terminal:  
 LockCreator Tests  
Creating  
Balance 10000000000000000  
locks length 1  
Locks index 0  
Locks index 1  
1) LockCreator Tests  
 should get all locks:  
 Error: VM Exception while processing transaction: reverted with panic code 0x32 (Array accessed at an out-of-bounds or negative index)  
Since arrays are 0 index based, an array with 1 item will store that item at the 0 index. In the above example, the if statement compares <= against the length of the array, so it tries to access the element in position 1, and crashes.  
Here's the simple solution:  
 function getAllLocks() external view returns(Lock[] memory result) {  
 result = new Lock[](locks.length);  
  
 console.log("locks length %s", locks.length);  
  
 for(uint i = 0; i < locks.length; i++){  
 console.log("Locks index %s", i);  
 result[i] = locks[i];  
 }  
}  
Which immediately solves the problem:  
 LockCreator Tests  
Creating  
Balance 10000000000000000  
locks length 1  
Locks index 0  
all locks Result(1) [ '0x83BA8C2028EE8a6476396145C7692fBD09337acD' ]  
 ✔ should get all locks  
  
  
 1 passing (3s)  
Conclusion  
In this tutorial, you've learned some techniques about how to debug smart contracts using Hardhat. You explored some common cases of various errors and how by simply using console.log and a proper test, you can identify and solve the problem.  
See also  
Tags:smart contracts  
ON THIS PAGE  
Overview  
Your first console.log  
Identifying common errors  
Revert errors  
Conclusion  
Revert errors  
Unintended behavior errors  
Out-of-bounds errors  
See also  
See also  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/hardhat-profiling-gas  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Hardhat: Optimizing the gas usage of smart contracts  
In this tutorial, you'll learn how to profile and optimize your smart contract's gas usage with Hardhat and the Hardhat Gas Reporter plugin.  
Objectives  
By the end of this tutorial you should be able to:  
Use the Hardhat Gas Reporter plugin to profile gas usage  
Describe common strategies for improving the gas usage of a contract  
Overview  
In the world of smart contract development, optimizing the gas consumption of your smart contracts is important. Smaller contracts consume fewer gas resources during deployment and execution, resulting in significant cost savings for your users. In this tutorial, you will leverage the Hardhat Gas Reporter plugin to help you analyze and optimize your smart contract's gas usage.  
The following provides further information about smart contract profiling and gas optimization.  
Setting up the Hardhat Gas Reporter plugin  
The Hardhat Gas Reporter plugin is an invaluable tool for profiling gas usage in your smart contracts. It allows you to gain insights into the gas consumption of various contract functions, making it easier to identify potential optimization opportunities. This tool is particularly useful during development when you want to ensure your contracts are as gas-efficient as possible.  
To install, run npm install -D hardhat-gas-reporter.  
Then, import hardhat-gas-reporter in hardhat.config.ts:  
import "hardhat-gas-reporter"  
Configure the plugin in the hardhat.config.ts file:  
const config: HardhatUserConfig = {  
 // ....  
 gasReporter: {  
 enabled: true,  
 },  
};  
When finished, you are ready to use the plugin.  
Your first gas profiling  
Create a contract called Store with the following settings:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Store {  
 address public owner;  
 uint256 public numberOfItems;  
  
 struct Item {  
 uint256 id;  
 string description;  
 uint256 price;  
 }  
  
 // id => item  
 mapping(uint256 => Item) public items;  
  
 constructor() {  
 owner = msg.sender;  
 }  
  
 function addItem(string memory description, uint256 price) external {  
 require(msg.sender == owner, "invalid owner");  
  
 numberOfItems++;  
  
 items[numberOfItems] = Item(  
 numberOfItems,  
 description,  
 price  
 );  
 }  
}  
  
Add a test file called Store.test.ts in order to test the gas reporter plugin. The test file should contain the following:  
import { expect } from 'chai';  
import { ethers } from 'hardhat';  
import { HardhatEthersSigner } from '@nomicfoundation/hardhat-ethers/signers';  
  
import { Store, Store\_\_factory } from '../typechain-types';  
  
describe('Store tests', function () {  
 let instance: Store;  
 let owner: HardhatEthersSigner;  
  
 before(async () => {  
 const signers = await ethers.getSigners();  
 owner = signers[0];  
 instance = await new Store\_\_factory().connect(owner).deploy();  
 });  
  
 it('should add an item', async () => {  
 const description = 'TShirt';  
 const price = ethers.parseEther('1');  
  
 await instance.addItem(description, price);  
  
 expect(await instance.numberOfItems()).to.equal(1);  
 });  
});  
Run npx hardhat test. The following report appears:  
·------------------------|---------------------------|---------------|-----------------------------·  
| Solc version: 0.8.18 · Optimizer enabled: true · Runs: 10000 · Block limit: 30000000 gas │  
·························|···························|···············|······························  
| Methods │  
·············|···········|·············|·············|···············|···············|··············  
| Contract · Method · Min · Max · Avg · # calls · usd (avg) │  
·············|···········|·············|·············|···············|···············|··············  
| Store · addItem · - · - · 113601 · 1 · - │  
·············|···········|·············|·············|···············|···············|··············  
| Deployments · · % of limit · │  
·························|·············|·············|···············|···············|··············  
| Store · - · - · 428837 · 1.4 % · - │  
·------------------------|-------------|-------------|---------------|---------------|-------------·  
The reporter provides a detailed overview of the gas costs for the function addItem and the deployment costs.  
Common strategies to optimize contract sizes  
After performing the first gas profiling, you can start ideating strategies to improve the gas costs. These strategies are certainly vast and this tutorial only covers some basic examples.  
Using the optimizer  
From the previous report, you can identify that the optimizer of the project has a value of 10000 runs. This means the deployment costs will be more expensive. However, if you modify that value to 200, you get:  
·------------------------|---------------------------|-------------|-----------------------------·  
| Solc version: 0.8.18 · Optimizer enabled: true · Runs: 200 · Block limit: 30000000 gas │  
·························|···························|·············|······························  
| Methods │  
·············|···········|·············|·············|·············|···············|··············  
| Contract · Method · Min · Max · Avg · # calls · usd (avg) │  
·············|···········|·············|·············|·············|···············|··············  
| Store · addItem · - · - · 113619 · 1 · - │  
·············|···········|·············|·············|·············|···············|··············  
| Deployments · · % of limit · │  
·························|·············|·············|·············|···············|··············  
| Store · - · - · 357505 · 1.2 % · - │  
·------------------------|-------------|-------------|-------------|---------------|-------------·  
This automatically gives you some improvements for deployment gas costs but slightly more for transaction executions.  
Using immutable variables  
In our Store contract, you can identify certain variables that are only set during the creation of the contract. This means that an opportunity is possible to turn those variables into immutable, since immutable variables can still be assigned at construction time.  
If you modify the Store contract to:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Store {  
 address immutable owner;  
 uint256 public numberOfItems;  
  
 struct Item {  
 uint256 id;  
 string description;  
 uint256 price;  
 }  
  
 // id => item  
 mapping(uint256 => Item) public items;  
  
 constructor() {  
 owner = msg.sender;  
 }  
  
 function addItem(string memory description, uint256 price) external {  
 require(msg.sender == owner, "invalid owner");  
  
 numberOfItems++;  
  
 items[numberOfItems] = Item(  
 numberOfItems,  
 description,  
 price  
 );  
 }  
}  
Then, run the gas reporter. You should see:  
·------------------------|---------------------------|-------------|-----------------------------·  
| Solc version: 0.8.18 · Optimizer enabled: true · Runs: 200 · Block limit: 30000000 gas │  
·························|···························|·············|······························  
| Methods │  
·············|···········|·············|·············|·············|···············|··············  
| Contract · Method · Min · Max · Avg · # calls · usd (avg) │  
·············|···········|·············|·············|·············|···············|··············  
| Store · addItem · - · - · 111525 · 1 · - │  
·············|···········|·············|·············|·············|···············|··············  
| Deployments · · % of limit · │  
·························|·············|·············|·············|···············|··············  
| Store · - · - · 329580 · 1.1 % · - │  
·------------------------|-------------|-------------|-------------|---------------|-------------·  
Which already presents some improvements.  
Avoid unnecessary data storage  
Storing data and not storing data in a smart contract is a design decision that has pros and cons. Some of the pros are certainly that all the information is stored in the smart contract and you don't necessarily need to rely on events or any other service to access the storage of a contract. However, the cons of storing all the information on the contract is the fact that it will be more expensive to perform actions against the smart contract.  
In the Store smart contract, you have the following:  
struct Item {  
 uint256 id;  
 string description;  
 uint256 price;  
 }  
  
// id => item  
mapping(uint256 => Item) public items;  
Looking closely, you can see that the Id of the Item struct and the id used in the mapping are similar. You can avoid duplicating this information by removing the id of the Item struct.  
The contract looks like:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Store {  
 address immutable owner;  
 uint256 public numberOfItems;  
  
 struct Item {  
 string description;  
 uint256 price;  
 }  
  
 // id => item  
 mapping(uint256 => Item) public items;  
  
 constructor() {  
 owner = msg.sender;  
 }  
  
 function addItem(string memory description, uint256 price) external {  
 require(msg.sender == owner, "invalid owner");  
  
 numberOfItems++;  
  
 items[numberOfItems] = Item(  
 description,  
 price  
 );  
 }  
}  
If you run the gas reporter, you then get:  
·------------------------|---------------------------|-------------|-----------------------------·  
| Solc version: 0.8.18 · Optimizer enabled: true · Runs: 200 · Block limit: 30000000 gas │  
·························|···························|·············|······························  
| Methods │  
·············|···········|·············|·············|·············|···············|··············  
| Contract · Method · Min · Max · Avg · # calls · usd (avg) │  
·············|···········|·············|·············|·············|···············|··············  
| Store · addItem · - · - · 89371 · 1 · - │  
·············|···········|·············|·············|·············|···············|··············  
| Deployments · · % of limit · │  
·························|·············|·············|·············|···············|··············  
| Store · - · - · 322251 · 1.1 % · - │  
·------------------------|-------------|-------------|-------------|---------------|-------------·  
This presents another improvement in the gas consumption of the Store smart contract. However, you can go further and instead of storing the items in a mapping, you can simply emit events and use the events as a cheap form of storage.  
For instance, you can modify the contract to look like:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Store {  
 address immutable owner;  
 uint256 public numberOfItems;  
  
 struct Item {  
 string description;  
 uint256 price;  
 }  
  
 event ItemCreated(uint256 id, Item item);  
  
 constructor() {  
 owner = msg.sender;  
 }  
  
 function addItem(string memory description, uint256 price) external {  
 require(msg.sender == owner, "invalid owner");  
  
 numberOfItems++;  
  
 emit ItemCreated(numberOfItems, Item(description, price));  
 }  
}  
Notice how instead of storing the items, you emit an ItemCreated event, which reduces the gas costs for deployment and execution:  
·------------------------|---------------------------|-------------|-----------------------------·  
| Solc version: 0.8.18 · Optimizer enabled: true · Runs: 200 · Block limit: 30000000 gas │  
·························|···························|·············|······························  
| Methods │  
·············|···········|·············|·············|·············|···············|··············  
| Contract · Method · Min · Max · Avg · # calls · usd (avg) │  
·············|···········|·············|·············|·············|···············|··············  
| Store · addItem · - · - · 47315 · 1 · - │  
·············|···········|·············|·············|·············|···············|··············  
| Deployments · · % of limit · │  
·························|·············|·············|·············|···············|··············  
| Store · - · - · 208252 · 0.7 % · - │  
·------------------------|-------------|-------------|-------------|---------------|-------------·  
As you can see, the improvements in terms of gas consumption are significant. However, the draw back is that now in order to access all of the items, you must go through all of the ItemCreated events emitted by the contract.  
Using custom errors  
Another common way to optimize gas costs is by removing requires and use custom errors. For instance, you can do the following:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Store {  
 address immutable owner;  
 uint256 public numberOfItems;  
  
 error InvalidOwner();  
  
 struct Item {  
 string description;  
 uint256 price;  
 }  
  
 event ItemCreated(uint256 id, Item item);  
  
 constructor() {  
 owner = msg.sender;  
 }  
  
 function addItem(string memory description, uint256 price) external {  
 if(msg.sender != owner){  
 revert InvalidOwner();  
 }  
  
 numberOfItems++;  
  
 emit ItemCreated(numberOfItems, Item(description, price));  
 }  
}  
Which gives you the following report:  
·------------------------|---------------------------|-------------|-----------------------------·  
| Solc version: 0.8.18 · Optimizer enabled: true · Runs: 200 · Block limit: 30000000 gas │  
·························|···························|·············|······························  
| Methods │  
·············|···········|·············|·············|·············|···············|··············  
| Contract · Method · Min · Max · Avg · # calls · usd (avg) │  
·············|···········|·············|·············|·············|···············|··············  
| Store · addItem · - · - · 47315 · 1 · - │  
·············|···········|·············|·············|·············|···············|··············  
| Deployments · · % of limit · │  
·························|·············|·············|·············|···············|··············  
| Store · - · - · 200683 · 0.7 % · - │  
·------------------------|-------------|-------------|-------------|---------------|-------------·  
Notice the improvement in deployment gas costs.  
Conclusion  
In this tutorial, you've learned some common strategies to profile and optimize the gas usage of your smart contracts using the Hardhat development environment and the Hardhat Gas Reporter plugin. By implementing these strategies and leveraging the Hardhat Gas Reporter plugin, you can create more efficient and cost-effective smart contracts for the benefit of the users, since this means less gas costs.  
Tags:smart contracts  
ON THIS PAGE  
Overview  
Setting up the Hardhat Gas Reporter plugin  
Your first gas profiling  
Common strategies to optimize contract sizes  
Conclusion  
Using the optimizer  
Using immutable variables  
Avoid unnecessary data storage  
Using custom errors  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/hardhat-profiling-size  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Hardhat: Optimizing the size of smart contracts  
In this tutorial, you'll learn how to profile and optimize smart contract sizes with Hardhat and the Hardhat Contract Sizer plugin.  
Objectives  
By the end of this tutorial you should be able to:  
Use Hardhat Contract Sizer plugin to profile contract size  
Describe common strategies for managing the contract size limit  
Describe the impact that inheritance has on the byte code size limit  
Describe the impact that external contracts have on the byte code size limit  
Describe the impact of using libraries has on the byte code size limit  
Describe the impact of using the Solidity optimizer  
Overview  
In the world of blockchain and Ethereum, optimizing smart contract sizes is crucial. Smaller contracts consume less gas during deployment and execution, which is translated into gas costs savings for your users. Fortunately, you can use in Hardhat the hardhat-contract-sizer plugin that helps you analyze and optimize the size of your smart contracts.  
Setting up the Hardhat Contract Sizer plugin  
Hardhat Contract Sizer is a community-developed plugin that enables the profiling of smart contract by printing the size of your smart contracts in the terminal. This is helpful during development since it allows you to immediately identify potential issues with the size of your smart contracts. Keep in mind that the maximum size of a smart contract in Ethereum is 24 KiB.  
To install, run npm install -D hardhat-contract-sizer.  
Then, import hardhat-contract-sizer in hardhat.config.ts:  
import "hardhat-contract-sizer"  
When finished, you are ready to use the plugin.  
Your first size profiling  
Similar to the previous tutorials, you begin by profiling the smart contract Lock.sol.  
Run npx hardhat size-contracts, which is a task added to Hardhat once you set up and configure the hardhat-contract-sizer plugin.  
You are then able to see:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: false · Runs: 200 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | BalanceReader · 0.612 () · 0.644 () │  
 ·························|································|·································  
 | Lock · 1.009 () · 1.461 () │  
Although your contract is simple, you can see immediately the power of the hardhat-contract-sizer plugin, since it show you the size of your contracts.  
Common strategies to optimize contract sizes  
In order to illustrate some of the strategies to optimize the size of your contracts, create two smart contracts, Calculator.sol and ScientificCalculator.sol, with the following:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Calculator {  
 function add(uint256 a, uint256 b) external pure returns(uint256) {  
 require(a > 0 && b > 0, "Invalid values");  
 return a + b;  
 }  
  
 function sub(uint256 a, uint256 b) external pure returns(uint256) {  
 require(a > 0 && b > 0, "Invalid values");  
 return a - b;  
 }  
  
 function mul(uint256 a, uint256 b) external pure returns(uint256) {  
 require(a > 0 && b > 0, "Invalid values");  
 return a \* b;  
 }  
  
 function div(uint256 a, uint256 b) external pure returns(uint256) {  
 require(a > 0 && b > 0, "Invalid values");  
 return a / b;  
 }  
}  
contract ScientificCalculator is Calculator {  
 function power(uint256 base, uint256 exponent) public pure returns (uint256) {  
 require(base > 0 && exponent > 0, "Invalid values");  
  
 return base \*\* exponent;  
 }  
}  
Then, run the command npx hardhat size-contracts again and you should be able to see:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: false · Runs: 200 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | BalanceReader · 0.612 (0.000) · 0.644 (0.000) │  
 ·························|································|·································  
 | Lock · 1.009 (0.000) · 1.461 (0.000) │  
 ·························|································|·································  
 | Calculator · 1.299 () · 1.330 () │  
 ·························|································|·································  
 | ScientificCalculator · 1.827 () · 1.858 () │  
 ·------------------------|--------------------------------|--------------------------------·  
Notice how the size of ScientificCalculator is bigger than Calculator. This is because ScientificCalculator is inheriting the contract Calculator, which means all of its functionality and code is available in ScientificCalculator and that will influence its size.  
Code abstraction and modifiers  
At this point as a smart contract developer, you can review your smart contract code and look for ways into you can optimize it.  
The first thing you notice in the source code is the extensive use of:  
require(a > 0 && b > 0, "Invalid values");  
A possible optimization is to abstract repetitive code into modifiers, such as the following:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Calculator {  
 error InvalidInput();  
  
 function add(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a + b;  
 }  
  
 function sub(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a - b;  
 }  
  
 function mul(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a \* b;  
 }  
  
 function div(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a / b;  
 }  
  
 modifier onlyValidInputs(uint256 a, uint256 b) {  
 if(a == 0 && b == 0){  
 revert InvalidInput();  
 }  
 \_;  
 }  
}  
And for ScientificCalculator:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "./Calculator.sol";  
  
contract ScientificCalculator is Calculator {  
 function power(uint256 base, uint256 exponent) public pure onlyValidInputs(base,exponent) returns (uint256) {  
 return base \*\* exponent;  
 }  
}  
Notice the usage of the modifier and the replacement of the require to use a custom error.  
When you run the npx hardhat size-contracts command, you should be able to see:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: false · Runs: 200 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | BalanceReader · 0.612 (0.000) · 0.644 (0.000) │  
 ·························|································|·································  
 | Lock · 1.009 (0.000) · 1.461 (0.000) │  
 ·························|································|·································  
 | Calculator · 1.165 (0.000) · 1.196 (0.000) │  
 ·························|································|·································  
 | ScientificCalculator · 1.690 (0.000) · 1.722 (0.000) │  
 ·------------------------|--------------------------------|--------------------------------·  
Although the optimization is small, you can see that there are some improvements.  
You can continue this process until you feel comfortable with the size of the contract.  
Split into multiple contracts  
It is common to split your smart contracts into multiple contracts, not only because of the size limitations but to create better abstractions, to improve readability, and to avoid repetition.  
From a contract size perspective, having multiple independent contracts will reduce the size of each contract. For example, the original size of a smart contract was 30 KiB: by splitting into 2, you will end up with 2 smart contracts of ~15 KiB that are within the limits of Solidity. Keep in mind that this will influence gas costs during the execution of the contract because it will require it to call an external contract.  
In order to explain this example, create a contract called Computer that contains a function called executeProcess:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "hardhat/console.sol";  
  
contract Computer {  
 function executeProcess() external view {  
 // ...logic to be implemented  
 }  
}  
In this example, the executeProcess function of Computer requires certain functionality of Calculator and a new contract called Printer:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "hardhat/console.sol";  
  
contract Printer {  
 function print(string memory \_content) external view {  
 require(bytes(\_content).length > 0, "invalid length");  
 console.log(\_content);  
 }  
}  
The easiest way for Computer to access both functionalities is to inherit; however, as all of these contracts continue adding functionality, the size of the code will also increase. You will reach the contract size issue at some point, since you are copying the entire functionality into your contract. You can better allow that functionality to be kept with their specific contracts and if the Computer requires to access that functionality, you could call the Calculator and Printer contracts.  
But in this example, there is a process that must call both Calculator and Printer:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "hardhat/console.sol";  
  
import "./Calculator.sol";  
import "./Printer.sol";  
  
contract Computer {  
 Calculator calculator;  
 Printer printer;  
  
 constructor(address \_calculator, address \_printer) {  
 calculator = Calculator(\_calculator);  
 printer = Printer(\_printer);  
 }  
  
 function executeProcess() external view {  
 // call Calculator contract, i.e calculator.add(a, b);  
 // call Printer contract, i.e printer.print("value to print");  
 }  
}  
If you run the contract sizer plugin, you get:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: true · Runs: 10000 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | console · 0.084 (0.000) · 0.138 (0.000) │  
 ·························|································|·································  
 | Computer · 0.099 (0.000) · 0.283 (0.000) │  
 ·························|································|·································  
 | Calculator · 0.751 (0.000) · 0.782 (0.000) │  
 ·························|································|·································  
 | Printer · 0.761 (0.000) · 0.792 (0.000) │  
 ·························|································|·································  
 | ScientificCalculator · 1.175 (0.000) · 1.206 (0.000) │  
 ·------------------------|--------------------------------|--------------------------------·  
Notice how your Computer contract is very small but still has the capability to access all the functionality of Printer and Calculator.  
Although this will reduce the size of each contract, the costs of this are discussed more deeply in the Gas Optimization article.  
Using libraries  
Libraries are another common way to encapsulate and abstract common functionality that can be shared across multiple contracts. This can significantly impact the bytecode size of the smart contracts. Remember that in Solidity, libraries can be external and internal.  
The way internal libraries affect the contract size is very similar to the way inherited contracts affects a contract's size; this is because the internal functions of the library is included within the final bytecode.  
But when the libraries are external, the behavior is different: the way Solidity calls external libraries is by using a special function called delegate call.  
External libraries are commonly deployed independently and can be reused my multiple contracts. Since libraries don't keep a state, they behave like pure functions in the Blockchain.  
In this example, your computer will use the Calculator library only. Then, you would have the following:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
library Calculator {  
 error InvalidInput();  
  
 function add(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a + b;  
 }  
  
 function sub(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a - b;  
 }  
  
 function mul(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a \* b;  
 }  
  
 function div(uint256 a, uint256 b) external pure onlyValidInputs(a,b) returns(uint256) {  
 return a / b;  
 }  
  
 modifier onlyValidInputs(uint256 a, uint256 b) {  
 if(a == 0 && b == 0){  
 revert InvalidInput();  
 }  
 \_;  
 }  
}  
Then, Computer is:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "hardhat/console.sol";  
  
import "./Calculator.sol";  
import "./Printer.sol";  
  
contract Computer {  
 using Calculator for uint256;  
  
 function executeProcess() external view {  
 uint256 a = 1;  
 uint256 b = 2;  
 uint256 result = a.add(b);  
 // ... logic to be implemented  
 }  
}  
Notice how you instructing the smart contract to use the Calculator library for uint256 and how in the executeProcess function, you can now use the add function from the Calculator library in all of the uint256.  
If you run the npx hardhat size-contracts command, you then get:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: true · Runs: 10000 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | Calculator · 0.761 · 0.817 │  
 ·························|································|·································  
 | Printer · 0.771 · 0.827 │  
 ·························|································|·································  
 | Computer · 0.961 · 0.992 │  
 ·------------------------|--------------------------------|--------------------------------·  
In order to compare the impact, you can modify the external modifier from all of the Calculator library functions and you will then have:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: true · Runs: 10000 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | Calculator · 0.084 · 0.138 │  
 ·························|································|·································  
 | Printer · 0.084 · 0.138 │  
 ·························|································|·································  
 | Computer · 1.139 · 1.170 │  
 ·------------------------|--------------------------------|--------------------------------·  
Which demonstrates why using external libraries can be a good option in order to optimize the size of your contracts.  
Using the Solidity compiler optimizer  
Another way to optimize the size of the smart contracts is to simply use the Solidity optimizer.  
From the Solidity official docs:  
Overall, the optimizer tries to simplify complicated expressions, which reduces both code size and execution cost.  
You can enable the solidity optimizer in hardhat by simply adding the following to the hardhat.config.ts file:  
const config: HardhatUserConfig = {  
 solidity: {  
 version: "0.8.18",  
 settings: {  
 optimizer: {  
 enabled: true,  
 runs: 200  
 }  
 }  
 },  
 ...  
}  
Notice the optimizer is enabled and has a parameter runs. If you run the contract sizer command again, you will see the following:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: true · Runs: 200 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | BalanceReader · 0.351 (-0.262) · 0.382 (-0.262) │  
 ·························|································|·································  
 | Lock · 0.471 (-0.538) · 0.661 (-0.800) │  
 ·························|································|·································  
 | Calculator · 0.604 (-0.561) · 0.636 (-0.561) │  
 ·························|································|·································  
 | ScientificCalculator · 0.930 (-0.761) · 0.961 (-0.761) │  
 ·------------------------|--------------------------------|--------------------------------·  
Notice the bigger improvement, but see what happens if you increase the runs parameter value to 1000:  
 ·------------------------|--------------------------------|--------------------------------·  
 | Solc version: 0.8.18 · Optimizer enabled: true · Runs: 1000 │  
 ·························|································|·································  
 | Contract Name · Deployed size (KiB) (change) · Initcode size (KiB) (change) │  
 ·························|································|·································  
 | BalanceReader · 0.400 (+0.050) · 0.432 (+0.050) │  
 ·························|································|·································  
 | Lock · 0.537 (+0.066) · 0.728 (+0.066) │  
 ·························|································|·································  
 | Calculator · 0.604 (0.000) · 0.636 (0.000) │  
 ·························|································|·································  
 | ScientificCalculator · 0.945 (+0.016) · 0.977 (+0.016) │  
 ·------------------------|--------------------------------|--------------------------------·  
The size of the contract increased, however this means your code will be more efficient across the lifetime of the contract because the higher the runs value the more efficient during execution but more expensive during deployment. You can read more in the Solidity documentation.  
Conclusion  
In this tutorial, you've learned how to profile and optimise smart contracts using the Hardhat development environment and the Hardhat Contract Sizer plugin. By focusing on the critical aspect of contract size, we've equipped ourselves with tools and strategies to create more efficient Solidity code.  
As you continue your journey in smart contract development, keep in mind that optimizing contract sizes is a continuous process that requires careful consideration of trade-offs between size, readability, and gas efficiency.  
Tags:smart contracts  
ON THIS PAGE  
Overview  
Setting up the Hardhat Contract Sizer plugin  
Your first size profiling  
Common strategies to optimize contract sizes  
Conclusion  
Code abstraction and modifiers  
Split into multiple contracts  
Using libraries  
Using the Solidity compiler optimizer  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/hardhat-test-coverage  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Hardhat: Analyzing the test coverage of smart contracts  
In this tutorial, you'll learn how to profile the test coverage of your smart contracts with Hardhat and the Solidity Coverage community plugin.  
Objectives  
By the end of this tutorial, you should be able to:  
Use the Solidity Coverage plugin to analyze the coverage of your test suite  
Increase the coverage of your test suite  
Overview  
The Solidity Coverage plugin allows you to analyze and visualize the coverage of your smart contracts' test suite. This enables you to see what portions of your smart contract are being tested and what areas may have been overlooked. It's an indispensable plugin for developers seeking to fortify their testing practices and ensure robust smart contract functionality.  
Setting up the Solidity Coverage plugin  
The Solidity Coverage plugin is integrated into the Hardhat toolbox package, which is installed by default when you use the npx hardhat init command.  
To install manually, run npm install -D solidity-coverage.  
Then, import solidity-coverage in hardhat.config.ts:  
import "solidity-coverage"  
Once the installation completes either manually or via the default Hardhat template, the task coverage becomes available via the npx hardhat coverage command.  
My first test coverage  
Review the following contract and test suite (You'll recognize these if you completed the Hardhat testing lesson in our Base Learn series).  
Contract:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
contract Lock {  
 uint public unlockTime;  
 address payable public owner;  
  
 event Withdrawal(uint amount, uint when);  
  
 constructor(uint \_unlockTime) payable {  
 require(  
 block.timestamp < \_unlockTime,  
 "Unlock time should be in the future"  
 );  
  
 unlockTime = \_unlockTime;  
 owner = payable(msg.sender);  
 }  
  
 function withdraw() public {  
 require(block.timestamp >= unlockTime, "You can't withdraw yet");  
 require(msg.sender == owner, "You aren't the owner");  
  
 emit Withdrawal(address(this).balance, block.timestamp);  
  
 owner.transfer(address(this).balance);  
 }  
}  
Lock.test.ts:  
import { expect } from "chai";  
import { ethers } from "hardhat";  
import { time } from "@nomicfoundation/hardhat-network-helpers";  
import { SignerWithAddress } from '@nomicfoundation/hardhat-ethers/signers'  
import { Lock\_\_factory, Lock} from '../typechain-types'  
  
describe("Lock Tests", function () {  
 const UNLOCK\_TIME = 10000;  
 const VALUE\_LOCKED = ethers.parseEther("0.01");  
  
 let lastBlockTimeStamp: number;  
 let lockInstance: Lock;  
 let ownerSigner: SignerWithAddress  
 let otherUserSigner: SignerWithAddress;  
  
 before(async() => {  
 lastBlockTimeStamp = await time.latest()  
 const signers = await ethers.getSigners()  
 ownerSigner = signers[0]  
 otherUserSigner= signers[1]  
  
 const unlockTime = lastBlockTimeStamp + UNLOCK\_TIME;  
  
 lockInstance = await new Lock\_\_factory(ownerSigner).deploy(unlockTime, {  
 value: VALUE\_LOCKED  
 })  
 })  
  
  
 it('should get the unlockTime value', async() => {  
 const unlockTime = await lockInstance.unlockTime();  
  
 expect(unlockTime).to.equal(lastBlockTimeStamp + UNLOCK\_TIME)  
 })  
  
 it('should have the right ether balance', async() => {  
 const lockInstanceAddress = await lockInstance.getAddress()  
  
 const contractBalance = await ethers.provider.getBalance(lockInstanceAddress)  
  
 expect(contractBalance).to.equal(VALUE\_LOCKED)  
 })  
  
 it('should have the right owner', async()=> {  
 expect(await lockInstance.owner()).to.equal(ownerSigner.address)  
 })  
  
 it('should not allow to withdraw before unlock time', async()=> {  
 await expect(lockInstance.withdraw()).to.be.revertedWith("You can't withdraw yet")  
 })  
  
 it('should not allow to withdraw a non owner', async()=> {  
 const newLastBlockTimeStamp = await time.latest()  
  
 await time.setNextBlockTimestamp(newLastBlockTimeStamp + UNLOCK\_TIME)  
  
 const newInstanceUsingAnotherSigner = lockInstance.connect(otherUserSigner)  
  
 await expect(newInstanceUsingAnotherSigner.withdraw()).to.be.revertedWith("You aren't the owner")  
 })  
  
 it('should allow to withdraw a owner', async()=> {  
 const balanceBefore = await ethers.provider.getBalance(await lockInstance.getAddress());  
  
 expect(balanceBefore).to.equal(VALUE\_LOCKED)  
  
 const newLastBlockTimeStamp = await time.latest()  
  
 await time.setNextBlockTimestamp(newLastBlockTimeStamp + UNLOCK\_TIME)  
  
 await lockInstance.withdraw();  
  
 const balanceAfter = await ethers.provider.getBalance(await lockInstance.getAddress());  
 expect(balanceAfter).to.equal(0)  
 })  
});  
If you run npx hardhat coverage, you should get:  
 Lock Tests  
 ✔ should get the unlockTime value  
 ✔ should have the right ether balance  
 ✔ should have the right owner  
 ✔ shouldn't allow to withdraw before unlock time  
 ✔ shouldn't allow to withdraw a non owner  
 ✔ should allow to withdraw a owner  
  
  
 6 passing (195ms)  
  
------------|----------|----------|----------|----------|----------------|  
File | % Stmts | % Branch | % Funcs | % Lines |Uncovered Lines |  
------------|----------|----------|----------|----------|----------------|  
 contracts/ | 100 | 83.33 | 100 | 100 | |  
 Lock.sol | 100 | 83.33 | 100 | 100 | |  
------------|----------|----------|----------|----------|----------------|  
All files | 100 | 83.33 | 100 | 100 | |  
------------|----------|----------|----------|----------|----------------|  
Which then gives you a report of the test coverage of your test suite. Notice there is a new folder called coverage, which was generated by the solidity-coverage plugin. Inside the coverage folder there is a index.html file. Open it in a browser, you'll see a report similar to the following:  
Increasing test coverage  
Although the coverage of the previous test suite is almost perfect, there is one missing branch when creating the contract. Because you have not tested the condition that the \_unlockTime has to be greater than the block.timestamp:  
require(  
 block.timestamp < \_unlockTime,  
 "Unlock time should be in the future"  
 );  
In order to increase the coverage, include a new test with the following:  
 it('should verify the unlock time to be in the future', async () => {  
 const newLockInstance = new Lock\_\_factory(ownerSigner).deploy(lastBlockTimeStamp, {  
 value: VALUE\_LOCKED  
 })  
  
 await expect(newLockInstance).to.be.revertedWith("Unlock time should be in the future")  
 })  
Then, run npx hardhat coverage and you should get:  
 Lock Tests  
 ✔ should verify the unlock time to be in the future (39ms)  
 ✔ should get the unlockTime value  
 ✔ should have the right ether balance  
 ✔ should have the right owner  
 ✔ shouldn't allow to withdraw before unlock time  
 ✔ shouldn't allow to withdraw a non owner  
 ✔ should allow to withdraw a owner  
  
  
 7 passing (198ms)  
  
------------|----------|----------|----------|----------|----------------|  
File | % Stmts | % Branch | % Funcs | % Lines |Uncovered Lines |  
------------|----------|----------|----------|----------|----------------|  
 contracts/ | 100 | 100 | 100 | 100 | |  
 Lock.sol | 100 | 100 | 100 | 100 | |  
------------|----------|----------|----------|----------|----------------|  
All files | 100 | 100 | 100 | 100 | |  
------------|----------|----------|----------|----------|----------------|  
Conclusion  
In this tutorial, you've learned how to profile and analyze the test coverage of your smart contracts' test suite. You learned how to visualize the coverage report and improve the coverage of the test suite by using the Solidity Coverage plugin.  
See also  
Tags:smart contracts  
ON THIS PAGE  
Overview  
Setting up the Solidity Coverage plugin  
My first test coverage  
Increasing test coverage  
Conclusion  
See also  
See also  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/intro-to-foundry-setup  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Foundry: Setting up Foundry with Base  
In this tutorial, you'll learn how to set up Foundry, a toolchain for smart contract development. You'll also learn how to configure it to work with Base.  
Objectives  
By the end of this tutorial, you should be able to:  
Install Foundry  
Create a Foundry project  
Compile a smart contract using Foundry  
Configure Foundry to work with Base  
Overview  
Foundry is a smart contract development toolchain that is composed of multiple small command line tools:  
forge: Compile, test, and deploy your smart contracts  
cast: Interact with the Blockchain over RPC. You can make smart contract calls, send transactions, or retrieve any type of chain data  
chisel: A Solidity REPL. You can write Solidity code directly  
anvil: A local Blockchain node for testing and development  
Using Foundry you can manage your dependencies, compile your project, run tests, deploy smart contracts and interact with the chain from the command-line and via Solidity scripts.  
For a deep dive on the Foundry features and full capabilities, check out the Foundry Book.  
Installing Foundry  
In order to install Foundry, you can use Foundryup, the Foundry's toolchain installer.  
To install Foundryup you have to run in the terminal:  
$ curl -L https://foundry.paradigm.xyz | bash  
After Foundryup is installed, you can install Foundry by running:  
$ foundryup  
You can verify the installation by trying the following commands:  
$ forge --version  
$ cast --version  
$ chisel --version  
$ anvil --version  
My First Foundry Project  
To create a foundry project you can simply run:  
$ forge init hello\_foundry\_in\_base  
This will create a foundry project with the following structure:  
├── lib # all the libraries installed  
├── script # scripts folder, e.g., deploy scripts  
├── src # smart contracts folder  
├── test # tests folder  
└── foundry.toml # foundry configuration file  
You will also notice a .gitsubmodules file -- this is because Foundry handles dependencies using Git submodules.  
By default the Foundry structure stores smart contracts in the src folder. You can change this in the foundry.toml configuration file.  
For instance:  
[profile.default]  
src = 'contracts'  
In order to compile the project, simply run:  
forge build  
Setting up Foundry with Base  
In order to work with Base, you need to configure a couple of settings in the configuration foundry.toml file.  
The first thing is the Solidity version.  
You need to configure your config file as follows:  
[profile.default]  
src = 'src'  
out = 'out'  
libs = ['lib']  
solc\_version = "0.8.23"  
Be sure that you modify the pragma of your contracts and simply run forge build to ensure everything works well.  
We also recommend setting up JSON RPC endpoints for Base and the API key for Basescan in the configuration file so that your environment is ready to deploy your smart contracts.  
Your configuration file should look like the following:  
[profile.default]  
src = "src"  
out = "out"  
libs = ["lib"]  
solc\_version = "0.8.23"  
  
[rpc\_endpoints]  
base = "https://mainnet.base.org"  
baseSepolia = "https://sepolia.base.org"  
  
[etherscan]  
baseSepolia = { key = "${BASESCAN\_API\_KEY}", url = "https://api-sepolia.basescan.org/api" }  
base = { key = "${BASESCAN\_API\_KEY}", url = "https://api.basescan.org/api" }  
We included 2 JSON RPC endpoints for Base and Base Sepolia and similar for the Etherscan section, we included the configuration for Basescan for Sepolia and Mainnet. Both rely on the same API Key, BASESCAN\_API\_KEY.  
Conclusion  
In this tutorial, you've embarked on the journey of smart contract development with Base and Foundry. You've learned the essential steps, from installing Foundry using the convenient Foundryup toolchain installer to creating your first project and configuring Foundry to seamlessly integrate with Base.  
Tags:smart contracts  
ON THIS PAGE  
Overview  
Installing Foundry  
My First Foundry Project  
Setting up Foundry with Base  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/intro-to-foundry-testing  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Foundry: Testing smart contracts  
In this tutorial, you'll learn how to test your smart contracts using Foundry, the toolchain for smart contract development.  
Objectives  
By the end of this tutorial, you should be able to:  
Understand the increased importance of testing in smart contract development  
Write and execute tests written in Solidity using the Forge Standard Library with Foundry  
Use the cheatcodes that Foundry provides to test your smart contracts  
Overview  
Testing is a crucial aspect of smart contract development, ensuring the reliability and security of your code. Because it is impossible to patch a smart contract after deployment, you must thoroughly and completely test your code. Foundry provides a robust testing framework that allows developers to create comprehensive test suites for their projects using Solidity.  
My First Test with Foundry  
Consider the default test that the forge init hello\_foundry\_in\_base command provides in the seed Foundry project.  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "forge-std/Test.sol";  
import "../src/Counter.sol";  
  
contract CounterTest is Test {  
 Counter public counter;  
  
 function setUp() public {  
 counter = new Counter();  
 counter.setNumber(0);  
 }  
  
 function testIncrement() public {  
 counter.increment();  
 assertEq(counter.number(), 1);  
 }  
  
 function testSetNumber(uint256 x) public {  
 counter.setNumber(x);  
 assertEq(counter.number(), x);  
 }  
}  
Take note of the following:  
Foundry test files are named following the pattern: <ContractName>.t.sol  
Smart contract test files are named following the pattern: <ContractName>Test  
All tests inherit from forge-std/Test.sol.  
All tests contain a public function called setUp, which is executed before each test. This is similar to the beforeEach hook in the Mocha/Typescript world.  
Test cases start with the test keyword, for instance testIncrement.  
Test cases functions are public.  
For more information about writing tests in Foundry, you can follow the official guide for Writing tests  
In order to run the test in Foundry, run:  
$ forge test  
You should see in the terminal:  
Running 2 tests for test/Counter.t.sol:CounterTest  
[PASS] testIncrement() (gas: 28334)  
[PASS] testSetNumber(uint256) (runs: 256, μ: 27565, ~: 28343)  
Test result: ok. 2 passed; 0 failed; finished in 13.57ms  
Using Cheatcodes  
Foundry includes a set of cheatcodes, which are special instructions that are accessible using the vm instance in your tests. Cheatcodes allow you to perform various tasks, including:  
Manipulate the state of the blockchain  
Test reverts  
Test events  
Change block number  
Change identity  
And more!  
To start, use a cheatcode to modify the msg.sender of your tests, and add some console logs via importing the forge-std/console.sol contract.  
The Counter contract should look as follows:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "forge-std/console.sol";  
  
contract Counter {  
 uint256 public number;  
  
 function setNumber(uint256 newNumber) public {  
 console.log("The sender is %s", msg.sender);  
 number = newNumber;  
 }  
  
 function increment() public {  
 console.log("The sender is %s", msg.sender);  
 number++;  
 }  
}  
If you run the tests using forge test, you will see the following:  
Running 2 tests for test/Counter.t.sol:CounterTest  
[PASS] testIncrement() (gas: 31531)  
[PASS] testSetNumber(uint256) (runs: 256, μ: 30684, ~: 31540)  
Test result: ok. 2 passed; 0 failed; finished in 19.64ms  
It seems the logs are not being shown. The reason is because the forge test command includes a flag that enable you to include more details of the logs emitted during the execution of the tests.  
You can control that by including different levels of the verbose flag -- -vv up to -vvvvv. For more details about the level of verbosity you can refer to the Logs and Traces section of the Foundry documentation.  
Run the foundry test -vv. You should see:  
Running 2 tests for test/Counter.t.sol:CounterTest  
[PASS] testIncrement() (gas: 31531)  
Logs:  
 The sender is 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496  
 The sender is 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496  
  
[PASS] testSetNumber(uint256) (runs: 256, μ: 30607, ~: 31540)  
Logs:  
 The sender is 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496  
  
Test result: ok. 2 passed; 0 failed; finished in 17.89ms  
Now, modify the test file using prank cheatcode, which allow you to modify the msg.sender of the next transaction. You will also use the addr cheatcode, which allow you to generate an address using any private key, which can simply be a hex number.  
Include some console.log statements to understand better the execution flow.  
The code should look like:  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.19;  
  
import "forge-std/Test.sol";  
import "../src/Counter.sol";  
  
contract CounterTest is Test {  
 Counter public counter;  
  
 function setUp() public {  
 counter = new Counter();  
 console.log("Calling on Setup");  
 counter.setNumber(0);  
 }  
  
 function testIncrement() public {  
 console.log("Calling on testIncrement");  
 vm.prank(vm.addr(0x01));  
 counter.increment();  
 assertEq(counter.number(), 1);  
 }  
  
 function testSetNumber(uint256 x) public {  
 console.log("Calling on testSetNumber");  
 vm.prank(vm.addr(0x02));  
 counter.setNumber(x);  
 assertEq(counter.number(), x);  
 }  
}  
Then if you run the forge test -vv command, you should see:  
Running 2 tests for test/Counter.t.sol:CounterTest  
[PASS] testIncrement() (gas: 35500)  
Logs:  
 Calling on Setup  
 The sender is 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496  
 Calling on testIncrement  
 The sender is 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf  
  
[PASS] testSetNumber(uint256) (runs: 256, μ: 34961, ~: 35506)  
Logs:  
 Calling on Setup  
 The sender is 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496  
  
Test result: ok. 2 passed; 0 failed; finished in 48.75ms  
Notice how you call the cheatcode vm.prank before the call to the counter.increment() and counter.setNumber(x) functions. This allows you to specify a particular address to become the msg.sender in the contract. Since the vm.prank accepts an address, you simply generate an address using the cheatcode vm.addr, where you pass a simple hexadecimal number, which is a valid private key.  
Conclusion  
Congratulations! You've successfully completed your first step in your journey of testing smart contracts using Foundry. As you move forward, keep exploring its rich features and functionalities. The ability to write comprehensive tests and leverage cheatcodes ensures the reliability and security of your smart contracts.  
Happy coding and testing with Foundry!  
Tags:smart contracts  
ON THIS PAGE  
Overview  
My First Test with Foundry  
Using Cheatcodes  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/intro-to-providers  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Introduction to Providers  
This tutorial provides an introduction to providers and shows you how to connect your frontend to the blockchain using JSON RPC blockchain providers, and the RainbowKit, wagmi, and viem stack.  
Objectives  
By the end of this tutorial, you should be able to:  
Compare and contrast public providers vs. vendor providers vs. wallet providers  
Select the appropriate provider for several use cases  
Set up a provider in wagmi and use it to connect a wallet  
Protect API keys that will be exposed to the front end  
Prerequisites  
1. Be familiar with modern, frontend web development  
In this tutorial, we'll be working with a React frontend built with Next.js. While you don't need to be an expert, we'll assume that you're comfortable with the basics.  
2. Possess a general understanding of the EVM and smart contract development  
This tutorial assumes that you're reasonably comfortable writing basic smart contracts. If you're just getting started, jump over to our Base Learn guides and start learning!  
Types of Providers  
Onchain apps need frontends, sometimes called dApps, to enable your users to interact with your smart contracts. A provider makes the connection from frontend to the blockchain, and is used to read data and send transactions.  
In blockchain development, the term provider describes a company or service that provides an API enabling access to the blockchain as a service. This is distinct from the providers you wrap your app in using the React Context API, though you'll use one of those to pass your blockchain provider deeply into your app.  
These services enable interacting with smart contracts without the developer needing to run and maintain their own blockchain node. Running a node is expensive, complicated, and challenging. In most cases, you'll want to start out with a provider. Once you start to get traction, you can evaluate the need to run your own node, or switch to a more advanced architecture solution, such as utilizing Subgraph.  
Figuring out which type of provider to use can be a little confusing at first. As with everything blockchain, the landscape changes rapidly, and search results often return out-of-date information.  
INFO  
New onchain devs sometimes get the impression that there are free options for connecting their apps to the blockchain. Unfortunately, this is not really true. Blockchain data is still 1's and 0's, fetched by computation and served to the internet via servers.  
It costs money to run these, and you will eventually need to pay for the service.  
You'll encounter providers divided into three general categories: Public Providers, Wallet Providers, and Vendor Providers  
Public Providers  
Many tutorials and guides, including the getting started guide for wagmi, use a Public Provider as the default to get you up and running. Public means that they're open, permissionless, and free, so the guides will also usually warn you that you need to add another provider if you don't want to run into rate limiting. Listen to these warnings! The rate-limits of public providers are severe, and you'll start getting limited very quickly.  
In wagmi, a public client is automatically included in the default confit. This client is just a wrapper setting up a [JSON RPC] provider using the chain and rpcUrls listed in Viem's directory of chain information. You can view the data for Base Sepolia here.  
Most chains will list this information in their docs as well. For example, on the network information pages for Base and Optimism. If you wanted, you could manually set up a jsonRpcProvider in wagmi using this information.  
Wallet Providers  
Many wallets, including Coinbase Wallet and MetaMask, inject an Ethereum provider into the browser, as defined in EIP-1193. The injected provider is accessible via window.ethereum.  
Under the hood, these are also just JSON RPC providers. Similar to public providers, they are rate-limited.  
Older tutorials for early libraries tended to suggest using this method for getting started, so you'll probably encounter references to it. However, it's fallen out of favor, and you'll want to use the public provider for your initial connection experiments.  
Vendor Providers  
A growing number of vendors provide access to blockchain nodes as a service. Visiting the landing pages for QuickNode, Alchemy, or Coinbase Developer Platform (CDP) can be a little confusing. Each of these vendors provides a wide variety of services, SDKs, and information.  
Luckily, you can skip most of this if you're just trying to get your frontend connected to your smart contracts. You'll just need to sign up for an account, and get an endpoint, or a key, and configure your app to connect to the provider(s) you choose.  
It is worth digging in to get a better understanding of how these providers charge you for their services. The table below summarizes some of the more important API methods, and how you are charged for them by each of the above providers.  
Note that the information below may change, and varies by network. Each provider also has different incentives, discounts, and fees for each level of product. They also have different allowances for calls per second, protocols, and number of endpoints. Please check the source to confirm!  
Alchemy Costs QuickNode Costs CDP Costs  
Free Tier / Mo. 3M compute units 50M credits 500M billing units  
Mid Tier / Mo. 1.5B CUs @ $199 3B credits @ $299 Coming soon  
eth\_blocknumber 10 20 30  
eth\_call 26 20 30  
eth\_getlogs 75 20 100  
eth\_getbalance 19 20 30  
To give you an idea of usage amounts, a single wagmi useContractRead hook set to watch for changes on a single view via a TanStack query and useBlockNumber will call eth\_blocknumber and eth\_call one time each, every 4 seconds.  
Connecting to the Blockchain  
RainbowKit is a popular library that works with wagmi to make it easy to connect, disconnect, and change between multiple wallets. It's batteries-included out of the box, and allows for a great deal of customization of the list of wallets and connect/disconnect button.  
You'll be using RainbowKit's quick start to scaffold a new project for this tutorial. Note that at the time of writing, it does not use the Next.js app router. See Building an Onchain App if you wish to set this up instead.  
INFO  
The script doesn't allow you to use . to create a project in the root of the folder you run it from, so you'll want to run it from your src directory, or wherever you keep your project folders.  
It will create a folder with the project name you give, and create the files inside.  
Open up a terminal and run:  
yarn create @rainbow-me/rainbowkit  
Give your project a name, and wait for the script to build it. It will take a minute or two.  
DANGER  
If you get an error because you are on the wrong version of node, change to the correct version then delete everything and run the script again.  
Scaffolded App  
Open your new project in the editor of your choice, and open pages/\_app.tsx. Here, you'll find a familiar Next.js app wrapped in context providers for the TanStack QueryProvider, RainbowKit, and wagmi.  
function MyApp({ Component, pageProps }: AppProps) {  
 return (  
 <WagmiProvider config={config}>  
 <QueryClientProvider client={client}>  
 <RainbowKitProvider>  
 <Component {...pageProps} />  
 </RainbowKitProvider>  
 </QueryClientProvider>  
 </WagmiProvider>  
 );  
}  
Note that these providers are using React's context feature to pass the blockchain providers and configuration into your app. It can be confusing to have the word provider meaning two different things in the same file, or even the same line of code!  
Before you can do anything else, you need to obtain a WalletConnect projectId.  
Open up the WalletConnect homepage, and create an account, and/or sign in using the method of your choice.  
Click the Create button in the upper right of the Projects tab.  
Enter a name for your project, select the App option, and click Create.  
Copy the Project ID from the project information page, and paste it in as the projectId in getDefaultWallets.  
const { connectors } = getDefaultWallets({  
 appName: 'RainbowKit App',  
 projectId: 'YOUR\_PROJECT\_ID',  
 chains,  
});  
CAUTION  
Remember, anything you put on the frontend is public! That includes this id, even if you use environment variables to better manage this type of data. Next.js reminds you of the risk, by requiring you to prepend NEXT\_PUBLIC\_ to any environment variables that can be read by the browser.  
Before you deploy, make sure you configure the rest of the items in the control panel to ensure only your site can use this id.  
Public Provider  
By default, the setup script will configure your app to use the built-in public provider, and connect to a number of popular chains. To simply matters, remove all but mainnet and base.  
const config = getDefaultConfig({  
 appName: 'RainbowKit App',  
 projectId: 'YOUR\_APP\_ID\_HERE',  
 chains: [mainnet, base],  
 ssr: true,  
});  
Open the terminal and start the app with:  
yarn run dev  
Click the Connect Wallet button, select your wallet from the modal, approve the connection, and you should see your network, token balance, and address or ENS name at the top of the screen. Select your wallet from the modal.  
You've connected with the Public Provider!  
QuickNode  
To select your provider(s), you'll use createConfig instead of getDefaultConfig. The transports property allows you to configure how you wish to connect with multiple networks. If you need more than one connector for a given network, you can use [fallbacks].  
First, set up using QuickNode as your provider. Replace your import of the default config from RainbowKit with createConfig and http from wagmi:  
import { createConfig, http, WagmiProvider } from 'wagmi';  
// ...Chains import  
import { RainbowKitProvider } from '@rainbow-me/rainbowkit';  
You'll need an RPC URL, so open up QuickNode's site and sign up for an account if you need to. The free tier will be adequate for now, you may need to scroll down to see it. Once you're in, click Endpoints on the left side, then click + Create Endpoint.  
On the next screen, you'll be asked to select a chain. Each endpoint only works for one. Select Base, click Continue.  
For now, pick Base Mainnet, but you'll probably want to delete this endpoint and create a new one for Sepolia when you start building. The free tier only allows you to have one at a time.  
If you haven't already picked a tier, you'll be asked to do so, then you'll be taken to the endpoints page, which will display your endpoints for HTTP and WSS.  
CAUTION  
As with your WalletConnect Id, these endpoints will be visible on the frontend. Be sure to configure the allowlist!  
Use this endpoint to add an http transport to your config:  
const config = createConfig({  
 chains: [mainnet, base],  
 ssr: true,  
 transports: {  
 [base.id]: http('YOUR PROJECT URL'),  
 [mainnet.id]: http('TODO'),  
 },  
});  
Now, the app will use your QuickNode endpoint for the Base network. Note that you don't need an app name or WalletConnect Id, because you are no longer using WalletConnect.  
To test this out, switch networks a few times. You'll know it's working if you see your balance when Base is the selected network. You haven't added mainnet, so you'll get an error in the console and no balance when you switch to that.  
Alchemy  
Alchemy is no longer baked into wagmi, but it still works the same as any other RPC provider. As with QuickNode, you'll need an account and a key. Create an account and/or sign in, navigate to the Apps section in the left sidebar, and click Create new app.  
Select Base Mainnet, and give your app a name.  
CAUTION  
Once again, remember to configure the allowlist when you publish your app, as you'll be exposing your key to the world!  
On the dashboard for your new app, click the API key button, and copy the HTTPS link to the clipboard. Replace your todo with this link:  
const config = createConfig({  
 chains: [mainnet, base],  
 ssr: true,  
 transports: {  
 [base.id]: http('YOUR PROJECT URL'),  
 [mainnet.id]: http('ALCHEMY HTTP URL'),  
 },  
});  
As before, you can confirm the Alchemy Provider is working by running the app and changing the network. You should now no longer get an error and should be able to see your balance for Ethereum mainnet.  
Conclusion  
In this tutorial, you've learned how Providers supply blockchain connection as a service, eliminating the need for developers to run and maintain their own nodes. You also learned how to connect your app to the blockchain using several different providers, including the public provider(s).  
Tags:nodesfrontend  
ON THIS PAGE  
Prerequisites  
Types of Providers  
Public Providers  
Wallet Providers  
Connecting to the Blockchain  
Scaffolded App  
Public Provider  
QuickNode  
Conclusion  
Scaffolded App  
Public Provider  
QuickNode  
Alchemy  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/minting-nfts-with-zora  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
How to Mint on Zora with an App  
Zora's mission is to make creating on the internet free and valuable. To support this, they've made a number of no-code tools, SDKs, and contracts that make creating NFTs easier. They're also a part of the superchain ecosystem, which means most of their tools also work on Base!  
You can interact with Zora's contracts through your own frontend, which makes it easier to create secure, efficient, and feature-rich minting experiences for your users.  
For this tutorial, you'll use Zora's gasless 1155 premint with [An Onchain App in 100 Components] to allow not-yet-onchain artists to use your app to create and share NFT collections without needing a funded wallet. And because the template includes the Coinbase Smart Wallet, you can help these users create their first wallet and automatically receive their rewards and payments without them needing to figure anything out.  
In doing so, you'll help grow the ecosystem of Base users for everyone!  
Objectives  
By the end of this tutorial you should be able to:  
Programmatically use Zora's gasless 1155 premint create ERC-1155 minting experiences on a frontend  
Use the connected wallet address to receive minting rewards and payments  
Prerequisites  
ERC-1155 Tokens  
This tutorial assumes that you are familiar with the properties of ERC-1155 tokens, but you don't need to know how to write one.  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. Check out our tutorial on deploying with Vercel if you need a refresher!  
Onchain Apps  
The tutorial assumes you're comfortable with the basics of deploying an app and connecting it to a smart contract. If you're still learning this part, check out our tutorials in Base Learn for Building an Onchain App.  
Getting Started  
Begin by making a copy of the Onchain App Template by clicking the Use this Template button then cloning it locally.  
The team recommends using [Bun], so install it if you need to, then install the packages and run the app:  
# Install bun in case you don't have it  
curl -fsSL <https://bun.sh/install> | bash  
  
# Install packages  
bun i  
  
# Run Next app  
bun run dev  
Navigate to localhost:3000 and confirm the app is working.  
Building the App  
This tutorial won't cover all of the frontend development, auth, databases, or other details of making a production app, but it will walk you through the major pieces of enabling your users to use your app to:  
Create a Coinbase Smart Wallet, if they need one  
Create an NFT collection, gaslessly with Premint  
Add tokens to that collection  
Allow other people to mint the tokens  
Begin by opening src/app/page.tsx and doing some cleanup. Delete everything expect the wallet integration, and add some copy that is friendly to non-crypto-native users:  
'use client';  
import WalletComponents from '@/components/WalletComponents';  
  
export default function Page() {  
 return (  
 <div className="flex w-96 flex-col md:w-[600px]">  
 <section className="mb-6 flex w-full flex-col border-b border-sky-800 pb-6">  
 <aside className="mb-6 flex">  
 <h2 className="text-xl">Wallet</h2>  
 </aside>  
 <p className="text-body text-white">  
 Welcome! Please sign in with your wallet. If you are new at this, you can create a new  
 wallet by clicking the "Connect wallet" button and following the instructions!  
 </p>  
 <WalletComponents />  
 </section>  
 </div>  
 );  
}  
You can also delete the files for the components you removed. You still need OnchainProviders.tsx and WalletComponents.tsx.  
Creating the Premint from the App  
INFO  
Gas on Base is currently inexpensive enough that we're doing this tutorial on the live network! Zora supports Base Sepolia as well, so feel free to use that instead.  
Open src/app/wagmi.ts and convert the references of baseSepolia to base.  
Install the Zora Protocol SDK.  
bun add @zoralabs/protocol-sdk viem@2.x  
In src/app/components add a file called CreatePremint.tsx. Open CreatePremint.tsx.  
Import dependencies, instantiate a client, and stub out a component.  
import { createCreatorClient } from '@zoralabs/protocol-sdk';  
import { useEffect, useState } from 'react';  
import { useAccount, useChainId, usePublicClient, useSignTypedData } from 'wagmi';  
  
export default function CreatePremint() {  
 const chainId = useChainId();  
 const publicClient = usePublicClient()!;  
 const { address: creatorAddress } = useAccount();  
  
 const creatorClient = createCreatorClient({ chainId, publicClient });  
  
 const { signTypedData, data: signature } = useSignTypedData();  
  
 return <div>TODO</div>;  
}  
This tutorial isn't going to cover the management of storing and retrieving all of your users' premints or active mints, so add some debug state variables to hold the necessary data for the mint this example will create.  
// Debug to store collection info, stand-in for database, etc.  
const [debugGlobalAddress, setDebugGlobalAddress] = useState<string | null>(null);  
const [debugGlobalUid, setDebugGlobalUid] = useState<number | null>(null);  
You'll also need some state variables to capture the premint data and functions created by creatorClient.createPremint.  
const [premintConfig, setPremintConfig] = useState<any>(null);  
const [collectionAddress, setCollectionAddress] = useState<string | null>(null);  
const [typedDataDefinition, setTypedDataDefinition] = useState<any>(null);  
const [submit, setSubmit] = useState<Function | null>(null);  
Finally, you need a way to collect all of the data needed to create the mint. Add a type and use it for the CreatePremint components props.  
export type PremintProps = {  
 contractName: string;  
 maxSupply: bigint;  
 maxTokensPerAddress: bigint;  
 mintStart: bigint;  
 mintDuration: bigint;  
 pricePerToken: bigint;  
};  
  
export default function CreatePremint({  
 contractName,  
 maxSupply,  
 maxTokensPerAddress,  
 mintStart,  
 mintDuration,  
 pricePerToken,  
}: PremintProps) {  
 // Existing code...  
}  
The example in the premint docs goes into detail about the parameters, but they should be largely self-evident.  
Open src/app/page.tsx and add a debug instance of your props. As you develop the app, replace this with data gathered from other components to allow your users to upload images and enter the name, etc. into a form that you pin to IPFS for them.  
const debugPremintProps = {  
 contractName: 'Pixel Seasons',  
 contractURI: 'ipfs://QmYjwarNweXhQAfu3phirz8vwnwFEqo5t8m3xt3HWpFd8N',  
 tokenURI: 'ipfs://QmXr9NuvX9afZhHTpd2jRgFHbVnaWPKD315AtnTT6H67hz',  
 maxSupply: 1000n,  
 maxTokensPerAddress: 5n,  
 mintStart: 0n,  
 mintDuration: 0n,  
 pricePerToken: 0n,  
};  
The contractURI is a file pinned to IPFS containing contract metadata, such as:  
{  
 "name": "Pixel Seasons",  
 "description": "Retro images representing each season.",  
 "image": "ipfs://QmaSaa3VkFiywQi2HmE1YmuXZCfp992Awso9SquEnYpcC1"  
}  
The tokenURI is an IPFS link to valid NFT metadata, such as:  
{  
 "name": "Pixel Summer",  
 "description": "",  
 "image": "ipfs://Qmb8EYToz59cGExgaGyMxuGf22WtrNYXo5cNRApKrkaxdB",  
 "content": {  
 "mime": "image/jpg",  
 "uri": "ipfs://Qmb8EYToz59cGExgaGyMxuGf22WtrNYXo5cNRApKrkaxdB"  
 }  
}  
Add your component to the page and pass it the debug data:  
<section className="mb-6 flex w-full flex-col border-b border-sky-800 pb-6">  
 <aside className="mb-6 flex">  
 <h2 className="text-xl">Create Premint</h2>  
 </aside>  
 <p className="text-body text-white">  
 You can use this component to set up a premint for your non-engineer customers. After the  
 premint is created, it will demonstrate how their customers can mint the NFT.  
 </p>  
 <p className="text-body text-white">  
 For a production app, put a form here to collect info and upload images.  
 </p>  
 <br />  
 <CreatePremint {...debugPremintProps} />  
</section>  
Return to CreatePremint.tsx. createPremint is an asynchronous function, so you'll need to wrap it in a useEffect to trigger it and update your state when the createorAddress is fetched by useAccount.  
Be sure to put your address for createReferral. Zora will pay you a share of the revenue automatically to this address for your part in helping create the mint!  
useEffect(() => {  
 async function createPremint() {  
 const {  
 premintConfig: pC,  
 collectionAddress: cA,  
 typedDataDefinition: tDD,  
 submit: sub,  
 } = await creatorClient.createPremint({  
 // info of the 1155 contract to create.  
 contract: {  
 // the account that will be the admin of the collection.  
 // Must match the signer of the premint.  
 contractAdmin: creatorAddress!,  
 contractName,  
 contractURI,  
 },  
 // token info of token to create  
 token: {  
 tokenURI,  
 // Put your address as `createReferral`, you get rewards!  
 createReferral: '<YOUR ADDRESS HERE>',  
 maxSupply,  
 maxTokensPerAddress,  
 mintStart,  
 mintDuration,  
 pricePerToken,  
 payoutRecipient: creatorAddress!,  
 },  
 });  
  
 setPremintConfig(pC);  
 setCollectionAddress(cA);  
 setTypedDataDefinition(tDD);  
 setSubmit(() => sub); // Ensure submit is set as a function  
 }  
  
 if (creatorAddress) {  
 createPremint();  
 }  
}, [creatorAddress]);  
You've decomposed a signTypedData function and data, renamed as signature, from the useSignTypedData hook. The signature will be updated when that function is called. You can use this with useEffect to trigger sending the signed message to Zora with the submit function you decomposed from createPremint. This is also a good place to set your debug variables, or however you choose capturing the contract information in a production app:  
useEffect(() => {  
 if (signature) {  
 if (submit) {  
 submit({  
 signature,  
 });  
 // Debug to store collection info  
 setDebugGlobalAddress(collectionAddress);  
 setDebugGlobalUid(premintConfig.uid);  
 } else {  
 console.error('Submit function is not set.');  
 }  
 }  
}, [signature]);  
Finally, set up your return to show some instructions and the Create Premint button if the premint is not yet created, and some information about the mint if it has been:  
if (!debugGlobalAddress || !debugGlobalUid) {  
 return (  
 <main className="flex h-10 items-center space-x-4">  
 <button  
 className="rounded bg-blue-500 px-4 py-2 font-bold text-white hover:bg-blue-700"  
 onClick={() => signTypedData(typedDataDefinition)}  
 >  
 Create Premint  
 </button>  
 </main>  
 );  
} else {  
 return (  
 <main className="flex h-10 items-center space-x-4">  
 <p>Collection Address: {debugGlobalAddress}</p>  
 <p>UID: {debugGlobalUid}</p>  
 </main>  
 );  
}  
Test the app! You should be able to sign the message, and see the address and uid for you mint after the message is signed.  
Minting from the Premint  
Add a new file in src/app/components called ZoraCollectPremint.tsx and open it.  
Create a stub with dependencies that instantiates a collectorClient instance:  
import { createCollectorClient } from '@zoralabs/protocol-sdk';  
import { useAccount, useChainId, usePublicClient, useWriteContract } from 'wagmi';  
  
export type CollectPremintProps = {  
 contractAddress: string;  
 uid: number;  
};  
  
export default function ZoraCollectPremint({ contractAddress, uid }: CollectPremintProps) {  
 const chainId = useChainId();  
 const publicClient = usePublicClient()!;  
  
 const { address: minterAccount } = useAccount();  
 const { writeContract } = useWriteContract();  
  
 const collectorClient = createCollectorClient({ chainId, publicClient });  
  
 return <div>TODO</div>;  
}  
Next, add an async function to collectPremint. This will use collectorClient.mint to create a transaction that will deploy the NFT contract, if it hasn't been deployed yet, and claim quantityToMint NFTs. It then uses writeContract from wagmi to execute the transaction for the user to sign.  
async function collectPremint() {  
 if (!minterAccount) {  
 console.error('No minter account');  
 return;  
 }  
 const { parameters } = await collectorClient.mint({  
 tokenContract: contractAddress as `0x${string}`,  
 mintType: 'premint',  
 uid,  
 quantityToMint: 1,  
 mintComment: 'Neat!!!',  
 minterAccount,  
 });  
  
 writeContract(parameters);  
}  
Update the return to return a button that calls the function with the props for the component:  
return (  
 <button  
 className="rounded bg-blue-500 px-4 py-2 font-bold text-white hover:bg-blue-700"  
 onClick={collectPremint}  
 >  
 Collect Premint  
 </button>  
);  
Return to CreatePremint.tsx. You wouldn't necessarily want to use this architecture for a production app since the creator probably doesn't want to be the first minter, but this is the most convenient place to test the component.  
Add an instance of ZoraCollectPremint to the return for after the premint has been created and pass it the debug variables:  
return (  
 <main className="flex h-10 items-center space-x-4">  
 <p>Collection Address: {debugGlobalAddress}</p>  
 <p>UID: {debugGlobalUid}</p>  
 <br />  
 <ZoraCollectPremint contractAddress={debugGlobalAddress} uid={debugGlobalUid} />  
 </main>  
);  
Test out the app. When you click the button, you should get a transaction to mint the NFT. Finally, open your Zora Dashboard. Select Base as the network. You'll see your collection, and the rewards you've received from Zora for facilitating the creation of the collection and the mint of the NFT!  
Conclusion  
In this tutorial, you've learned how to build an experience using the Zora Protocol SDK where non-crypto-native creators can create NFT collections without needing to fund a wallet and without you needing to subsidize their gas. You've also learned how to create a mint UI/UX from these premints, and how to collect your own rewards from Zora.  
Tags:nftsmart wallet  
ON THIS PAGE  
Prerequisites  
Getting Started  
ERC-1155 Tokens  
Vercel  
Onchain Apps  
Building the App  
Conclusion  
Creating the Premint from the App  
Minting from the Premint  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/onchain-email-campaigns-using-resend  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Create email campaigns for smart wallets using Resend  
In today’s digital landscape, onchain interactions are becoming increasingly common, but email remains a powerful tool for personal and business communication. When a user logs into your application, capturing their onchain information is just the first step. To build lasting relationships and keep your users engaged, you need to connect with them where they’re most likely to respond—through their inbox.  
This tutorial will guide you through the process of seamlessly prompting users to join your company's mailing list after they sign up with your app. By the end, you'll be equipped to launch effective email campaigns that bridge the gap between onchain activity and offchain communication.  
Prerequisites  
Coinbase Developer Platform (CDP) Account  
You’ll need to set up an account on the Coinbase Developer Platform (CDP). The CDP provides various tools and services for blockchain development, including access to API endpoints and other resources that will be instrumental in your project. Once you’ve created your account, you’ll be ready to move forward with integrating these services into your application.  
WalletConnect Project ID  
You’ll need to set up a cloud account with [WalletConnect], a protocol that enables secure wallet connections across different platforms.  
Resend Account  
You’ll need to set up an account with [Resend], a service that allows you to send email campaigns programmatically. If you don’t already have an account, visit their website and sign up. Once your account is created, you’ll be able to generate API keys, which are essential for integrating Resend with your application.  
After creating your Resend account, navigate to the API keys page within your Resend dashboard. Here, you’ll generate a new API key that will be used to authenticate your requests when sending emails from your application. Make sure to store this key securely, as it will be required in later steps.  
Once your WalletConnect account is set up, log in to obtain an API key. This key allows your application to interact with WalletConnect’s services. Navigate to the API keys section in your WalletConnect dashboard, generate a new key, and store it securely.  
With your WalletConnect API key in hand, it’s time to create a project within the WalletConnect Cloud.  
This project will house your integration settings and project-specific credentials. Go to the projects section of your WalletConnect dashboard and create a new project. After creating your WalletConnect project, you’ll be provided with a unique Project ID. Copy this Project ID and keep it handy, as you’ll need it for the upcoming integration steps.  
You will now set up your development environment.  
INTEGRATING RESEND TO AN EXISTING PROJECT?  
If you’re planning to integrate Resend into an existing project, feel free to skip ahead to the backend section where we’ll create custom API routes for interacting with Resend.  
To begin, you’ll need to fork the OnchainKit App template from GitHub by clicking the green Use this template button. This template provides a solid foundation for building onchain applications and will be used as the base for our demo.  
Once you’ve forked the repository, it’s time to clone it to your local machine. Open your terminal and run the following command, replacing the repository URL with the appropriate one if different:  
git clone git@github.com:<REPLACE-WITH-YOUR-GITHUB-USERNAME>/onchain-app-template.git resend-demo  
This command clones the repository into a directory named resend-demo.  
After cloning the repository, navigate into the project directory using the following command:  
cd resend-demo  
This will switch your terminal’s context to the project’s root directory, where you can begin working on the code.  
The OnchainKit template uses Bun as the package manager. If you don’t have Bun installed, you can install it by running the following command:  
# Install bun in case you don't have it  
bun curl -fsSL <https://bun.sh/install> | bash  
IS BUN INSTALLED?  
After installation, you may need to restart your terminal or run source ~/.bashrc (or ~/.zshrc) to ensure Bun is recognized as a command.  
Next, install the Resend package to handle email campaign functionality within your app:  
bun install resend  
This command adds Resend as a dependency to your project, making it available for use in your application.  
In this templates, environment variables are stored in a .env.example file. You’ll need to rename this file to .env to ensure the environment variables are properly loaded. Run the following command:  
mv .env.local.example .env  
With the .env file in place, open it in your preferred text editor and update it with your API keys and project IDs. These keys are essential for connecting to Resend, WalletConnect, and the Coinbase Developer Platform.  
Here’s an example of how your .env file should look:  
NEXT\_CDP\_API\_KEY="YOUR\_COINBASE\_API\_KEY"  
NEXT\_PUBLIC\_WC\_PROJECT\_ID="YOUR\_WALLET\_CONNECT\_PROJECT\_ID"  
RESEND\_API\_KEY="YOUR\_RESEND\_API\_KEY"  
RESEND\_AUDIENCE\_ID="YOUR\_RESEND\_AUDIENCE\_ID"  
NOTE  
Make sure to replace the placeholder values (YOUR\_COINBASE\_API\_KEY, etc.) with your actual keys.  
Deploy template to Vercel  
To send emails from your application using Resend, you’ll need to deploy your project to a live environment. Vercel is a popular platform for deploying web applications, and it’s ideal for this purpose. By deploying your cloned repo on Vercel, you’ll obtain a live domain where your app can interact with Resend.  
If you don’t already have a Vercel account, head over to [Vercel’s website] and sign up. You can sign up using your GitHub account, which will make importing your project easier.  
Once logged in, go to the Projects section of your Vercel dashboard. Click on the Add New button to start the process of deploying a new project.  
In the next step, Vercel will prompt you to import a Git repository. Click on Import Git Repository and search for the OnchainKit app that you forked earlier. Select the repository to proceed.  
This step connects your GitHub (or other Git provider) account with Vercel, allowing Vercel to pull the code from your repository.  
PRIVATE REPOS  
If your project is private, you’ll see an option to Configure GitHub App. Click this button to give Vercel the necessary permissions to access your private repository. Follow the prompts to complete the authorization process.  
After Vercel has access to your repository, you’ll be guided through the final deployment steps. Vercel will automatically detect the settings for your project, but you may want to double-check that everything is correct, such as the project name and deployment settings.  
Environment Variables: Before completing the deployment, ensure that your environment variables (from the .env file) are correctly set up in Vercel. Vercel provides an interface to input these variables during the deployment process. Follow Vercel's guide for adding environment variables.  
After the build is complete, Vercel will provide you with a deployment URL. This URL is your live domain, where your application will be hosted. You can visit this URL to see your deployed site in action.  
Set up logic and functionality  
Let's start by removing a few imports to clean our page.tsx file up.  
Start by removing TransactionWrapper and WalletWrapper imports and the code within the second <section/> html element.  
Your src/app/page.tsx file should look like this:  
'use client';  
import Footer from 'src/components/Footer';  
import { ONCHAINKIT\_LINK } from 'src/links';  
import OnchainkitSvg from 'src/svg/OnchainkitSvg';  
import { useAccount } from 'wagmi';  
import LoginButton from '../components/LoginButton';  
import SignupButton from '../components/SignupButton';  
  
export default function Page() {  
 const { address } = useAccount();  
  
 return (  
 <div className="flex h-full w-96 max-w-full flex-col px-1 md:w-[1008px]">  
 <section className="mb-6 mt-6 flex w-full flex-col md:flex-row">  
 <div className="flex w-full flex-row items-center justify-between gap-2 md:gap-0">  
 <a href={ONCHAINKIT\_LINK} title="onchainkit" target="\_blank" rel="noreferrer">  
 <OnchainkitSvg />  
 </a>  
 <div className="flex items-center gap-3">  
 <SignupButton />  
 {!address && <LoginButton />}  
 </div>  
 </div>  
 </section>  
 <section className="templateSection flex w-full flex-col items-center justify-center gap-4 rounded-xl bg-gray-100 px-2 py-4 md:grow"></section>  
 <Footer />  
 </div>  
 );  
}  
Use react hooks and create the following state variables  
import { useState, useEffect } from 'react';  
src/app/page.tsx:  
src/app/page.tsx  
const [showForm, setShowForm] = useState(false);  
const [formData, setFormData] = useState({ firstName: '', lastName: '', email: '' });  
const [isSubscribed, setIsSubscribed] = useState(false);  
const [isMember, setIsMember] = useState(false);  
Add useEffect hook: Add this useEffect hook to update form visibility and member status based on wallet connection:  
src/app/page.tsx  
useEffect(() => {  
 if (address) {  
 setShowForm(true);  
 } else {  
 setShowForm(false);  
 setIsMember(false);  
 }  
}, [address]);  
Add form submission handler: Add the handleSubscribe function to handle form submissions:  
src/app/page.tsx  
const handleSubscribe = async (e: React.FormEvent) => {  
 e.preventDefault();  
 try {  
 const { firstName, email } = formData;  
  
 // Send email using API  
 const emailResponse = await fetch('/api/send', {  
 method: 'POST',  
 headers: { 'Content-Type': 'application/json' },  
 body: JSON.stringify({ firstName, email }),  
 });  
 const emailData = await emailResponse.json();  
 if (!emailResponse.ok) throw new Error(emailData.error || 'Failed to send email');  
  
 // Create contact using API  
 const contactResponse = await fetch('/api/create', {  
 method: 'POST',  
 headers: { 'Content-Type': 'application/json' },  
 body: JSON.stringify({ firstName, email }),  
 });  
 const contactData = await contactResponse.json();  
 if (!contactResponse.ok) throw new Error(contactData.error || 'Failed to create contact');  
  
 console.log('Subscription successful:', { emailData, contactData });  
 setIsSubscribed(true);  
 setIsMember(true);  
  
 // Close the form after 3 seconds  
 setTimeout(() => {  
 setShowForm(false);  
 setIsSubscribed(false);  
 }, 3000);  
 } catch (error) {  
 console.error('Error subscribing:', error);  
 alert(error instanceof Error ? error.message : 'An unknown error occurred');  
 }  
};  
Add input change handler: Add the handleChange function to handle input changes:  
src/app/page.tsx  
const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {  
 setFormData({ ...formData, [e.target.name]: e.target.value });  
};  
Add outside click handler: Add the handleOutsideClick function to close the form when clicking outside:  
src/app/page.tsx  
const handleOutsideClick = (e: React.MouseEvent<HTMLDivElement>) => {  
 if (e.target === e.currentTarget) {  
 setShowForm(false);  
 }  
};  
Create a component that will serve as the email template:  
src/components/EmailTemplate.tsx:  
src/components/EmailTemplate.tsx  
import \* as React from 'react';  
  
interface EmailTemplateProps {  
 firstName: string;  
}  
export const EmailTemplate: React.FC<Readonly<EmailTemplateProps>> = ({ firstName }) => (  
 <div>  
 <h1>Welcome, {firstName}!</h1>  
 </div>  
);  
In src/app/page.tsx add the following section to display wether the user is a member or not:  
<section  
 className="templateSection flex w-full flex-col items-center justify-center gap-4 rounded-xl bg-gray-100 px-2 py-4 md:grow"  
>  
 <div  
 className="flex h-[450px] w-[450px] max-w-full items-center justify-center rounded-xl bg-[#030712]"  
 >  
 <div className="flex flex-col items-center justify-center gap-4">  
 <h1 className="text-2xl font-bold text-white font-inter">  
 {isMember ? "You're now a member." : "Welcome to Smart Wallet"}  
 </h1>  
 <p className="text-sm text-white font-inter">  
 {isMember ? "Thank you for joining!" : "Demo app to showcase the Smart Wallet with Resend"}  
 </p>  
 </div>  
 </div>  
</section>  
In the same file (src/app/page.tsx) add the following logic to display the form after the last section/> element:  
{ showForm && (  
<div  
 className="fixed inset-0 flex items-center justify-center bg-black bg-opacity-50"  
 onClick="{handleOutsideClick}"  
>  
 <div className="w-80 rounded-lg bg-white p-8 shadow-lg">  
 {isSubscribed ? (  
 <h2 className="text-center text-2xl font-bold text-black">Subscribed!</h2>  
 ) : (  
 <form onSubmit="{handleSubscribe}" className="space-y-4">  
 <h2 className="mb-6 text-center text-2xl font-bold text-gray-800">Join our mailing list</h2>  
 {/\* Form inputs \*/}  
 <div className="space-y-2">  
 <input  
 type="text"  
 name="firstName"  
 placeholder="First Name"  
 value="{formData.firstName}"  
 onChange="{handleChange}"  
 className="focus:black w-full rounded-md border border-gray-300 p-3 text-sm focus:outline-none focus:ring-2"  
 required  
 />  
 <input  
 type="text"  
 name="lastName"  
 placeholder="Last Name"  
 value="{formData.lastName}"  
 onChange="{handleChange}"  
 className="focus:black w-full rounded-md border border-gray-300 p-3 text-sm focus:outline-none focus:ring-2"  
 required  
 />  
 <input  
 type="email"  
 name="email"  
 placeholder="Email"  
 value="{formData.email}"  
 onChange="{handleChange}"  
 className="focus:black w-full rounded-md border border-gray-300 p-3 text-sm focus:outline-none focus:ring-2"  
 required  
 />  
 </div>  
 <button  
 type="submit"  
 className="w-full rounded-md bg-blue-600 p-3 text-sm font-semibold text-white transition duration-300 ease-in-out hover:bg-blue-700"  
 >  
 Subscribe  
 </button>  
 </form>  
 )}  
 </div>  
</div>  
) }  
Now, let's set up our API routes for creating contacts and sending emails. In your project's app directory, create a new folder called api. Inside this api folder, create two more folders: create and send.  
In the create folder, we'll create a file named route.ts. This route will handle creating a new contact using the Resend API. In the send folder, create another route.ts file. This route will be responsible for sending an email to the user, also using the Resend API.  
These two routes will work together to add a new subscriber to your list and send them a welcome email.  
src/app/api/send/route.ts:  
src/app/api/send/route.ts  
import { EmailTemplate } from '../../../components/EmailTemplate';  
import { Resend } from 'resend';  
  
const resend = new Resend(process.env.RESEND\_API\_KEY);  
  
export async function POST() {  
 try {  
 const { data, error } = await resend.emails.send({  
 from: 'Acme <onboarding@resend.dev>',  
 to: ['delivered@resend.dev'],  
 subject: 'Hello world',  
 react: EmailTemplate({ firstName: 'John' }),  
 });  
  
 if (error) {  
 return Response.json({ error }, { status: 500 });  
 }  
  
 return Response.json(data);  
 } catch (error) {  
 return Response.json({ error }, { status: 500 });  
 }  
}  
src/app/api/send/route.ts:  
src/app/api/send/route.ts  
import { Resend } from 'resend';  
  
const resend = new Resend(process.env.RESEND\_API\_KEY);  
  
export async function POST(request: Request) {  
 try {  
 const { firstName, email } = await request.json();  
  
 const { data, error } = await resend.contacts.create({  
 email,  
 firstName,  
 audienceId: process.env.RESEND\_AUDIENCE\_ID || '',  
 });  
  
 if (error) {  
 console.error('Contact creation error:', error);  
 return Response.json({ error: 'Failed to create contact' }, { status: 500 });  
 }  
  
 return Response.json({ success: true, data });  
 } catch (error) {  
 console.error('Unexpected error:', error);  
 return Response.json({ error: 'An unexpected error occurred' }, { status: 500 });  
 }  
}  
Now that everything is set up, it’s time to test your integration to ensure everything is working as expected.  
Start your development server by running the following command:  
bun run dev  
Open your application in a web browser by navigating to http://localhost:3000 or the port specified in your package.json. You can sign up or log in using your smart wallet. This will trigger the wallet connection process.  
After connecting your wallet, you should see a prompt to join the mailing list. Enter your name and email address in the form provided.  
Once you’ve submitted the form, navigate to your Resend Audience dashboard. You should see a new contact with the name and email information you provided while testing on the development server.  
Conclusion  
Congratulations! You've set up a seamless process to capture user emails after signing in with a Smart Wallet. You can better engage with your users more effectively and build stronger, lasting relationships. Keep exploring the potential of onchain apps and continue enhancing your user experience!  
Tags:account abstraction  
ON THIS PAGE  
Deploy template to Vercel  
Set up logic and functionality  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/oracles-chainlink-price-feeds  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Accessing real-world data using Chainlink Data Feeds  
This tutorial will guide you through the process of creating a smart contract on Base that utilizes Chainlink Data Feeds to access real-world data, such as asset prices, directly from your smart contracts.  
Objectives  
By the end of this tutorial you should be able to do the following:  
Set up a smart contract project for Base using Foundry  
Install the Chainlink smart contracts  
Consume a Chainlink price data feed within your smart contract  
Deploy and test your smart contracts on Base  
Prerequisites  
Foundry  
This tutorial requires you to have Foundry installed.  
From the command-line (terminal), run: curl -L https://foundry.paradigm.xyz | bash  
Then run foundryup, to install the latest (nightly) build of Foundry  
For more information, see the Foundry Book installation guide.  
Coinbase Wallet  
In order to deploy a smart contract, you will first need a wallet. You can create a wallet by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
Wallet funds  
Deploying contracts to the blockchain requires a gas fee. Therefore, you will need to fund your wallet with ETH to cover those gas fees.  
For this tutorial, you will be deploying a contract to the Base Goerli test network. You can fund your wallet with Base Goerli ETH using one of the faucets listed on the Base Network Faucets page.  
What are Chainlink Data Feeds?  
Accurate price data is essential in DeFi applications. However, blockchain networks lack the capability to directly fetch external real-world data, leading to the "Oracle Problem".  
Chainlink Data Feeds offer a solution to this problem by serving as a secure middleware layer that bridges the gap between real-world asset prices and onchain smart contracts.  
Creating a project  
Before you can begin writing smart contracts for Base and consuming Chainlink data feeds, you need to set up your development environment by creating a Foundry project.  
To create a new Foundry project, first create a new directory:  
mkdir myproject  
Then run:  
cd myproject  
forge init  
This will create a Foundry project, which has the following basic layout:  
.  
├── foundry.toml  
├── script  
 │ └── Counter.s.sol  
├── src  
 │ └── Counter.sol  
└── test  
 └── Counter.t.sol  
Installing Chainlink smart contracts  
To use Chainlink's data feeds within your project, you need to install Chainlink smart contracts as a project dependency using forge install.  
To install Chainlink smart contracts, run:  
forge install smartcontractkit/chainlink --no-commit  
Once installed, update your foundry.toml file by appending the following line:  
remappings = ['@chainlink/contracts/=lib/chainlink/contracts']  
Writing and compiling the Smart Contract  
Once your project has been created and dependencies have been installed, you can now start writing a smart contract.  
The Solidity code below defines a smart contract named DataConsumerV3. The code uses the AggregatorV3Interface interface from the Chainlink contracts library to provide access to price feed data.  
The smart contract passes an address to AggregatorV3Interface. This address (0xcD2A119bD1F7DF95d706DE6F2057fDD45A0503E2) corresponds to the ETH/USD price feed on the Base Goerli network.  
INFO  
Chainlink provides a number of price feeds for Base. For a list of available price feeds on Base, visit the Chainlink documentation.  
 // SPDX-License-Identifier: MIT  
 pragma solidity ^0.8.0;  
  
 import "@chainlink/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol";  
  
 contract DataConsumerV3 {  
 AggregatorV3Interface internal priceFeed;  
  
 /\*\*  
 \* Network: Base Goerli  
 \* Aggregator: ETH/USD  
 \* Address: 0xcD2A119bD1F7DF95d706DE6F2057fDD45A0503E2  
 \*/  
 constructor() {  
 priceFeed = AggregatorV3Interface(0xcD2A119bD1F7DF95d706DE6F2057fDD45A0503E2);  
 }  
  
 function getLatestPrice() public view returns (int) {  
 (  
 /\* uint80 roundID \*/,  
 int price,  
 /\* uint startedAt \*/,  
 /\* uint timeStamp \*/,  
 /\* uint80 answeredInRound \*/  
 ) = priceFeed.latestRoundData();  
 return price;  
 }  
 }  
In your project, add the code provided above to a new file named src/DataConsumerV3.sol, and delete the src/Counter.sol contract that was generated with the project. (you can also delete the test/Counter.t.sol and script/Counter.s.sol files).  
To compile the new smart contract, run:  
forge build  
Deploying the smart contract  
Setting up your wallet as the deployer  
Before you can deploy your smart contract to the Base network, you will need to set up a wallet to be used as the deployer.  
To do so, you can use the cast wallet import command to import the private key of the wallet into Foundry's securely encrypted keystore:  
cast wallet import deployer --interactive  
After running the command above, you will be prompted to enter your private key, as well as a password for signing transactions.  
CAUTION  
For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation.  
It is critical that you do NOT commit this to a public repo.  
To confirm that the wallet was imported as the deployer account in your Foundry project, run:  
cast wallet list  
Setting up environment variables for Base Goerli  
To setup your environment for deploying to the Base network, create an .env file in the home directory of your project, and add the RPC URL for the Base Goerli testnet:  
BASE\_GOERLI\_RPC="https://goerli.base.org"  
Once the .env file has been created, run the following command to load the environment variables in the current command line session:  
source .env  
Deploying the smart contract to Base Goerli  
With your contract compiled and environment setup, you are ready to deploy the smart contract to the Base Goerli Testnet!  
For deploying a single smart contract using Foundry, you can use the forge create command. The command requires you to specify the smart contract you want to deploy, an RPC URL of the network you want to deploy to, and the account you want to deploy with.  
To deploy the DataConsumerV3 smart contract to the Base Goerli test network, run the following command:  
forge create ./src/DataConsumerV3.sol:DataConsumerV3 --rpc-url $BASE\_GOERLI\_RPC --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet's private key.  
INFO  
Your wallet must be funded with ETH on the Base Goerli Testnet to cover the gas fees associated with the smart contract deployment. Otherwise, the deployment will fail.  
To get testnet ETH for Base Goerli, see the prerequisites.  
After running the command above, the contract will be deployed on the Base Goerli test network. You can view the deployment status and contract by using a block explorer.  
Interacting with the Smart Contract  
Foundry provides the cast command-line tool that can be used to interact with the smart contract that was deployed and call the getLatestPrice() function to fetch the latest price of ETH.  
To call the getLatestPrice() function of the smart contract, run:  
cast call <DEPLOYED\_ADDRESS> --rpc-url $BASE\_GOERLI\_RPC "getLatestPrice()"  
You should receive the latest ETH / USD price in hexadecimal form.  
Conclusion  
Congratulations! You have successfully deployed and interacted with a smart contract that consumes a Chainlink price feed on the Base blockchain network.  
To learn more about Oracles and using Chainlink to access real-world data within your smart contracts on Base, check out the following resources:  
Oracles  
Chainlink Data Feeds on Base  
Tags:oracles  
ON THIS PAGE  
Prerequisites  
What are Chainlink Data Feeds?  
Foundry  
Coinbase Wallet  
Wallet funds  
Creating a project  
Installing Chainlink smart contracts  
Writing and compiling the Smart Contract  
Deploying the smart contract  
Interacting with the Smart Contract  
Setting up your wallet as the deployer  
Setting up environment variables for Base Goerli  
Deploying the smart contract to Base Goerli  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/oracles-pyth-price-feeds  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Accessing real-time asset data using Pyth Price Feeds  
This tutorial will guide you through the process of creating a smart contract on Base that utilizes Pyth Network oracles to consume a price feed.  
Objectives  
By the end of this tutorial you should be able to do the following:  
Set up a smart contract project for Base using Foundry  
Install the Pyth smart contracts  
Consume a Pyth Network price feed within your smart contract  
Deploy and test your smart contracts on Base  
Prerequisites  
Foundry  
This tutorial requires you to have Foundry installed.  
From the command-line (terminal), run: curl -L https://foundry.paradigm.xyz | bash  
Then run foundryup, to install the latest (nightly) build of Foundry  
For more information, see the Foundry Book installation guide.  
Coinbase Wallet  
In order to deploy a smart contract, you will first need a wallet. You can create a wallet by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
Wallet funds  
Deploying contracts to the blockchain requires a gas fee. Therefore, you will need to fund your wallet with ETH to cover those gas fees.  
For this tutorial, you will be deploying a contract to the Base Sepolia test network. You can fund your wallet with Base Sepolia ETH using one of the faucets listed on the Base Network Faucets page.  
What is Pyth Network?  
Pyth Network focuses on ultra-low latency and real-time data, making it suitable for financial applications that require sub-second updates. Pyth's design emphasizes performance, and it is designed to provide data for a range of traditional and DeFi assets.  
Creating a project  
Before you can begin writing smart contracts for Base and consuming Pyth price feeds, you need to set up your development environment by creating a Foundry project.  
To create a new Foundry project, first create a new directory:  
mkdir myproject  
Then run:  
cd myproject  
forge init  
This will create a Foundry project, which has the following basic layout:  
.  
├── foundry.toml  
├── script  
 │ └── Counter.s.sol  
├── src  
 │ └── Counter.sol  
└── test  
 └── Counter.t.sol  
Installing Pyth smart contracts  
To use Pyth price feeds within your project, you need to install Pyth oracle contracts as a project dependency using forge install.  
To install Pyth oracle contracts, run:  
forge install pyth-network/pyth-sdk-solidity@v2.2.0 --no-git --no-commit  
Once installed, update your foundry.toml file by appending the following line:  
remappings = ['@pythnetwork/pyth-sdk-solidity/=lib/pyth-sdk-solidity']  
Writing and compiling the Smart Contract  
Once your project has been created and dependencies have been installed, you can now start writing a smart contract.  
The Solidity code below defines a smart contract named ExampleContract. The code uses the IPyth interface from the Pyth Solidity SDK.  
An instance ofIPyth is defined within the contract that provides functions for consuming Pyth price feeds. The constructor for the IPyth interface expects a contract address to be provided. This address provided in the code example below (0xA2aa501b19aff244D90cc15a4Cf739D2725B5729) corresponds to the Pyth contract address for the Base Sepolia testnet.  
INFO  
Pyth also supports other EVM networks, such as Base Mainnet. For a list of all network contract addresses, visit the Pyth documentation.  
The contract also contains a function named getLatestPrice. This function takes a provided priceUpdateData that is used to get updated price data, and returns the price given a priceId of a price feed. The smart contract provided below uses a priceId of 0xff61491a931112ddf1bd8147cd1b641375f79f5825126d665480874634fd0ace, which corresponds to the price feed for ETH / USD.  
INFO  
Pyth provides a number of price feeds. For a list of available price feeds, visit the Pyth documentation.  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
import "@pythnetwork/pyth-sdk-solidity/IPyth.sol";  
import "@pythnetwork/pyth-sdk-solidity/PythStructs.sol";  
  
contract ExampleContract {  
 IPyth pyth;  
  
 /\*\*  
 \* Network: Base Sepolia (testnet)  
 \* Address: 0xA2aa501b19aff244D90cc15a4Cf739D2725B5729  
 \*/  
 constructor() {  
 pyth = IPyth(0xA2aa501b19aff244D90cc15a4Cf739D2725B5729);  
 }  
  
 function getLatestPrice(  
 bytes[] calldata priceUpdateData  
 ) public payable returns (PythStructs.Price memory) {  
 // Update the prices to the latest available values and pay the required fee for it. The `priceUpdateData` data  
 // should be retrieved from our off-chain Price Service API using the `pyth-evm-js` package.  
 // See section "How Pyth Works on EVM Chains" below for more information.  
 uint fee = pyth.getUpdateFee(priceUpdateData);  
 pyth.updatePriceFeeds{ value: fee }(priceUpdateData);  
  
 bytes32 priceID = 0xff61491a931112ddf1bd8147cd1b641375f79f5825126d665480874634fd0ace;  
 // Read the current value of priceID, aborting the transaction if the price has not been updated recently.  
 // Every chain has a default recency threshold which can be retrieved by calling the getValidTimePeriod() function on the contract.  
 // Please see IPyth.sol for variants of this function that support configurable recency thresholds and other useful features.  
 return pyth.getPrice(priceID);  
 }  
}  
In your project, add the code provided above to a new file named src/ExampleContract.sol and delete the src/Counter.sol contract that was generated with the project (You can also delete the test/Counter.t.sol and script/Counter.s.sol files).  
To compile the new smart contract, run:  
forge build  
Deploying the smart contract  
Setting up your wallet as the deployer  
Before you can deploy your smart contract to the Base network, you will need to set up a wallet to be used as the deployer.  
To do so, you can use the cast wallet import command to import the private key of the wallet into Foundry's securely encrypted keystore:  
cast wallet import deployer --interactive  
After running the command above, you will be prompted to enter your private key, as well as a password for signing transactions.  
CAUTION  
For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation.  
It is critical that you do NOT commit this to a public repo.  
To confirm that the wallet was imported as the deployer account in your Foundry project, run:  
cast wallet list  
Setting up environment variables for Base Sepolia  
To setup your environment for deploying to the Base network, create an .env file in the home directory of your project, and add the RPC URL for the Base Sepolia testnet:  
BASE\_SEPOLIA\_RPC="https://sepolia.base.org"  
Once the .env file has been created, run the following command to load the environment variables in the current command line session:  
source .env  
Deploying the smart contract to Base Sepolia  
With your contract compiled and environment setup, you are ready to deploy the smart contract to the Base Sepolia Testnet!  
For deploying a single smart contract using Foundry, you can use the forge create command. The command requires you to specify the smart contract you want to deploy, an RPC URL of the network you want to deploy to, and the account you want to deploy with.  
To deploy the ExampleContract smart contract to the Base Sepolia test network, run the following command:  
forge create ./src/ExampleContract.sol:ExampleContract --rpc-url $BASE\_SEPOLIA\_RPC --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet's private key.  
INFO  
Your wallet must be funded with ETH on the Base Sepolia Testnet to cover the gas fees associated with the smart contract deployment. Otherwise, the deployment will fail.  
To get testnet ETH for Base Sepolia, see the prerequisites.  
After running the command above, the contract will be deployed on the Base Sepolia test network. You can view the deployment status and contract by using a block explorer.  
Interacting with the Smart Contract  
The getLatestPrice(bytes[]) function of the deployed contract takes a priceUpdateData argument that is used to get the latest price. This data can be fetched using the Hermes web service. Hermes allows users to easily query for recent price updates via a REST API. Make a curl request to fetch the priceUpdateData the priceId, 0xff61491a931112ddf1bd8147cd1b641375f79f5825126d665480874634fd0ace:  
curl https://hermes.pyth.network/api/latest\_vaas?ids[]=0xff61491a931112ddf1bd8147cd1b641375f79f5825126d665480874634fd0ace  
Once you have the priceUpdateData, you can use Foundry’s cast command-line tool to interact with the smart contract and call the getLatestPrice(bytes[]) function to fetch the latest price of ETH.  
To call the getLatestPrice(bytes[]) function of the smart contract, run the following command, replacing <DEPLOYED\_ADDRESS> with the address of your deployed contract, and <PRICE\_UPDATE\_DATA> with the priceUpdateData returned by the Hermes endpoint:  
cast call <DEPLOYED\_ADDRESS> --rpc-url $BASE\_SEPOLIA\_RPC "getLatestPrice(bytes[])" <PRICE\_UPDATE\_DATA>  
You should receive the latest ETH / USD price in hexadecimal form.  
Conclusion  
Congratulations! You have successfully deployed and interacted with a smart contract that consumes a Pyth Network oracle to access a real-time price feed on Base.  
To learn more about Oracles and using Pyth Network price feeds within your smart contracts on Base, check out the following resources:  
Oracles  
Pyth Network Price Feeds  
Tags:oracles  
ON THIS PAGE  
Prerequisites  
What is Pyth Network?  
Foundry  
Coinbase Wallet  
Wallet funds  
Creating a project  
Installing Pyth smart contracts  
Writing and compiling the Smart Contract  
Deploying the smart contract  
Interacting with the Smart Contract  
Setting up your wallet as the deployer  
Setting up environment variables for Base Sepolia  
Deploying the smart contract to Base Sepolia  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/oracles-supra-vrf  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Generating random numbers contracts using Supra dVRF  
This tutorial will guide you through the process of creating a smart contract on Base that utilizes Supra dVRF to serve random numbers using an onchain randomness generation mechanism directly within your smart contracts.  
Objectives  
By the end of this tutorial you should be able to do the following:  
Set up a smart contract project for Base using Foundry  
Install the Supra dVRF as a dependency  
Use Supra dVRF within your smart contract  
Deploy and test your smart contracts on Base  
Prerequisites  
Foundry  
This tutorial requires you to have Foundry installed.  
From the command-line (terminal), run: curl -L https://foundry.paradigm.xyz | bash  
Then run foundryup, to install the latest (nightly) build of Foundry  
For more information, see the Foundry Book installation guide.  
Coinbase Wallet  
In order to deploy a smart contract, you will first need a wallet. You can create a wallet by downloading the Coinbase Wallet browser extension.  
Download Coinbase Wallet  
Wallet funds  
Deploying contracts to the blockchain requires a gas fee. Therefore, you will need to fund your wallet with ETH to cover those gas fees.  
For this tutorial, you will be deploying a contract to the Base Sepolia test network. You can fund your wallet with Base Sepolia ETH using one of the faucets listed on the Base Network Faucets page.  
Supra wallet registration  
CAUTION  
Supra dVRF V2 requires subscription to the service with a customer controlled wallet address to act as the main reference.  
Therefore you must register your wallet with the Supra team if you plan to consume Supra dVRF V2 within your smart contracts.  
Please refer to the Supra documentation for the latest steps on how to register your wallet for their service.  
What is a Verifiable Random Function (VRF)?  
A Verifiable Random Function (VRF) provides a solution for generating random outcomes in a manner that is both decentralized and verifiably recorded onchain. VRFs are crucial for applications where randomness integrity is paramount, such as in gaming or prize drawings.  
Supra dVRF provides a decentralized VRF that ensures that the outcomes are not only effectively random but also responsive, scalable, and easily verifiable, thereby addressing the unique needs of onchain applications for trustworthy and transparent randomness.  
Creating a project  
Before you can begin writing smart contracts for Base, you need to set up your development environment by creating a Foundry project.  
To create a new Foundry project, first create a new directory:  
mkdir myproject  
Then run:  
cd myproject  
forge init  
This will create a Foundry project, which has the following basic layout:  
.  
├── foundry.toml  
├── script  
 │ └── Counter.s.sol  
├── src  
 │ └── Counter.sol  
└── test  
 └── Counter.t.sol  
Writing the Smart Contract  
Once your Foundry project has been created, you can now start writing a smart contract.  
The Solidity code below defines a basic contract named RNGContract. The smart contract's constructor takes in a single address and assigns it to a member variable named supraAddr. This address corresponds to the contract address of the Supra Router Contract that will be used to generate random numbers. The contract address of the Supra Router Contract on Base Sepolia testnet is 0x99a021029EBC90020B193e111Ae2726264a111A2.  
The contract also assigns the contract deployer (msg.sender) to a member variable named supraClientAddress. This should be the client wallet address that is registered and whitelisted to use Supra VRF (see: Prerequisites).  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
contract RNGContract {  
 address supraAddr;  
 address supraClientAddress;  
  
 constructor(address supraSC) {  
 supraAddr = supraSC;  
 supraClientAddress = msg.sender;  
 }  
}  
In your project, add the code provided above to a new file named src/ExampleContract.sol, and delete the src/Counter.sol contract that was generated with the project. (you can also delete the test/Counter.t.sol and script/Counter.s.sol files).  
The following sections will guide you step-by-step on how to update your contract to generate random numbers using the Supra Router contract.  
Adding the Supra Router Contract interface  
In order to help your contract (the requester contract) interact with the Supra Router contract and understand what methods it can call, you will need to add the following interface to your contract file.  
interface ISupraRouter {  
 function generateRequest(string memory \_functionSig, uint8 \_rngCount, uint256 \_numConfirmations, uint256 \_clientSeed, address \_clientWalletAddress) external returns(uint256);  
 function generateRequest(string memory \_functionSig, uint8 \_rngCount, uint256 \_numConfirmations, address \_clientWalletAddress) external returns(uint256);  
}  
The ISupraRouter interface defines a generateRequest function. This function is used to create a request for random numbers. The generateRequest function is defined twice, because one of the definitions allows for an optional \_clientSeed (defaults to 0) for additional unpredictability.  
INFO  
Alternatively, you can add the ISupraRouter interface in a separate interface file and inherit the interface in your contract.  
Adding a request function  
Once you have defined the ISupraRouter, you are ready to add the logic to your smart contract for requesting random numbers.  
For Supra dVRF, adding logic for requesting random numbers requires two functions:  
A request function  
A callback function  
The request function is a custom function defined by the developer. There are no requirements when it comes to the signature of the request function.  
The following code is an example of a request function named rng that requests random numbers using the Supra Router Contract. Add this function to your smart contract:  
function rng() external returns (uint256) {  
 // Amount of random numbers to request  
 uint8 rngCount = 5;  
 // Amount of confirmations before the request is considered complete/final  
 uint256 numConfirmations = 1;  
 uint256 nonce = ISupraRouter(supraAddr).generateRequest(  
 "requestCallback(uint256,uint256[])",  
 rngCount,  
 numConfirmations,  
 supraClientAddress  
 );  
 return nonce;  
 // store nonce if necessary (e.g., in a hashmap)  
 // this can be used to track parameters related to the request in a lookup table  
 // these can be accessed inside the callback since the response from supra will include the nonce  
}  
The rng function above requests 5 random numbers (defined by rngCount), and waits 1 confirmation (defined by numConfirmations) before considering the result to be final. It makes this request by calling the generateRequest function of the Supra Router contract, while providing a callback function with the signature requestCallback(uint256,uint256[]).  
Adding a callback function  
As seen in the previous section, the generateRequest method of the Supra Router contract expects a signature for a callback function. This callback function must be of the form: uint256 nonce, uint256[] calldata rngList, and must include validation code, such that only the Supra Router contract can call the function.  
To do this, add the following callback function (requestCallback) to your smart contract:  
function requestCallback(uint256 \_nonce ,uint256[] \_rngList) external {  
 require(msg.sender == supraAddr, "Only the Supra Router can call this function.");  
 uint8 i = 0;  
 uint256[] memory x = new uint256[](rngList.length);  
 rngForNonce[nonce] = x;  
 for(i=0; i<rngList.length;i++){  
 rngForNonce[nonce][i] = rngList[i] % 100;  
 }  
}  
Once a random number is generated, requestCallback is executed by the Supra Router. The code above stores the resulting random numbers list in a map named rngForNonce using the \_nonce argument. Because of this, you will need to add the following mapping to your contract:  
mapping (uint256 => uint256[] ) rngForNonce;  
Adding a function to view the result  
To fetch resulting random numbers based on their associated nonce, you add a third function:  
function viewRngForNonce(uint256 nonce) external view returns (uint256[] memory) {  
 return rngForNonce[nonce];  
}  
Final smart contract code  
After following all the steps above, your smart contract code should look like the following:  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
interface ISupraRouter {  
 function generateRequest(string memory \_functionSig, uint8 \_rngCount, uint256 \_numConfirmations, uint256 \_clientSeed, address \_clientWalletAddress) external returns (uint256);  
 function generateRequest(string memory \_functionSig, uint8 \_rngCount, uint256 \_numConfirmations, address \_clientWalletAddress) external returns (uint256);  
}  
  
contract RNGContract {  
 address supraAddr;  
 address supraClientAddress;  
  
 mapping (uint256 => uint256[]) rngForNonce;  
  
 constructor(address supraSC) {  
 supraAddr = supraSC;  
 supraClientAddress = msg.sender;  
 }  
  
 function rng() external returns (uint256) {  
 // Amount of random numbers to request  
 uint8 rngCount = 5;  
 // Amount of confirmations before the request is considered complete/final  
 uint256 numConfirmations = 1;  
 uint256 nonce = ISupraRouter(supraAddr).generateRequest(  
 "requestCallback(uint256,uint256[])",  
 rngCount,  
 numConfirmations,  
 supraClientAddress  
 );  
 return nonce;  
 }  
  
 function requestCallback(uint256 \_nonce, uint256[] memory \_rngList) external {  
 require(msg.sender == supraAddr, "Only the Supra Router can call this function.");  
 uint8 i = 0;  
 uint256[] memory x = new uint256[](\_rngList.length);  
 rngForNonce[\_nonce] = x;  
 for (i = 0; i < \_rngList.length; i++) {  
 rngForNonce[\_nonce][i] = \_rngList[i] % 100;  
 }  
 }  
  
 function viewRngForNonce(uint256 nonce) external view returns (uint256[] memory) {  
 return rngForNonce[nonce];  
 }  
}  
CAUTION  
You must whitelist this smart contract under the wallet address you registered with Supra, and deposit funds to be paid for the gas fees associated with transactions for your callback function.  
Follow the guidance steps provided by Supra for whitelisting your contract and depositing funds.  
If you have not yet registered your wallet with Supra, see the Prerequisites section.  
Compiling the Smart Contract  
To compile your smart contract code, run:  
forge build  
Deploying the smart contract  
Setting up your wallet as the deployer  
Before you can deploy your smart contract to the Base network, you will need to set up a wallet to be used as the deployer.  
To do so, you can use the cast wallet import command to import the private key of the wallet into Foundry's securely encrypted keystore:  
cast wallet import deployer --interactive  
After running the command above, you will be prompted to enter your private key, as well as a password for signing transactions.  
CAUTION  
For instructions on how to get your private key from Coinbase Wallet, visit the Coinbase Wallet documentation.  
It is critical that you do NOT commit this to a public repo.  
To confirm that the wallet was imported as the deployer account in your Foundry project, run:  
cast wallet list  
Setting up environment variables for Base Sepolia  
To setup your environment for deploying to the Base network, create an .env file in the home directory of your project, and add the RPC URL for the Base Sepolia testnet, as well as the Supra Router contract address for Base Sepolia testnet:  
BASE\_SEPOLIA\_RPC="https://sepolia.base.org"  
ISUPRA\_ROUTER\_ADDRESS=0x99a021029EBC90020B193e111Ae2726264a111A2  
Once the .env file has been created, run the following command to load the environment variables in the current command line session:  
source .env  
Deploying the smart contract to Base Sepolia  
With your contract compiled and environment setup, you are ready to deploy the smart contract to the Base Sepolia Testnet!  
For deploying a single smart contract using Foundry, you can use the forge create command. The command requires you to specify the smart contract you want to deploy, an RPC URL of the network you want to deploy to, and the account you want to deploy with.  
To deploy the RNGContract smart contract to the Base Sepolia test network, run the following command:  
forge create ./src/RNGContract.sol:RNGContract --rpc-url $BASE\_SEPOLIA\_RPC --constructor-args $ISUPRA\_ROUTER\_ADDRESS --account deployer  
When prompted, enter the password that you set earlier, when you imported your wallet’s private key.  
INFO  
Your wallet must be funded with ETH on the Base Sepolia Testnet to cover the gas fees associated with the smart contract deployment. Otherwise, the deployment will fail.  
To get testnet ETH for Base Sepolia, see the prerequisites.  
After running the command above, the contract will be deployed on the Base Sepolia test network. You can view the deployment status and contract by using a block explorer.  
Interacting with the Smart Contract  
Foundry provides the cast command-line tool that can be used to interact with the smart contract that was deployed and call the getLatestPrice() function to fetch the latest price of ETH.  
To call the getLatestPrice() function of the smart contract, run:  
cast call <DEPLOYED\_ADDRESS> --rpc-url $BASE\_SEPOLIA\_RPC "rng()"  
You should receive a nonce value.  
You can use this nonce value to call the viewRngForNonce(uint256) function to get the resulting list of randomly generated numbers:  
cast call <DEPLOYED\_ADDRESS> --rpc-url $BASE\_SEPOLIA\_RPC "viewRngForNonce(uint256)" <NONCE>  
Conclusion  
Congratulations! You have successfully deployed and interacted with a smart contract that generates a list of random numbers using Supra dVRF on the Base blockchain network.  
To learn more about VRF and using Supra dVRF to generate random numbers within your smart contracts on Base, check out the following resources:  
Oracles  
Supra dVRF - Developer Guide V2  
Tags:oraclesvrf  
ON THIS PAGE  
Prerequisites  
What is a Verifiable Random Function (VRF)?  
Foundry  
Coinbase Wallet  
Wallet funds  
Supra wallet registration  
Creating a project  
Writing the Smart Contract  
Compiling the Smart Contract  
Adding the Supra Router Contract interface  
Adding a request function  
Adding a callback function  
Adding a function to view the result  
Final smart contract code  
Deploying the smart contract  
Interacting with the Smart Contract  
Setting up your wallet as the deployer  
Setting up environment variables for Base Sepolia  
Deploying the smart contract to Base Sepolia  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/run-a-base-node  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Running a Base Node  
This tutorial will walk you through setting up your own Base Node.  
Objectives  
By the end of this tutorial you should be able to:  
Deploy and sync a Base node  
Prerequisites  
CAUTION  
Running a node is time consuming, resource expensive, and potentially costly. If you don't already know why you want to run your own node, you probably don't need to.  
  
If you're just getting started and need an RPC URL, you can use our free endpoints:  
  
Mainnet: https://mainnet.base.org  
Testnet (Sepolia): https://sepolia.base.org  
Note: Our RPCs are rate-limited, they are not suitable for production apps.  
If you're looking to harden your app and avoid rate-limiting for your users, please check out one of our partners.  
Hardware requirements  
We recommend you have this configuration to run a node:  
8-Core CPU  
at least 16 GB RAM  
a locally attached NVMe SSD drive  
adequate storage capacity to accommodate both the snapshot restoration process (if restoring from snapshot) and chain data, ensuring a minimum of (2 \* current\_chain\_size) + snapshot\_size + 20%\_buffer  
INFO  
If utilizing Amazon Elastic Block Store (EBS), ensure timing buffered disk reads are fast enough in order to avoid latency issues alongside the rate of new blocks added to Base during the initial synchronization process; io2 block express is recommended.  
Docker  
This tutorial assumes you are familiar with Docker and have it running on your machine.  
L1 RPC URL  
You'll need your own L1 RPC URL. This can be one that you run yourself, or via a third-party provider, such as our partners.  
Running a Node  
Clone the repo.  
Ensure you have an Ethereum L1 full node RPC available (not Base), and set OP\_NODE\_L1\_ETH\_RPC & OP\_NODE\_L1\_BEACON (in the .env.\* file if using docker-compose). If running your own L1 node, it needs to be synced before Base will be able to fully sync.  
Uncomment the line relevant to your network (.env.sepolia, or .env.mainnet) under the 2 env\_file keys in docker-compose.yml.  
Run docker compose up. Confirm you get a response from:  
curl -d '{"id":0,"jsonrpc":"2.0","method":"eth\_getBlockByNumber","params":["latest",false]}' \  
 -H "Content-Type: application/json" http://localhost:8545  
CAUTION  
Syncing your node may take days and will consume a vast amount of your requests quota. Be sure to monitor usage and up your plan if needed.  
Snapshots  
If you're a prospective or current Base Node operator and would like to restore from a snapshot to save time on the initial sync, it's possible to always get the latest available snapshot of the Base chain on mainnet and/or testnet by using the following CLI commands. The snapshots are updated every week.  
Restoring from snapshot  
In the home directory of your Base Node, create a folder named geth-data or reth-data. If you already have this folder, remove it to clear the existing state and then recreate it. Next, run the following code and wait for the operation to complete.  
Network Client Snapshot Type Command  
Testnet Geth Full wget https://sepolia-full-snapshots.base.org/$(curl https://sepolia-full-snapshots.base.org/latest)  
Testnet Geth Archive wget https://sepolia-archive-snapshots.base.org/$(curl https://sepolia-archive-snapshots.base.org/latest)  
Testnet Reth Archive wget https://sepolia-reth-archive-snapshots.base.org/$(curl https://sepolia-reth-archive-snapshots.base.org/latest)  
Mainnet Geth Full wget https://mainnet-full-snapshots.base.org/$(curl https://mainnet-full-snapshots.base.org/latest)  
Mainnet Geth Archive wget https://mainnet-archive-snapshots.base.org/$(curl https://mainnet-archive-snapshots.base.org/latest)  
Mainnet Reth Archive wget https://mainnet-reth-archive-snapshots.base.org/$(curl https://mainnet-reth-archive-snapshots.base.org/latest)  
You'll then need to untar the downloaded snapshot and place the geth subfolder inside of it in the geth-data folder you created (unless you changed the location of your data directory).  
Return to the root of your Base node folder and start your node.  
cd ..  
docker compose up --build  
Your node should begin syncing from the last block in the snapshot.  
Check the latest block to make sure you're syncing from the snapshot and that it restored correctly. If so, you can remove the snapshot archive that you downloaded.  
Syncing  
You can monitor the progress of your sync with:  
echo Latest synced block behind by: $((($(date +%s)-$( \  
 curl -d '{"id":0,"jsonrpc":"2.0","method":"optimism\_syncStatus"}' \  
 -H "Content-Type: application/json" http://localhost:7545 | \  
 jq -r .result.unsafe\_l2.timestamp))/60)) minutes  
You'll also know that the sync hasn't completed if you get Error: nonce has already been used if you try to deploy using your node.  
Tags:nodes  
ON THIS PAGE  
Prerequisites  
Running a Node  
Snapshots  
Syncing  
L1 RPC URL  
Running a Node  
Snapshots  
Syncing  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/shopify-storefront-commerce  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Deploy a Shopify Storefront with Coinbase Commerce  
Deploy a Shopify Storefront with Coinbase Commerce Plugin Shopify provides a robust platform for creating online stores.  
Shopify Storefronts allow you to create unique experiences for your online store.  
This tutorial will guide you through setting up a Shopify storefront that integrates with Coinbase Commerce, allowing you to accept cryptocurrency payments.  
Learn more about Hydrogen and Oxygen.  
Objectives  
By the end of this tutorial will be able to:  
Create a Shopify storefront  
Link your Coinbase Commerce account to Shopify  
Link Shopify credentials to your storefront  
Deploy a test site using Hydrogen/Oxygen  
Prerequisites  
Coinbase Commerce  
Coinbase Commerce is a platform that enables merchants to accept cryptocurrency payments in a decentralized manner. It provides tools for integrating crypto payments into online stores, offering a secure and straightforward way to receive hundreds of tokens across Base, Polygon, and Ethereum.  
To proceed, you will need a Coinbase Commerce Managed account. You can sign up for a business account here.  
Shopify  
You will need a Shopify Basic plan for this demo, creating a custom storefront, and using Hydrogen/Oxygen  
Hydrogen  
Hydrogen is Shopify's React-based framework for developing custom storefronts, offering components, utilities, and design patterns to simplify working with Shopify APIs. These projects are Remix apps preconfigured with Shopify-specific features.  
Oxygen  
Oxygen is Shopify’s global serverless hosting platform for deploying Hydrogen storefronts at the edge, managing deployment environments, environment variables, caching, and integrating with Shopify’s CDN.  
Getting Started  
To comply with Shopify's requirements, the Coinbase Commerce account linked to Shopify must be a Coinbase Managed account.  
To create one, sign up for a business account and select Coinbase Commerce as the account type. See this Coinbase Commerce Merchant help article for additional information on how to sign your business up.  
Link Coinbase Commerce to your Shopify store  
Once, you've created a Coinbase Commerce Managed account you can now add the plugin to your Shopify store.  
To add Coinbase Commerce as a payment method on your Shopify store, start by navigating to your admin page at https://admin.shopify.com/store/<YOUR-STORE-NAME>. Once there, click on Settings , located in the bottom left panel of the screen (or at https://admin.shopify.com/store/<YOUR-STORE-NAME>settings/general). Then select the Payments tab. Proceed by clicking Add a payment method and choose Search by provider. In the search field, type in Coinbase and select Coinbase Commerce from the results. Click Install and you should be redirected to Coinbase Commerce with a prompt log into your Coinbase account. Finally, click Activate to enable the Coinbase Commerce plugin.  
Create a Storefront  
This tutorial will guide you through the steps to create a new Hydrogen storefront for your Shopify store. This will allow you to showcase the products you already have in your Shopify account. Make sure you've already created your products by following the Shopify products guide.  
[SHOPIFY BASIC PLAN REQUIRED]  
To access the Hydrogen and Oxygen APIs, Shopify requires users to have at least a Basic Plan. The following steps will not work without this plan configured.  
To get started, clone a Shopify demo store using the Hydrogen framework. This will give us a quick setup to work with.  
npm create @shopify/hydrogen@latest -- --quickstart  
Once the demo store is cloned, navigate into the project directory and start the development server:  
cd hydrogen-quickstart  
npx shopify hydrogen dev  
Next, open a new terminal. You'll need to link your Hydrogen project to your Shopify store. This step connects the demo storefront with your Shopify account, allowing you to display your products.  
npx shopify hydrogen link  
[INSTALL HYDROGEN SALES CHANNEL]  
You will need to create access tokens for your own Shopify store. This is done by installing the Hydrogen sales channel, which includes built-in support for Oxygen, Shopify’s global edge hosting platform.  
The Shopify quickstart comes with a Mock Shop as a template. To ensure your storefront is configured with your products, update the project environment variables.  
The following code pulls the necessary settings from your Shopify account into the Hydrogen project:  
npx shopify hydrogen env pull  
[NOT SEEING YOUR PRODUCTS?]  
Your Shopify store should have products.  
Visit Shopify products guide for more details.  
Now, verify that everything is set up correctly and your site is running. Start the development server again:  
npx shopify hydrogen dev  
Finally, deploy your Hydrogen storefront to Oxygen, Shopify's hosting platform for Hydrogen apps. Run the following script:  
npx shopify hydrogen deploy  
Select Preview as the deployment type. After deployment, your terminal should display a URL where you can view your live site, such as:  
Your terminal will display a url like: https://hydrogen-quickstart-20c3648d482c7a17d77d.o2.myshopify.dev/  
You've successfully created and deployed your Hydrogen storefront.  
Accepting payments  
Visit your new storefront and add an item to your cart. Proceed to view your cart and then proceed to checkout. On the payment screen, you should see Coinbase Commerce automatically appear as an additional payment method alongside your existing payment options.  
[NOT SEEING A CRYPTO PAYMENT OPTION?]  
Remember to link "activate" your Coinbase Commerce plugin.  
Conclusion  
You now have a custom store front with the products from your Shopify account.  
In this tutorial, you learned how to integrate Coinbase as a payment provider for your Shopify store. You've also learned how to use Hydrogen and Oxygen to create and deploy a version of your store separate from Shopify.  
Tags:ecommercecryptoshopifyonchain summerocs  
ON THIS PAGE  
Prerequisites  
Getting Started  
Link Coinbase Commerce to your Shopify store  
Create a Storefront  
Accepting payments  
Link Coinbase Commerce to your Shopify store  
Create a Storefront  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/signature-mint-nft  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Signature Mint NFT  
A signature mint is a term for an NFT mint in which the recipient of the NFT pays for their own gas to receive the NFT, but may only do so if they possess a correct message signed by the owner or authorized address of the mint contract. Reasons for doing this include allowing fiat payment of minting fees, allowing holders of an NFT on one chain to mint that NFT on an unrelated chain, or gating the mint to users who meet other specific offchain requirements.  
Signature mints are not particularly complex, but they remain challenging to implement. Because they make use of both hashing and cryptography, there are no partially-correct states - either everything is exactly right and the mint works, or something is wrong somewhere and it doesn't.  
Combined with the rapid changes in Solidity, library contracts, and frontend libraries, troubleshooting errors is particularly difficult.  
Objectives  
By the end of this tutorial you should be able to:  
Cryptographically sign a message with a wallet  
Validate a signed message in a smart contract  
Implement a signature ERC-721 mint  
Prerequisites  
ERC-721 Tokens  
This tutorial assumes that you are able to write, test, and deploy your own ERC-721 tokens using the Solidity programming language. If you need to learn that first, check out our content in Base Learn or the sections specific to ERC-721 Tokens!  
Vercel  
You'll need to be comfortable deploying your app to Vercel, or using another solution on your own. Check out our tutorial on deploying with Vercel if you need a refresher!  
Building the Contract  
Start by setting up an OpenZeppelin ERC-721 contract. Set up variables and use the constructor to assign:  
A name for the collection  
Symbol for the collection  
Description  
IPFS Hash for the NFT art (assuming the same art for each NFT)  
A counter to keep track of which NFT id is next  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity 0.8.24;  
  
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";  
import "@openzeppelin/contracts/access/Ownable.sol";  
  
contract SoulboundSignatureMint is ERC721, Ownable {  
 string public nameString;  
 string public description;  
 string public tokenArtIPFSId;  
 uint public counter;  
  
 constructor(  
 string memory \_nameString,  
 string memory \_symbol,  
 string memory \_tokenArtIPFSId,  
 string memory \_description  
 ) ERC721(\_nameString, \_symbol) Ownable(msg.sender) {  
 nameString = \_nameString;  
 description = \_description;  
 tokenArtIPFSId = \_tokenArtIPFSId;  
 }  
}  
You're also using Ownable to assign an owner to this contract. You could instead just save an address for the authorized signer if you aren't going to add any functionality only the owner can invoke.  
Public Mint Function  
For the public-facing mint function, create a function called mintTo that accepts an address for the \_recipient.  
INFO  
A common pattern used to be to simply mint the token and give it to msg.sender. This practice is falling out of favor. Allowing the recipient to be different than the sender gives greater flexibility. Doing so is also necessary to assign the right NFT owner in the event the user is using a smart contract wallet, paymaster, or other form of account abstraction.  
function mintTo(address \_recipient, bytes memory \_signature) public {  
 counter++;  
 \_safeMint(msg.sender, counter);  
}  
Onchain Metadata  
Rather than pointing to a json file on the traditional internet, you can put your metadata directly in the contract. To do so, first import some helper libraries:  
import "@openzeppelin/contracts/utils/Base64.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
Next, override the functions for \_baseURI and tokenURI to return base 64 encoded json metadata with the information you supplied in the constructor:  
function \_baseURI() internal pure override returns (string memory) {  
 return "data:application/json;base64,";  
}  
  
function tokenURI(uint \_tokenId) public view override returns (string memory) {  
 if(\_tokenId > counter) {  
 revert InvalidTokenId(\_tokenId);  
 }  
  
 string memory json = Base64.encode(  
 bytes(  
 string(  
 abi.encodePacked(  
 '{"name": "',  
 nameString,  
 ' #: ',  
 Strings.toString(\_tokenId),  
 '","description": "',  
 description,  
 '", "image": "ipfs://',  
 tokenArtIPFSId,  
 '"}'  
 )  
 )  
 )  
 );  
  
 return string(abi.encodePacked(\_baseURI(), json));  
}  
Be very careful setting up the single and double quotes above and be sure to test this function to make sure the result is valid json metadata. An error here will break the NFT and it won't show up correctly in wallets or marketplaces!  
Preventing Transfers  
Soulbound is a video-game term that means that an item is permanently attached to the receiver - it can not be transferred. It's up to you if this restriction fits your design goals. We use it often because our NFTs are intended to be fun mementos or markers of personal accomplishment and not something that will ever have monetary value. Preventing trading reduces speculation and farming on something we did for fun!  
To prevent transfers other than the initial mint, you can override the \_update function.  
INFO  
Previously, this was done with the \_beforeTransfer function. Current versions of OpenZeppelin's ERC-721 implementation have replaced that function with \_update.  
/\*\*  
 \* Disallow transfers (Soulbound NFT)  
 \*/  
/\*\*  
 \* @dev Internal function to handle token transfers.  
 \* Restricts the transfer of Soulbound tokens.  
 \*/  
function \_update(address to, uint256 tokenId, address auth)  
 internal  
 override(ERC721)  
 returns (address)  
{  
 address from = \_ownerOf(tokenId);  
 if (from != address(0) && to != address(0)) {  
 revert SoulboundToken();  
 }  
  
 return super.\_update(to, tokenId, auth);  
}  
Deploy and Test  
Before getting into the complexity of validating a cryptographic signature, it's a good idea to validate your contract and make sure it is working as expected. You'll need to pin an image to IPFS and get a hash for it to use in your metadata. You can use a service like Pinata to help with that.  
Adding Signature Validation  
To validate the signature that you'll later create in your backend, you'll use a pair of cryptography utilities from OpenZeppelin:  
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";  
import "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";  
using ECDSA for bytes32;  
using MessageHashUtils for bytes32;  
These utilities abstract away most of the complexity involved in working with messages adhering to the ERC-191 and EIP-712 specifications. Importantly, they work with the message format that prefixes "\x19Ethereum Signed Message:\n32" to the message. You must also do this when creating the signed message!  
Add a function to validateSignature:  
function validateSignature(address \_recipient, bytes memory \_signature) public view returns (bool) {  
 bytes32 messageHash = keccak256(abi.encodePacked(\_recipient));  
 bytes32 ethSignedMessageHash = messageHash.toEthSignedMessageHash();  
 address signer = ethSignedMessageHash.recover(\_signature);  
  
 return signer == owner();  
}  
The way the verification works is a little obtuse, particularly given that you haven't created the \_signature yet. The function has two inputs:  
The message or variables in the signed message  
Here, this is the address of the \_recipient  
The \_signature, or signed message provided by the user claiming they have been given permission to mint the NFT  
First, the function recreates the hash of the data to be signed. If you were including other variables, you'd include them here as well. Next, messageHash.toEthSignedMessageHash prepends the bytes representation of "\x19Ethereum Signed Message:\n32" to the message, then hashes the result.  
Finally, calling recover with ethSignedMessageHash and the \_signature attempts to recover the signing address from the \_signature using the independently constructed message data.  
If the recovered address matches the expected address, in this case, the contract owner, then the provided \_signature is valid. If the addresses do not match, then the \_signature is not valid.  
Update your mintTo function to make use of the validation:  
function mintTo(address \_recipient, bytes memory \_signature) public {  
 if(!validateSignature(\_recipient, \_signature)) {  
 revert InvalidSignature();  
 }  
  
 counter++;  
 \_safeMint(msg.sender, counter);  
}  
DANGER  
Nothing in the above validation method prevents a user, or a third party, from obtaining a valid, signed message from a previous transaction and reusing it for a new transaction. In this case, it doesn't matter because signature re-use would only allow minting more soulbound NFTs for the address within the signature.  
Other design requirements should use a nonce as a part of the signed data to prevent signature reuse.  
Signing the Message  
If you're using Hardhat with viem, you can write tests to verify the signing and validation mechanisms are working. Otherwise, there isn't a point, as success is dependent on the exact and specific way and order signing happens. If you're using a different toolkit to write your smart contracts, continue in your backend directly.  
If you're using a different library, you'll need to do research to figure out how to exactly reproduce the steps below.  
Setting Up the Test  
Add a new test file and fill out a skeleton to deploy your contract and run a test:  
import { loadFixture } from '@nomicfoundation/hardhat-toolbox-viem/network-helpers';  
import { expect } from 'chai';  
import hre from 'hardhat';  
  
describe('Test', function () {  
 // We define a fixture to reuse the same setup in every test.  
 // We use loadFixture to run this setup once, snapshot that state,  
 // and reset Hardhat Network to that snapshot in every test.  
 async function deploySignatureFixture() {  
 // Contracts are deployed using the first signer/account by default  
 const [owner, signer0, signer1] = await hre.viem.getWalletClients();  
  
 const soulboundSignatureMint = await hre.viem.deployContract('SoulboundSignatureMint', [  
 'Cool NFT Name', // Name  
 'CNFT', // Symbol  
 'QmRsQCyTEALYnHvBupFcs2ofzeeswEEEGN...', // IPFS Hash  
 'This is a cool NFT!', // Description  
 ]);  
  
 const publicClient = await hre.viem.getPublicClient();  
  
 return {  
 soulboundSignatureMint,  
 owner,  
 signer1,  
 publicClient,  
 };  
 }  
  
 describe('Mint', function () {  
 it('Should validate the signed message', async function () {  
 const { soulboundSignatureMint, owner, signer0, signer1 } = await loadFixture(  
 deploySignatureFixture,  
 );  
  
 const ownerAddress = await owner.account.address;  
 const signer1Address = await signer1.account.address;  
  
 // TODO...  
  
 // Signer 1 calls the mintTo function with the signature  
 expect(await soulboundSignatureMint.write.mintTo([signer1Address, signature])).to.be.ok;  
 });  
 });  
});  
You can use the example in the documentation for signMessage in the viem wallet client to get started, but it will not work as expected.  
// BAD CODE EXAMPLE DO NOT USE!  
const signature = await owner.signMessage({  
 message: signer1Address,  
});  
Try it, and it will fail. Add a log to your contract and you'll see that the recovered signer address is random, rather than the first address in the list of default Hardhat accounts.  
The reason for this is that while signMessage does follow the previously mentioned standards, prepends "\x19Ethereum Signed Message:\n32" to the message, and correctly signs it, it does not prepare the data to be signed in exactly the same way as the smart contract converts the address to bytes32.  
To fix this, first import some helper functions from viem:  
import { keccak256, encodePacked, toBytes } from 'viem';  
Then encodePacked and keccak256 hash your variables and turn them into bytes, just like you did in the contract in validateSignature:  
const message = keccak256(encodePacked(['address'], [signer1Address]));  
const messageBytes = toBytes(message);  
Finally, call the wallet signMessage function with the newly assembled messageBytes. You'll need to mark the data representation as raw:  
const signature = await owner.signMessage({  
 message: { raw: messageBytes },  
});  
Test again, and it will pass!  
Signing from the Backend  
It's up to you to determine the conditions that you're willing to sign a message. Once those conditions are met, you can use a similar process to load a wallet from your private key and sign the message on any TypeScript backend:  
const authorizedAccount = privateKeyToAccount(COINBASE\_WALLET\_KEY as `0x${string}`);  
  
const authorizedClient = createWalletClient({  
 account: authorizedAccount,  
 chain: base,  
 transport: http(), // Leave empty for local account  
});  
  
// Align signed message with OpenZeppelin/Solidity  
  
const messageToSign = keccak256(encodePacked(['address'], [userAddress as `0x${string}`]));  
const messageBytes = getBytes(messageToSign);  
  
// Create an Ethereum Signed Message with the user's address  
const signedMessage = await authorizedClient.signMessage({  
 message: { raw: messageBytes },  
});  
console.log('User address:', userAddress);  
console.log('Signed message:', signedMessage);  
  
const data = encodeFunctionData({  
 abi: mintContractData.abi,  
 functionName: 'mintTo',  
 args: [userAddress, signedMessage],  
});  
INFO  
privateKeyToAccount expects that your key starts with 0x. You may need to manually add that depending on the tool you exported it from.  
Conclusion  
In this tutorial, you've learned how to create a signature mint, which allows you to set conditions on a backend before a user is allowed to mint your NFT. You've learned the detailed and specific steps needed to align viem's method of signing messages with OpenZeppelin's method of verifying them. Finally, you've learned a few of the risks and considerations needed in designing this type of mint.  
Tags:nft  
ON THIS PAGE  
Prerequisites  
Building the Contract  
Public Mint Function  
Onchain Metadata  
Signing the Message  
Setting Up the Test  
Onchain Metadata  
Preventing Transfers  
Deploy and Test  
Adding Signature Validation  
Signing from the Backend  
Setting Up the Test  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/simple-onchain-nfts  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Simple Onchain NFTs  
Many NFTs are dependent on offchain metadata and images. Some use immutable storage locations, such as IPFS. Others use traditional web locations, and many of these allow the owner of the contract to modify the URL returned by a contract when a site or user attempts to retrieve the location of the token art and metadata. This power isn't inherently bad, because we probably want someone to be able to fix the contract if the storage location goes down. However, it does introduce a requirement to trust the contract owner.  
In this tutorial, we'll show you how to do this to create a simple NFT that is fully onchain. This contract is used in our tutorials for Thirdweb and Unreal - NFT Items and the Coinbase Smart Wallet.  
The result of this tutorial is used in other tutorials. Below, you can find the complete contract and ABI. Feel free to use it if you're working on one of those and don't want to get sidetracked.  
Objectives  
By the end of this tutorial you should be able to:  
Programmatically generate and return json metadata for ERC-721 tokens  
Deterministically construct unique SVG art in a smart contract  
Generate deterministic, pseudorandom numbers  
Prerequisites  
ERC-721 Tokens  
This tutorial assumes that you are able to write, test, and deploy your own ERC-721 tokens using the Solidity programming language. If you need to learn that first, check out our content in Base Camp or the sections specific to ERC-721 Tokens!  
Vector Art  
You'll need some familiarity with the SVG art format and a basic level of ability to edit and manipulate vector art. If you don't have this, find an artist friend and collaborate!  
Building the Contract  
Start by setting up an [OpenZeppelin ERC-721] contract. You'll need to set up a mintTo function that accepts the address that should receive the NFT.  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.24;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";  
  
contract RandomColorNFT is ERC721 {  
 uint public counter;  
  
 constructor() ERC721("RandomColorNFT", "RCNFT") {  
 }  
  
 function mintTo(address \_to) public {  
 counter++;  
 \_safeMint(\_to, counter);  
 }  
}  
INFO  
With the Smart Wallet, msg.sender is the users custodial address - where you want to send the NFT. This is not always the case with account abstraction. In some other implementations, msg.sender is the smart contract address, even if the user signs in with an EOA. Regardless, it's becoming a common practice to pass the address you want the NFT to go to explicitly.  
Onchain Metadata  
Rather than pointing to a json file on the traditional internet, you can put your metadata directly in the contract. To do so, first import some helper libraries:  
import "@openzeppelin/contracts/utils/Base64.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
Next, override the functions for \_baseURI and tokenURI to return base 64 encoded json metadata with the appropriate information:  
function \_baseURI() internal pure override returns (string memory) {  
 return "data:application/json;base64,";  
}  
  
function tokenURI(uint \_tokenId) public view override returns (string memory) {  
 if(\_tokenId > counter) {  
 revert InvalidTokenId(\_tokenId);  
 }  
  
 string memory json = Base64.encode(  
 bytes(  
 string(  
 abi.encodePacked(  
 '{"name": "',  
 name(),  
 ' #: ',  
 Strings.toString(\_tokenId),  
 '","description": "Random colors are pretty or boring!", "image": "image": "data:image/svg+xml;base64,',  
 // TODO...,  
 '"}'  
 )  
 )  
 )  
 );  
  
 return string(abi.encodePacked(\_baseURI(), json));  
}  
Be very careful setting up the single and double quotes above and be sure to test this function to make sure the result is valid json metadata. An error here will break the NFT and it won't show up correctly in wallets or marketplaces!  
Onchain SVG Image  
For this NFT, the art will consist of a simple onchain SVG containing a square with a pseudo-randomly chosen color. Check out our tutorial on [Building Onchain NFTs] if you want to try something more complicated.  
Start by scaffolding out a render function:  
function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<svg xmlns='http://www.w3.org/2000/svg' viewBox='0 0 1024 1024'>",  
 // TODO: add a rectangle with a random color fill  
 "</svg>"  
 )  
 );  
}  
Rectangles in SVG images are created with the [rect] element. To cover the whole background, you can set the width and height to the size of the viewbox. Although not listed directly in the MDN page for rectangles, you can add a fill property to add a fill color to any SVG element. You can use color names, or hex codes for colors:  
<rect width="100" height="100" fill="#aabbcc" />  
Generating a Random Color  
Instead of a fixed color, your design calls for a unique color for each NFT. Add a function to generate this:  
// Function to generate a random color hex code  
function generateRandomColor() public view returns (string memory) {  
 // Generate a pseudo-random number using block.prevrandao  
 uint256 randomNum = uint256(keccak256(abi.encodePacked(block.prevrandao, block.timestamp, msg.sender)));  
  
 // Extract RGB components from the random number  
 bytes memory colorBytes = new bytes(3);  
 colorBytes[0] = bytes1(uint8(randomNum >> 16));  
 colorBytes[1] = bytes1(uint8(randomNum >> 8));  
 colorBytes[2] = bytes1(uint8(randomNum));  
  
 // Convert RGB components to hex string  
 string memory colorHex = string(abi.encodePacked(  
 "#",  
 toHexDigit(uint8(colorBytes[0]) >> 4),  
 toHexDigit(uint8(colorBytes[0]) & 0x0f),  
 toHexDigit(uint8(colorBytes[1]) >> 4),  
 toHexDigit(uint8(colorBytes[1]) & 0x0f),  
 toHexDigit(uint8(colorBytes[2]) >> 4),  
 toHexDigit(uint8(colorBytes[2]) & 0x0f)  
 ));  
  
 return colorHex;  
}  
  
// Helper function to convert a uint8 to a hex character  
function toHexDigit(uint8 d) internal pure returns (bytes1) {  
 if (d < 10) {  
 return bytes1(uint8(bytes1('0')) + d);  
 } else {  
 return bytes1(uint8(bytes1('a')) + d - 10);  
 }  
}  
CAUTION  
Remember, randomness generated using onchain information is not fully secure. A determined attacker could manipulate a block to compromise your contract. That being said, prevrandao is a passable solution for anything not involving a large amount of money.  
Saving the Color to the NFT  
You'll need to generate this color with the function, then save it in a way that it can be retrieved when the tokenURI function is called. Add a mapping to store this relationship:  
mapping (uint => string) public tokenIdToColor;  
Then set the color when the token is minted:  
function mintTo(address \_to) public {  
 counter++;  
 \_safeMint(\_to, counter);  
 tokenIdToColor[counter] = generateRandomColor();  
}  
Finishing the tokenURI Function  
Update your render function to generate the SVG.  
function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<svg xmlns='http://www.w3.org/2000/svg' viewBox='0 0 1024 1024'>",  
 "<rect width='1024' height='1024' fill='",  
 tokenIdToColor[\_tokenId],  
 "' />",  
 "</svg>"  
 )  
 );  
}  
Then update your tokenURI function to use it, and return the SVG as base64 encoded data:  
function tokenURI(uint \_tokenId) public view override returns (string memory) {  
 if(\_tokenId > counter) {  
 revert InvalidTokenId(\_tokenId);  
 }  
  
 string memory json = Base64.encode(  
 bytes(  
 string(  
 abi.encodePacked(  
 '{"name": "',  
 name(),  
 ' #: ',  
 Strings.toString(\_tokenId),  
 '","description": "Random colors are pretty or boring!", "image": "data:image/svg+xml;base64,',  
 Base64.encode(bytes(render(\_tokenId))),  
 '"}'  
 )  
 )  
 )  
 );  
  
 return string(abi.encodePacked(\_baseURI(), json));  
}  
List of NFTs Owned  
Most ERC-721 implementations don't contain an on-contract method to retrieve a list of all the NFTs owned by a single address. The reason for this is that it costs extra gas go manage this list, and the information can be retrieved by using read-only services that analyze blockchain data.  
However, gas prices are getting lower, and adding this data to your contract will reduce your dependency on third-party APIs.  
To track ownership in-contract, first import EnumerableSet from OpenZeppelin:  
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";  
Then enable it for uint sets and add a mapping to relate addresses to token ids.  
// Inside the RandomColorNFT contract  
using EnumerableSet for EnumerableSet.UintSet;  
  
mapping (address => EnumerableSet.UintSet) tokensOwned;  
Finally, utilize the \_update function to handle changes of ownership, including minting:  
function \_update(address to, uint256 tokenId, address auth) internal override(ERC721) returns(address) {  
 // Only remove the token if it is not being minted  
 if (tokenId != counter){  
 tokensOwned[auth].remove(tokenId);  
 }  
 tokensOwned[to].add(tokenId);  
  
 return super.\_update(to, tokenId, auth);  
}  
Now that you have a list of NFTs owned by an address, you can add a function to retrieve all of them. While you're at it, add the json metadata for each token. Doing so lets you get the complete list of NFTs and their metadata for just one RPC call!  
function getNFftsOwned(address owner) public view returns (TokenAndMetatdata[] memory) {  
 TokenAndMetatdata[] memory tokens = new TokenAndMetatdata[](tokensOwned[owner].length());  
 for (uint i = 0; i < tokensOwned[owner].length(); i++) {  
 uint tokenId = tokensOwned[owner].at(i);  
 tokens[i] = TokenAndMetatdata(tokenId, tokenURI(tokenId));  
 }  
 return tokens;  
}  
Testing  
Write some local tests, then [deploy] and test your contract. It can be very tricky to get all the commas, brackets, and single, and double quotes all lined up properly. The surest way to make sure it is working is to check the collection on [Testnet Opensea] or similar.  
Remember, it can take a few minutes for them to register and add the collection. If the metadata or image don't show up correctly, use [Sepolia Basescan] to pull the tokenURI and an online or console base64 decoder to decode and check the json metadata and SVG image.  
Conclusion  
In this lesson, you learned how to make a simple NFT that is entirely onchain. You generated an SVG with a random color, and set up the JSON metadata for your NFT -- entirely onchain! Next, check out our tutorial for Complex Onchain NFTs!  
Random Color NFT Contract  
// SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.24;  
  
import "hardhat/console.sol";  
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";  
import "@openzeppelin/contracts/utils/Base64.sol";  
import "@openzeppelin/contracts/utils/Strings.sol";  
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";  
  
contract RandomColorNFT is ERC721 {  
 using EnumerableSet for EnumerableSet.UintSet;  
  
 mapping (address => EnumerableSet.UintSet) tokensOwned;  
  
 uint public counter;  
  
 mapping (uint => string) public tokenIdToColor;  
  
 error InvalidTokenId(uint tokenId);  
 error OnlyOwner(address);  
  
 constructor() ERC721("RandomColorNFT", "RCNFT") {  
 }  
  
 function mintTo(address \_to) public {  
 counter++;  
 \_safeMint(\_to, counter);  
 tokenIdToColor[counter] = generateRandomColor();  
 }  
  
 struct TokenAndMetatdata {  
 uint tokenId;  
 string metadata;  
 }  
  
 function getNftsOwned(address owner) public view returns (TokenAndMetatdata[] memory) {  
 TokenAndMetatdata[] memory tokens = new TokenAndMetatdata[](tokensOwned[owner].length());  
 for (uint i = 0; i < tokensOwned[owner].length(); i++) {  
 uint tokenId = tokensOwned[owner].at(i);  
 tokens[i] = TokenAndMetatdata(tokenId, tokenURI(tokenId));  
 }  
 return tokens;  
 }  
  
 function shuffleColor(uint \_tokenId) public {  
 if(\_tokenId > counter) {  
 revert InvalidTokenId(\_tokenId);  
 }  
 if(ownerOf(\_tokenId) != msg.sender) {  
 revert OnlyOwner(msg.sender);  
 }  
 tokenIdToColor[\_tokenId] = generateRandomColor();  
 }  
  
 function \_update(address to, uint256 tokenId, address auth) internal override(ERC721) returns(address) {  
 // Only remove the token if it is not being minted  
 if (tokenId != counter){  
 tokensOwned[auth].remove(tokenId);  
 }  
 tokensOwned[to].add(tokenId);  
  
 return super.\_update(to, tokenId, auth);  
 }  
  
 function \_baseURI() internal pure override returns (string memory) {  
 return "data:application/json;base64,";  
 }  
  
 function tokenURI(uint \_tokenId) public view override returns (string memory) {  
 if(\_tokenId > counter) {  
 revert InvalidTokenId(\_tokenId);  
 }  
  
 string memory json = Base64.encode(  
 bytes(  
 string(  
 abi.encodePacked(  
 '{"name": "',  
 name(),  
 ' #: ',  
 Strings.toString(\_tokenId),  
 '","description": "Random colors are pretty or boring!", "image": "data:image/svg+xml;base64,',  
 Base64.encode(bytes(render(\_tokenId))),  
 '"}'  
 )  
 )  
 )  
 );  
  
 return string(abi.encodePacked(\_baseURI(), json));  
 }  
  
 function render(uint \_tokenId) public view returns (string memory) {  
 return string(  
 abi.encodePacked(  
 "<svg xmlns='http://www.w3.org/2000/svg' viewBox='0 0 1024 1024'>",  
 "<rect width='1024' height='1024' fill='",  
 tokenIdToColor[\_tokenId],  
 "' />",  
 "</svg>"  
 )  
 );  
 }  
  
 // Function to generate a random color hex code  
 function generateRandomColor() public view returns (string memory) {  
 // Generate a pseudo-random number using block.prevrandao  
 uint256 randomNum = uint256(keccak256(abi.encodePacked(block.prevrandao, block.timestamp, msg.sender)));  
  
 // Extract RGB components from the random number  
 bytes memory colorBytes = new bytes(3);  
 colorBytes[0] = bytes1(uint8(randomNum >> 16));  
 colorBytes[1] = bytes1(uint8(randomNum >> 8));  
 colorBytes[2] = bytes1(uint8(randomNum));  
  
 // Convert RGB components to hex string  
 string memory colorHex = string(abi.encodePacked(  
 "#",  
 toHexDigit(uint8(colorBytes[0]) >> 4),  
 toHexDigit(uint8(colorBytes[0]) & 0x0f),  
 toHexDigit(uint8(colorBytes[1]) >> 4),  
 toHexDigit(uint8(colorBytes[1]) & 0x0f),  
 toHexDigit(uint8(colorBytes[2]) >> 4),  
 toHexDigit(uint8(colorBytes[2]) & 0x0f)  
 ));  
  
 return colorHex;  
 }  
  
 // Helper function to convert a uint8 to a hex character  
 function toHexDigit(uint8 d) internal pure returns (bytes1) {  
 if (d < 10) {  
 return bytes1(uint8(bytes1('0')) + d);  
 } else {  
 return bytes1(uint8(bytes1('a')) + d - 10);  
 }  
 }  
}  
{  
 "address": "0x59c35beE5eAdeEDDc2c34d419459243bD70AFD72",  
 "abi": [  
 {  
 "inputs": [],  
 "stateMutability": "nonpayable",  
 "type": "constructor"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "sender",  
 "type": "address"  
 },  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 },  
 {  
 "internalType": "address",  
 "name": "owner",  
 "type": "address"  
 }  
 ],  
 "name": "ERC721IncorrectOwner",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "operator",  
 "type": "address"  
 },  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "ERC721InsufficientApproval",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "approver",  
 "type": "address"  
 }  
 ],  
 "name": "ERC721InvalidApprover",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "operator",  
 "type": "address"  
 }  
 ],  
 "name": "ERC721InvalidOperator",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "owner",  
 "type": "address"  
 }  
 ],  
 "name": "ERC721InvalidOwner",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "receiver",  
 "type": "address"  
 }  
 ],  
 "name": "ERC721InvalidReceiver",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "sender",  
 "type": "address"  
 }  
 ],  
 "name": "ERC721InvalidSender",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "ERC721NonexistentToken",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "InvalidTokenId",  
 "type": "error"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "",  
 "type": "address"  
 }  
 ],  
 "name": "OnlyOwner",  
 "type": "error"  
 },  
 {  
 "anonymous": false,  
 "inputs": [  
 {  
 "indexed": true,  
 "internalType": "address",  
 "name": "owner",  
 "type": "address"  
 },  
 {  
 "indexed": true,  
 "internalType": "address",  
 "name": "approved",  
 "type": "address"  
 },  
 {  
 "indexed": true,  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "Approval",  
 "type": "event"  
 },  
 {  
 "anonymous": false,  
 "inputs": [  
 {  
 "indexed": true,  
 "internalType": "address",  
 "name": "owner",  
 "type": "address"  
 },  
 {  
 "indexed": true,  
 "internalType": "address",  
 "name": "operator",  
 "type": "address"  
 },  
 {  
 "indexed": false,  
 "internalType": "bool",  
 "name": "approved",  
 "type": "bool"  
 }  
 ],  
 "name": "ApprovalForAll",  
 "type": "event"  
 },  
 {  
 "anonymous": false,  
 "inputs": [  
 {  
 "indexed": true,  
 "internalType": "address",  
 "name": "from",  
 "type": "address"  
 },  
 {  
 "indexed": true,  
 "internalType": "address",  
 "name": "to",  
 "type": "address"  
 },  
 {  
 "indexed": true,  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "Transfer",  
 "type": "event"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "to",  
 "type": "address"  
 },  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "approve",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "owner",  
 "type": "address"  
 }  
 ],  
 "name": "balanceOf",  
 "outputs": [  
 {  
 "internalType": "uint256",  
 "name": "",  
 "type": "uint256"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [],  
 "name": "counter",  
 "outputs": [  
 {  
 "internalType": "uint256",  
 "name": "",  
 "type": "uint256"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [],  
 "name": "generateRandomColor",  
 "outputs": [  
 {  
 "internalType": "string",  
 "name": "",  
 "type": "string"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "getApproved",  
 "outputs": [  
 {  
 "internalType": "address",  
 "name": "",  
 "type": "address"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "owner",  
 "type": "address"  
 }  
 ],  
 "name": "getNFftsOwned",  
 "outputs": [  
 {  
 "components": [  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 },  
 {  
 "internalType": "string",  
 "name": "metadata",  
 "type": "string"  
 }  
 ],  
 "internalType": "struct RandomColorNFT.TokenAndMetatdata[]",  
 "name": "",  
 "type": "tuple[]"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "owner",  
 "type": "address"  
 },  
 {  
 "internalType": "address",  
 "name": "operator",  
 "type": "address"  
 }  
 ],  
 "name": "isApprovedForAll",  
 "outputs": [  
 {  
 "internalType": "bool",  
 "name": "",  
 "type": "bool"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "\_to",  
 "type": "address"  
 }  
 ],  
 "name": "mintTo",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [],  
 "name": "name",  
 "outputs": [  
 {  
 "internalType": "string",  
 "name": "",  
 "type": "string"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "ownerOf",  
 "outputs": [  
 {  
 "internalType": "address",  
 "name": "",  
 "type": "address"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "\_tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "render",  
 "outputs": [  
 {  
 "internalType": "string",  
 "name": "",  
 "type": "string"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "from",  
 "type": "address"  
 },  
 {  
 "internalType": "address",  
 "name": "to",  
 "type": "address"  
 },  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "safeTransferFrom",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "from",  
 "type": "address"  
 },  
 {  
 "internalType": "address",  
 "name": "to",  
 "type": "address"  
 },  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 },  
 {  
 "internalType": "bytes",  
 "name": "data",  
 "type": "bytes"  
 }  
 ],  
 "name": "safeTransferFrom",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "operator",  
 "type": "address"  
 },  
 {  
 "internalType": "bool",  
 "name": "approved",  
 "type": "bool"  
 }  
 ],  
 "name": "setApprovalForAll",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "\_tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "shuffleColor",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "bytes4",  
 "name": "interfaceId",  
 "type": "bytes4"  
 }  
 ],  
 "name": "supportsInterface",  
 "outputs": [  
 {  
 "internalType": "bool",  
 "name": "",  
 "type": "bool"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [],  
 "name": "symbol",  
 "outputs": [  
 {  
 "internalType": "string",  
 "name": "",  
 "type": "string"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "",  
 "type": "uint256"  
 }  
 ],  
 "name": "tokenIdToColor",  
 "outputs": [  
 {  
 "internalType": "string",  
 "name": "",  
 "type": "string"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "uint256",  
 "name": "\_tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "tokenURI",  
 "outputs": [  
 {  
 "internalType": "string",  
 "name": "",  
 "type": "string"  
 }  
 ],  
 "stateMutability": "view",  
 "type": "function"  
 },  
 {  
 "inputs": [  
 {  
 "internalType": "address",  
 "name": "from",  
 "type": "address"  
 },  
 {  
 "internalType": "address",  
 "name": "to",  
 "type": "address"  
 },  
 {  
 "internalType": "uint256",  
 "name": "tokenId",  
 "type": "uint256"  
 }  
 ],  
 "name": "transferFrom",  
 "outputs": [],  
 "stateMutability": "nonpayable",  
 "type": "function"  
 }  
 ]  
}  
Tags:nft  
ON THIS PAGE  
Prerequisites  
Building the Contract  
Onchain Metadata  
Onchain SVG Image  
Conclusion  
Onchain Metadata  
Onchain SVG Image  
Generating a Random Color  
Saving the Color to the NFT  
Finishing the tokenURI Function  
List of NFTs Owned  
Testing  
Random Color NFT Contract  
Random Color NFT Contract  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/smart-wallet-and-eoa-with-onchainkit  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Use the Coinbase Smart Wallet and EOAs with OnchainKit  
The Coinbase Smart Wallet is a great way to onboard new users to onchain apps, and offers a number of experiential improvements to existing crypto users as well. As we're in a period of adoption and transition, a pain point has developed where a user with both the Smart Wallet and a browser-extension EOA can't always select the wallet they want while connecting to an app.  
In this tutorial, you'll learn how to more easily improve that experience by using Wallet component from OnchainKit to connect your users to your app.  
Objectives  
By the end of this tutorial, you should be able to:  
Create a connection experience using the Wallet component from OnchainKit  
Customize the Wallet component to allow your users to connect from a chosen list of wallets  
Give users the option to select the Coinbase Smart Wallet or EOA while connecting to your app with RainbowKit  
Prerequisites  
Be familiar with modern, frontend web development  
In this tutorial, you'll be working with a React frontend built with Next.js. While you don't need to be an expert, we'll assume that you're comfortable with the basics.  
Possess a general understanding of the EVM and smart contract development  
This tutorial assumes that you're reasonably comfortable writing basic smart contracts. If you're just getting started, jump over to our Base Learn guides and start learning!  
Understand providers and connectors  
You'll need to be familiar with how to connect an onchain app to the blockchain with a provider. If you're not, start with the Onchain App Development section of Base Learn, or at least complete the tutorial Introduction to Providers.  
Coinbase Wallets  
You need to have both the Coinbase Wallet and Coinbase Smart Wallet for this tutorial. You need to set up the Coinbase Wallet in advance, but you can create a smart wallet during the tutorial.  
The Default Experience  
Begin by using the Onchain App Template to quickstart a new app. Click the green Use this template button, create a new repo, and clone it.  
You'll need a WalletConnect id if you don't have one already. Obtain it, the duplicate .env.local.example and rename it .env.local. Change NEXT\_PUBLIC\_WC\_PROJECT\_ID to your id.  
Then install dependencies, and run the app.  
bun install  
bun run dev  
::info  
To bring the world onchain, we'll need to speak in terms people are already comfortable with. Consider using Log In rather than Connect Wallet!  
:::  
The demo app contains sections showing a number of OnchainKit components, including the Wallet. Click Log In in your browser with the Coinbase wallet extension present, select Coinbase Wallet, and you'll get the expected experience for an EOA wallet user with the browser extension.  
Next, open the app in a private browser window with extensions disabled, and try again. This time, you'll get the Smart Wallet experience. If you don't have one already, you can create one now. They're neat!  
Customizing the List of Wallets  
You can customize the list of wallets with the Wallet Aggregator.  
Setting Up the Wallet Aggregator  
Open src/app/page.tsx. Here, you find the root page for the app. You can see the <WalletWrapper />, which is what you used to log in and out.  
Open src/components/WalletWrapper.tsx, and src/wagmi.ts. Take a quick look at WalletWrapper. It's a robust implementation of several OnchainKit components that shows how you can create a login experience that works well for all types of users, and provides the <Identity> component once logged in.  
You can customize this by making some changes to wagmi.ts.  
Start with wagmi.ts. Update it import the RainbowKit connectors for the wallets you want to use (We just picked the ones a the end of the supported list. Do your homework on which wallets to support!):  
import { http, createConfig } from 'wagmi';  
import { baseSepolia } from 'wagmi/chains';  
import {  
 coinbaseWallet,  
 metaMaskWallet,  
 rainbowWallet,  
 uniswapWallet,  
 walletConnectWallet,  
 xdefiWallet,  
 zerionWallet,  
} from '@rainbow-me/rainbowkit/wallets';  
import { connectorsForWallets } from '@rainbow-me/rainbowkit';  
Then, update the connectorsForWallets to use the wallets you've selected in the groupings of your choice:  
const connectors = connectorsForWallets(  
 [  
 {  
 groupName: 'Recommended Wallet',  
 wallets: [coinbaseWallet],  
 },  
 {  
 groupName: 'Other Wallets',  
 wallets: [rainbowWallet, metaMaskWallet, walletConnectWallet],  
 },  
 {  
 groupName: 'U Wallets',  
 wallets: [uniswapWallet],  
 },  
 {  
 groupName: 'X Wallets',  
 wallets: [xdefiWallet],  
 },  
 {  
 groupName: 'Z Wallets',  
 wallets: [zerionWallet],  
 },  
 ],  
 {  
 appName: 'onchainkit',  
 projectId,  
 },  
);  
Try connecting again. You'll see your updated list with the wallets organized to your preference!  
Tuning the Coinbase Wallet Connection  
DANGER  
If you've already connected with the site, the below won't work unless you clear site data! It will appear that the flag does nothing.  
In Chrome: Developer Tools -> Application Tab -> Storage Tab -> Clear Site Data button In Firefox: Developer Tools -> Storage -> Right-click each item -> Delete All  
With RainbowKit, you can force the connector to use the smart wallet or EOA with:  
coinbaseWallet.preference = 'smartWalletOnly';  
And:  
coinbaseWallet.preference = 'eoaOnly';  
The default is:  
coinbaseWallet.preference = 'all';  
Set this line at the root level of wagmi.ts, under the imports:  
import { http, createConfig } from 'wagmi';  
import { baseSepolia } from 'wagmi/chains';  
import {  
 coinbaseWallet,  
 metaMaskWallet,  
 rainbowWallet,  
 walletConnectWallet,  
} from '@rainbow-me/rainbowkit/wallets';  
import { connectorsForWallets } from '@rainbow-me/rainbowkit';  
  
coinbaseWallet.preference = 'all';  
The all setting exhibits the behavior you've observed before. If the EOA is present, it will be used. Otherwise, the Smart Wallet UI/UX will pop up.  
You can use these settings to direct users of your app to your preferred use case, or tie it to a UI element to give them a choice.  
INFO  
The Coinbase Smart Wallet will support user-selected choice of which wallet to use soon!  
Conclusion  
In this tutorial, you've learned how to use OnchainKit to log your users into your onchain app. You've also learned how to customize the connection experience with RainbowKit, enabling your users a broader choice of which wallet to use with your app and making sure all are able to enjoy it.  
Tags:account abstractionfrontendOnchainKitSmart Wallet  
ON THIS PAGE  
Prerequisites  
The Default Experience  
Be familiar with modern, frontend web development  
Possess a general understanding of the EVM and smart contract development  
Understand providers and connectors  
Coinbase Wallets  
Customizing the List of Wallets  
Tuning the Coinbase Wallet Connection  
Setting Up the Wallet Aggregator  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/smart-wallet-and-rainbowkit  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Coinbase Smart Wallet with RainbowKit  
The Coinbase Smart Wallet is a great way to onboard new users to onchain apps, and offers a number of experiential improvements to existing crypto users as well. As we're in a period of adoption and transition, a pain point has developed where a user with both the Smart Wallet and a browser-extension EOA can't always select the wallet they want while connecting to an app.  
In this tutorial, you'll learn how to improve that experience when using RainbowKit to connect your users to your app. An even easier development path can be found in our tutorial for using the Wallet component from OnchainKit, which uses RainbowKit under the hood.  
Objectives  
By the end of this tutorial, you should be able to:  
Customize the list of Wallets in the RainbowKit connection experience  
Give users the option to select the Coinbase Smart Wallet or EOA while connecting to your app with RainbowKit  
Prerequisites  
Be familiar with modern, frontend web development  
In this tutorial, you'll be working with a React frontend built with Next.js. While you don't need to be an expert, we'll assume that you're comfortable with the basics.  
Possess a general understanding of the EVM and smart contract development  
This tutorial assumes that you're reasonably comfortable writing basic smart contracts. If you're just getting started, jump over to our Base Learn guides and start learning!  
Understand providers and connectors  
You'll need to be familiar with how to connect an onchain app to the blockchain with a provider. If you're not, start with the Onchain App Development section of Base Learn, or at least complete the tutorial Introduction to Providers.  
Coinbase Wallets  
You need to have both the Coinbase Wallet and Coinbase Smart Wallet for this tutorial. You need to set up the Coinbase Wallet in advance, but you can create a smart wallet during the tutorial.  
The Default Experience  
Start a new project with the RainbowKit quick start. Install dependencies, run the project, and attempt to connect with a browser that has the Coinbase Wallet browser extension installed. Clicking on Coinbase will automatically connect with your EOA browser extension wallet.  
What about the smart wallet? Isn't it supposed to work automatically?  
It does, but only if the user does not have the browser extension installed. Open a private window with extensions disabled and try again. Now, you will be directed to use the smart wallet to log in.  
As mentioned above, this experience isn't bad for users of one type of wallet or the other, but it makes things difficult for users who are using both types of wallet and may want to choose one or the other when interacting with your app. It also might not be the listing or ordering of wallets you prefer.  
Customizing the List of Wallets  
To fix this UI/UX problem, you can create a custom wallet list. Open src/wagmi.ts:  
As mentioned above, this experience isn't bad for users of one type of wallet or the other, but it makes things difficult for users who are using both types of wallet and may want to choose one or the other when interacting with your app.  
Customizing the List of Wallets  
To fix this UI/UX problem, you can create a custom wallet list. Open src/wagmi.ts:  
Change the list imported networks to base and baseSepolia  
Import connectorsForWallets from RainbowKit, instead of getDefaultConfig  
Import the coinbaseWallet, and any other wallets you wish to support  
Import http and createConfig from wagmi  
import { http, createConfig } from 'wagmi';  
import { base, baseSepolia } from 'wagmi/chains';  
import {  
 coinbaseWallet,  
 metaMaskWallet,  
 rainbowWallet,  
 walletConnectWallet,  
} from '@rainbow-me/rainbowkit/wallets';  
import { connectorsForWallets } from '@rainbow-me/rainbowkit';  
Next, use connectorsForWallets to create a list of wallets, organized in the groups you prefer. You'll need to get a projectId from WalletConnect if you don't have one already.  
const connectors = connectorsForWallets(  
 [  
 {  
 groupName: 'Recommended',  
 wallets: [coinbaseWallet],  
 },  
 {  
 groupName: 'Popular',  
 wallets: [rainbowWallet, metaMaskWallet],  
 },  
 {  
 groupName: 'Wallet Connect',  
 wallets: [walletConnectWallet],  
 },  
 ],  
 {  
 appName: 'Your App Name',  
 projectId: '<YOUR WALLETCONNECT PROJECT ID>',  
 },  
);  
Finally, export your config using the connectors and networks you've selected.  
export const config = createConfig({  
 connectors,  
 chains: [base, baseSepolia],  
 ssr: true,  
 transports: {  
 [base.id]: http(),  
 [baseSepolia.id]: http(),  
 },  
});  
Now, the connection experience contains the wallets in the order you selected!  
Tuning the Coinbase Wallet Connection  
DANGER  
If you've already connected with the site, the below won't work unless you clear site data! It will appear that the flag does nothing.  
In Chrome: Developer Tools -> Application Tab -> Storage Tab -> Clear Site Data button In Firefox: Developer Tools -> Storage -> Right-click each item -> Delete All  
With RainbowKit, you can force the connector to use the smart wallet or EOA with:  
coinbaseWallet.preference = 'smartWalletOnly';  
And:  
coinbaseWallet.preference = 'eoaOnly';  
The default is:  
coinbaseWallet.preference = 'all';  
Set this line at the root level of the document, under the imports:  
import { http, createConfig } from 'wagmi';  
import { baseSepolia } from 'wagmi/chains';  
import {  
 coinbaseWallet,  
 metaMaskWallet,  
 rainbowWallet,  
 walletConnectWallet,  
} from '@rainbow-me/rainbowkit/wallets';  
import { connectorsForWallets } from '@rainbow-me/rainbowkit';  
  
coinbaseWallet.preference = 'all';  
This setting exhibits the behavior you've observed before. If the EOA is present, it will be used. Otherwise, the Smart Wallet UI/UX will pop up.  
You can use these settings to direct users of your app to your preferred use case, or tie it to a UI element to give them a choice.  
INFO  
The Coinbase Smart Wallet will support user-selected choice of which wallet to use very soon!  
=======  
Import createConfig from wagmi  
import { connectorsForWallets } from '@rainbow-me/rainbowkit';  
import { createConfig } from 'wagmi';  
import {  
 coinbaseWallet,  
 metaMaskWallet,  
 walletConnectWallet,  
} from '@rainbow-me/rainbowkit/wallets';  
import { base, baseSepolia } from 'wagmi/chains';  
Conclusion  
In this tutorial, you've learned how connect users to your app using RainbowKit. You've also learned how to customize the list of wallets, their groupings, and their ordering. Finally, you've learned how to manage your preference for your users to be able to use the Coinbase Smart Wallet or browser extension EOA.  
Tags:account abstractionfrontendSmart Wallet  
ON THIS PAGE  
Prerequisites  
The Default Experience  
Be familiar with modern, frontend web development  
Possess a general understanding of the EVM and smart contract development  
Understand providers and connectors  
Coinbase Wallets  
Customizing the List of Wallets  
Customizing the List of Wallets  
Tuning the Coinbase Wallet Connection  
Conclusion  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/thirdweb-unreal-nft-items  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Thirdweb and Unreal - NFT Items  
thirdweb provides a number of contracts and tools for building onchain. Their Gaming SDK enables seamless onboarding, cross-platform support, and provides many other features. It's compatible with Unreal Engine and can be used to enable onchain elements in your games.  
In this tutorial, you'll learn how to add NFT item usage on top of the demo game you build in their Unreal Engine Quickstart. Specifically, you'll use an NFT collection of random colors to change the color of the player's race car.  
Objectives  
By the end of this tutorial you should be able to:  
Pull a user's NFTs into an Unreal Engine Game  
Apply elements from the NFT to game entities  
Award NFTs to players for game accomplishments or actions  
Prerequisites  
ERC-721 Tokens  
This tutorial assumes that you are able to write, test, and deploy your own ERC-721 tokens using the Solidity programming language. If you need to learn that first, check out our content in Base Learn or the sections specific to ERC-721 Tokens!  
Unreal Engine  
This tutorial will cover everything you need to know to accomplish the learning objectives, but it won't teach you how to make a game. You'll need to take further steps to learn the Unreal Engine on your own. You'll also need to have Visual Studio or Visual Studio Code set up to edit and compile Unreal files.  
Onchain Apps  
The tutorial assumes you're comfortable with the basics of deploying an app and connecting it to a smart contract. If you're still learning this part, check out our tutorials in Base Learn for [Building an Onchain App].  
Reviewing the Contract  
In our tutorial for building a Simple Onchain NFTs, you can find an example of an ERC-721 NFT contract. It's an extension of the OpenZeppelin ERC-721 implementation. When a user mints, they're granted an NFT with a random color. The metadata is fully onchain, as is the svg image. The image is a simple 1024\*1024 rect, with a fill of the randomly generated color.  
If the user dislikes the color, they may shuffle it and the NFT will change to a new randomly-selected color.  
These NFTs are not restricted for trading. The contract includes a utility function, getNftsOwned, which will return an array containing the tokenId and base64-encoded metadata string for all tokens currently owned by the provided address.  
Getting Started with Unreal  
Continue below for some tips on completing the Unreal Engine Quickstart tutorial provided by thirdweb. This tutorial will guide you through installing the Unreal Engine and setting up the major components of the thirdweb Gaming SDK.  
It will also guide you through setting up the website and backend that are needed to support the game integration. The client in their example uses Next.js, and the server is built with Node and Express.  
Setting up the Engine  
First, you need to set up the Engine. For testing, you can run it locally with a Docker container.  
If you need to, install or update Docker and Postgres.  
Start Postgres:  
docker run -p 5432:5432 -e POSTGRES\_PASSWORD=postgres -d postgres  
Make sure Docker desktop is running.  
Create a thirdweb API key. Allow localhost:3000 and localhost:8000 when creating your api key. When you deploy, you'll need to update the allowed domains.  
The command to launch the engine itself is complicated and has many parameters. You'll want to create a file and run it from that. Create thirdweb-engine.sh in a convenient location and add:  
docker run \  
 -e ENCRYPTION\_PASSWORD="<encryption\_password>" \  
 -e THIRDWEB\_API\_SECRET\_KEY="<thirdweb\_secret\_key>" \  
 -e ADMIN\_WALLET\_ADDRESS="<admin\_wallet\_address>" \  
 -e POSTGRES\_CONNECTION\_URL="postgresql://postgres:postgres@host.docker.internal:5432/postgres?sslmode=disable" \  
 -e ENABLE\_HTTPS=true \  
 -p 3005:3005 \  
 --pull=always \  
 --cpus="0.5" \  
 thirdweb/engine:latest  
Enter your THIRDWEB\_API\_SECRET\_KEY and the wallet address you sign into thirdweb with as the ADMIN\_WALLET\_ADDRESS. You can see a full list of environment variables in the docs, but shouldn't need to set any others now.  
Give your script permission to run with chmod +x ./thirdweb-engine.sh then run it with ./thirdweb-engine.sh .  
It will take awhile to spin up, and you can ignore the warning about the Chain Indexer Listener not started....  
Once the engine is running, navigate to https://localhost:3005/json. Click through the warning that the connection is not secure. Doing so allows your browser to connect to your engine instance.  
CAUTION  
We do not have an official browser recommendation, but during our testing, Chrome worked and Brave did not with the engine setup and configuration.  
Navigate to the thirdweb engine dashboard, and click the Import button. Enter a name and the local address for your engine instance:  
Next, you must add your wallet to the engine instance. Open up the instance in the dashboard, then click the Import button next to Backend Wallets. Enter your secret key for the wallet.  
DANGER  
Remember, the wallet key gives full access to all assets within any wallet. Use separate wallets for development and individual production tasks. Don't hold or fund a production wallet with any assets other than the minimum amount necessary for the task it is accomplishing.  
Be sure to fund this wallet with Base Sepolia ETH. It will be paying the gas for transactions.  
CAUTION  
The key to your wallet is stored in ephemeral memory in the engine itself. You'll need to re-add it whenever you restart the engine.  
Setting up the Client and Server  
Clone the engine-express repo. CD into the client and server repos and run yarn to install dependencies, then CD back to root and run yarn again.  
In both the client and server folders, copy or rename the .env.example files as .env.  
Client  
In the client .env:  
Set NEXT\_PUBLIC\_THIRDWEB\_CLIENT\_ID to the Client ID matching your thirdweb API key  
You don't need to change the NEXT\_PUBLIC\_BACKEND\_URL  
Set the NEXT\_PUBLIC\_THIRDWEB\_AUTH\_DOMAIN as localhost  
In the server .env:  
Don't change the THIRDWEB\_ENGINE\_URL. (It is supposed to be https)  
Set the THIRDWEB\_ENGINE\_BACKEND\_WALLET to the same as you used in the engine setup  
Set the THIRDWEB\_AUTH\_DOMAIN as localhost  
Set the THIRDWEB\_API\_SECRET\_KEY to the Secret Key matching your thirdweb API key  
Set the THIRDWEB\_AUTH\_PRIVATE\_KEY to the private key matching your backend engine wallet  
Open client/components/ThirdwebProvider.tsx. Update the activeChain to Base Sepolia.  
import { BaseSepoliaTestnet } from '@thirdweb-dev/chains';  
  
// This is the chainId your dApp will work on.  
const activeChain = BaseSepoliaTestnet;  
Server  
Open server/src/controllers/engineController.ts. Update the consts at the beginning to load from environment variables:  
const ENGINE\_URL = process.env.THIRDWEB\_ENGINE\_URL;  
const BACKEND\_WALLET = process.env.THIRDWEB\_ENGINE\_BACKEND\_WALLET;  
const ERC20\_CONTRACT = process.env.ERC20\_CONTRACT;  
const CHAIN = process.env.CHAIN;  
You'll need to deploy your own version of the Token Drop contract. Click Deploy Now, then enter the name, symbol, and image of your choosing.  
Select Base Sepolia Testnet as the Network / Chain.  
You can leave the Recipient Address as your connected address, and you don't need to do an advanced configuration.  
Click Deploy Now, and approve the transactions to deploy the contract and add it to your dashboard.  
Next, click the Claim Conditions tab on the left side nav. Then click the + Add Phase button and select Public. Review the options, but don't change any of them for this demo. Click Save Phases.  
CAUTION  
If later in the tutorial, you get an error when you attempt to claim a token, but not an error every four seconds failing to read the balance, it's because you missed this step.  
Copy the address from the dashboard:  
Return to the .env for your server, and add:  
ERC20\_CONTRACT=0x... # Your Address  
CHAIN=84532 # Base Sepolia  
Run the client and server with yarn client and yarn server. Navigate to localhost:3000, create a user, and link a wallet.  
Setting Up the Game  
Clone the thirdweb Unreal Demo, and open it with the Unreal Editor. Do so by clicking the Recent Projects tab in the upper left, then Browse, in the lower right.  
Open the folder cloned from the repo and select unreal\_demo.uproject. You may need to convert the project to the current version of Unreal. Click the Open a copy button.  
When the scene loads, double-click Scene\_Game in the upper-right corner.  
Before you can play, you need to do some config. Scroll down in the Outliner until you find ThirdWebManager. Click the Open Thirdweb Manager button to open the file in your editor.  
Then, click the green play button at the top of the viewport.  
Log in using the credentials you created on the website, and play the game for a minute or two. If you get a 404, check that your engine, client, and server are all still running.  
CAUTION  
The demo does not actually have a database connected for users. You'll need to recreate your user each time you restart the server. For production, you'll need to swap this out with an actual database.  
If you get an error 500 "No configured wallet found with address 0xABCD....", it's because you didn't add your wallet in the thirdweb engine dashboard.  
Otherwise, the game should run, and you should receive an ERC20 NFT every time you collect one of the orange orbs on the race track.  
Adding the Color Changer  
Your next goal is to make it so that your players can use their Random Color NFTs as skins on the race car. You'll need to deploy the contract provided below, set it up to be accessed via the server and engine, and finally, enable the colors from the NFTs to be used to change the color of the car.  
Deploying the Contract  
You'll use thirdweb's platform to deploy this contract as well. Open a new folder in your editor and run:  
npx thirdweb create contract  
Then:  
Name the project - random-color-nft, or . if you run the script from the folder where you want the project  
Select your preference of Forge or Hardhat  
Name the NFT contract - RandomColorNFT  
Select Empty Contract  
Open contracts/Contract.sol and replace the contents with the contract provided below.  
You'll need to import or install the OpenZeppelin contracts. You may also need to update the config for the development environment you're using to 0.8.24.  
Run yarn build.  
Select y to install the thirdweb package and wait for the script to complete.  
Run yarn deploy.  
If you haven't linked your device to your thirdweb account, the browser will open to a page asking you to make the connection. Do so now.  
After the script runs for a moment, it will open the thirdweb dashboard with the deployment UI open. Select Base Sepolia Testnet as your network, then click the Deploy Now button. Sign the transaction and wait for the contract to deploy.  
Adding the Contract to the Server  
Copy the address for the contract to the clipboard and return to thirdweb-engine-express. Open server/.env and add:  
RANDOM\_COLOR\_NFT\_CONTRACT=<your contract address>  
Open server/src/controllers/engineController.ts and add it there as well:  
const RANDOM\_COLOR\_NFT\_CONTRACT = process.env.RANDOM\_COLOR\_NFT\_CONTRACT;  
Now, using claimERC20 as a template, add a function to claimRandomColorNFT. It's identical, except the url, body, and error message are:  
// Other code...  
const url = `${ENGINE\_URL}/contract/${CHAIN}/${RANDOM\_COLOR\_NFT\_CONTRACT}/write`;  
// Other code  
const body = {  
 functionName: 'mintTo',  
 args: [user.ethAddress],  
};  
// Other code  
res.status(400).json({ message: 'Error claiming RandomColorNFT' });  
INFO  
A better practice for production would be to make a more generalized function that can handle multiple requests to your contracts. We're skipping that for now to avoid needing to refactor the existing collectibles in the game.  
Next, you need to add a route for this function. Open server/src/routes/engineRoutes.ts. Import claimRandomColorNFT and add a route for it:  
router.post('/claim-random-color-nft', claimRandomColorNFT);  
Collecting the NFT from the Game  
Return to the Unreal Editor and open ThirdwebManager.cpp:  
.  
Similarly to what you did in the server, use the existing PerformClaim() as a template to add a function for PerformNFTClaim(). The only thing different is the name of the function and the URL:  
HttpRequest->SetURL(this->ServerUrl + "/engine/claim-random-color-nft");  
INFO  
Again, it would be better practice to generalize this function, but you can skip that for now to avoid needing to update all the collectibles.  
Next, you need to let the editor know about this new function. Open Source/unreal\_demo/Public/ThirdwebManager.h. Add your new function under the one for PerformClaim();  
// Function to perform the NFT claim operation  
UFUNCTION(BlueprintCallable, Category = "Thirdweb")  
void PerformNFTClaim();  
Build your project  
Once it's done compiling, return UnrealEditor. In the Outliner, open the folder for Collectibles and click Edit Collectible. In the new window, click File->Save As... and save a copy as CollectibleNFT.  
Open the Content Drawer at the bottom, search for CollectibleNFT, and drag one into the scene. Find it in the Outliner and click Edit Collectible NFT.  
Find the Perform Claim function call and replace it with Perform NFT Claim. Note that the Target is passed from Get Actor of Class.  
You'll want to be able to tell this collectible apart, so click on the mesh for Collectible on the left side in the Component tree, then on the Details panel on the right, find the Materials section and change it to MI\_Solid\_Blue.  
Click the icons at the top to Compile and save your asset.  
From the content drawer, drag your asset into the viewport.  
You should now see a blue orb floating where you placed it.  
Make sure the orb is low enough to drive through, then run the game. Collect the orb, then verify on a block explorer that you received the NFT.  
Tinting the Car  
In the content browser, open All>Content>Vehicles>SportsCar>Materials. Right-click in an empty spot and select Material>Material Parameter Collection. Name yours NFT\_MPS. Open the collection, click the + to add an item to Vector Parameters and create the color of your choosing. Bright red is a good option to make your change very visible.  
Right-click in an empty spot again and select Create Basic Asset>Material. Name your new material M\_NFT\_Color. Open it by double-clicking.  
Right-click on the graph and add a Collection Parameter node. In the Details panel on the left, select your NFT\_MPS collection and pick the first vector for Parameter Name  
Connect the output to the Base Color of M\_NFT\_Color, then save and close the editor.  
Again in the content browser, right-click on the M\_NFT\_Color asset and select Create Material Instance. Name the instance MI\_NFT\_Color.  
Navigate to the sports car mesh located in VehicleTemplate>Blueprints>SportsCar and double-click to open the SportsCar\_pawn. Select the Mesh from the Components tree and you should see the car in the editor.  
On the right side, change the Element 2 material to MI\_NFT\_Color. The car is now bright red. Radical! Take your newly red car for a spin.  
Fetching the NFT Colors  
Return to engine-express and open engineController.ts. Add a function to getNFTColors that uses the read endpoint to call the getNFTsOwned function.  
export const getNFTColors = async (req: Request, res: Response) => {  
 const { authToken } = req.body;  
 if (!authToken || !userTokens[authToken]) {  
 return res.status(400).json({ message: 'Invalid auth token' });  
 }  
 const user = userTokens[authToken];  
 try {  
 const url = `${ENGINE\_URL}/contract/${CHAIN}/${RANDOM\_COLOR\_NFT\_CONTRACT}/read?functionName=getNftsOwned&args=${user.ethAddress}`;  
 const headers = {  
 'x-backend-wallet-address': BACKEND\_WALLET,  
 Authorization: `Bearer ${process.env.THIRDWEB\_API\_SECRET\_KEY}`,  
 };  
  
 const response = await axiosInstance.get(url, { headers: headers });  
  
 // TODO: Extract the color from the image  
  
 // TODO: Replace response  
 res.json(response.data);  
 } catch (error) {  
 console.error(error);  
 res.status(400).json({ message: 'Error getting NFT data' });  
 }  
};  
You'll also need to add this function to engineRoutes.ts:  
router.post('/get-nft-colors', getNFTColors);  
Return to engineController.ts.  
Because Unreal doesn't support SVGs, you'll need to extract the color from your NFT metadata, and pass that to use in the material you created. Start by adding a type for the response, and for the JSON metadata:  
type NFTData = {  
 tokenId: bigint;  
 metadata: string;  
};  
  
type JSONMetadata = {  
 name: string;  
 description: string;  
 image: string;  
};  
You'll also need helper functions to decode the base64 encoded metadata and SVG, then get the color from the SVG.  
function getJsonMetadata(nft: NFTData) {  
 const base64String = nft.metadata.split(',')[1];  
 const jsonString = atob(base64String);  
 return JSON.parse(jsonString) as JSONMetadata;  
}  
  
function getColorFromBase64StringSVG(base64String: string) {  
 const base64Data = base64String.split(',')[1];  
 const svgString = atob(base64Data);  
 const color = svgString.match(/fill=['"](#[0-9a-fA-F]{6})['"]/);  
 return color ? color[1] : '#000000';  
}  
Use these to extract an array of colors and return it:  
const nfts = response.data.result.map((item: any) => {  
 return {  
 tokenId: item[0],  
 metadata: item[1],  
 };  
});  
  
const metadata = nfts.map((nft: NFTData) => getJsonMetadata(nft));  
const colors = metadata.map((m: JSONMetadata) => getColorFromBase64StringSVG(m.image));  
  
res.json(colors);  
// Delete res.json(response.data);  
INFO  
To test with Postman or similar, comment out the check for a valid authToken and hardcode in an address that you know has NFTs.  
Getting the Colors into the Game  
Return to the game in your code editor, and open ThirdwebManager.cpp and ThirdwebManager.h. As before, add a function to call and endpoint on your server. This time to retrieve the array of colors. You'll need to do a little more for this one to set an in-game variable for the colors.  
First, you'll need to add a new multicast delegate type to handle the response in ThirdwebManager.h:  
// ThirdwebManager.h  
DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE\_TwoParams(FOnNFTColorsResponse, bool, bWasSuccessful, const TArray<FString> &, ResponseArray);  
And expose it to the editor:  
// ThirdwebManager.h  
// This delegate is triggered in C++, and Blueprints can bind to it.  
UPROPERTY(BlueprintAssignable, Category = "Thirdweb", meta = (DisplayName = "OnNFTColorsResponse"))  
FOnNFTColorsResponse OnNFTColorsResponse;;  
Then, add the function to ThirdwebManager.cpp. It's similar, but instead hits the endpoint for the NFT color array and uses the response you just created. It also expects the response to be an array of strings instead of searching for a property called result:  
// ThirdwebManager.cpp  
void AThirdwebManager::GetNFTColors()  
{  
 TSharedRef<IHttpRequest, ESPMode::ThreadSafe> HttpRequest = FHttpModule::Get().CreateRequest();  
 HttpRequest->SetURL(this->ServerUrl + "/engine/get-nft-colors"); // The endpoint to get the NFT colors  
 HttpRequest->SetVerb("POST");  
 HttpRequest->SetHeader(TEXT("Content-Type"), TEXT("application/json"));  
  
 TSharedPtr<FJsonObject> JsonObject = MakeShareable(new FJsonObject);  
 JsonObject->SetStringField("authToken", AuthToken);  
  
 FString OutputString;  
 TSharedRef<TJsonWriter<>> Writer = TJsonWriterFactory<>::Create(&OutputString);  
 FJsonSerializer::Serialize(JsonObject.ToSharedRef(), Writer);  
  
 UE\_LOG(LogTemp, Warning, TEXT("OutputString: %s"), \*OutputString);  
  
 HttpRequest->SetContentAsString(OutputString);  
  
 HttpRequest->OnProcessRequestComplete().BindLambda([this](FHttpRequestPtr Request, FHttpResponsePtr Response, bool bWasSuccessful)  
 {  
 if (bWasSuccessful && Response.IsValid())  
 {  
 int32 StatusCode = Response->GetResponseCode();  
 if (StatusCode == 200)  
 {  
 TArray<TSharedPtr<FJsonValue>> JsonArray;  
 TSharedRef<TJsonReader<>> Reader = TJsonReaderFactory<>::Create(Response->GetContentAsString());  
 if (FJsonSerializer::Deserialize(Reader, JsonArray) && JsonArray.Num() > 0)  
 {  
 TArray<FString> ResponseArray;  
 for (const TSharedPtr<FJsonValue>& Value : JsonArray)  
 {  
 FString StringValue;  
 if (Value->TryGetString(StringValue))  
 {  
 ResponseArray.Add(StringValue);  
 }  
 }  
 this->OnNFTColorsResponse.Broadcast(true, ResponseArray);  
 UE\_LOG(LogTemp, Warning, TEXT("Get NFT Color response: %s"), \*Response->GetContentAsString());  
 return;  
 }  
 this->OnNFTColorsResponse.Broadcast(false, TArray<FString>());  
 }  
 else  
 {  
 FString ErrorMsg = FString::Printf(TEXT("HTTP Error: %d, Response: %s"), StatusCode, \*(Response->GetContentAsString()));  
 TArray<FString> ErrorArray;  
 ErrorArray.Add(ErrorMsg);  
 this->OnNFTColorsResponse.Broadcast(false, ErrorArray);  
 UE\_LOG(LogTemp, Warning, TEXT("ErrorMsg: %s"), \*ErrorMsg);  
 }  
 }  
 else  
 {  
 TArray<FString> ErrorArray;  
 ErrorArray.Add(TEXT("Failed to connect to the server."));  
 this->OnNFTColorsResponse.Broadcast(false, ErrorArray);  
 UE\_LOG(LogTemp, Warning, TEXT("Failed to connect to the server."));  
 } });  
  
 HttpRequest->ProcessRequest();  
}  
Finally, expose this function to the editor.  
// ThirdwebManager.h  
// Function to perform the get NFT Colors operation  
UFUNCTION(BlueprintCallable, Category = "Thirdweb")  
void GetNFTColors();  
Compile and reload the project in the editor.  
In the content browser, find and open Content>\_Thirdweb>Blueprints>Canvas\_HUD.  
Under the text field for Tokens, drag a new Text widget in. Set the name at the top to Label\_Colors and check Is Variable. Change the Content to Colors. If you put it on the right side, move the Anchor to the upper right corner.  
In the upper right, click the Graph tab. Add a Sequence node to split the flow after Get Actor Of Class. Following the same pattern as the flow that gets the balance response, add one that gets the NFT colors.  
Create the Bind Event to OnNFTColorsResponse node first, then create the Custom Event node from dragging from Event.  
For now, simply grab the last color in the array and set it in the HUD. To get it, drag off the Response Array in OnNFTColorResponseReceived and add a Last Index node. Drag again from the Response Array and add a Get (Ref) node. Connect the output of Last Index to the Input of Get. From there, drag from the output of Get and add a To Text (String) node.  
Drag out of the exec (white) connector of OnNFTColorResponseReceived and add a Branch and connect Was Successful to the Condition. For the True state, drag and add a SetText (Text). Right click and add a reference to Label Colors and drag it to the Target of SetText. Connect the Return Value of To Text (String) to In Text.  
Finally, drag off Bind Event to OnNFTColorsResponse and add a Set Timer by Function Name node. Connect the Return Value of Get Actor Of Class to Object. Set the Function Name to GetNFTColors and the Time to 2.0.  
You should end up with something like this:  
Compile the blueprint then run the game. You should see that last color in the array in the HUD, and you should see the full list printed in the console every two seconds.  
CAUTION  
If you have an error in your GetNFTColors function that prevents .Broadcast from being called, nothing in the NFT Colors branch of this blueprint will run, including printing to the console.  
Changing the Color of the Car  
Now that you have the colors, you can use them to change the color of your car! For now, you can just set the car to the last color, but on your own you'll want to add a UI widget to allow the player to pick their color.  
INFO  
If you really wanted to get fancy, you could modify the contract to emit an event containing the color of the newly-minted NFT, extract that from the receipt, and optimistically make it available to the player a few seconds earlier.  
Unreal doesn't use hex colors, so you'll need to convert your hex string to a linear color and save it in the Material Parameter Collection you created earlier.  
Converting the hex code with a blueprint is very complicated. Luckily, Unreal has a helpful community that has created many utilities, including a conversion function.  
DANGER  
Copying and pasting code for a game engine isn't quite as dangerous as copying and pasting unknown smart contract code, but you're working at the intersection of these worlds. Be sure to review and make sure you understand anything you find online.  
In the content browser, add a Blueprints>Blueprint Function Library called ColorUtils. In it, add a function called HexStringToColor.  
Copy the code from the community site and paste it into the function. Connect the Hex String to Color node to the SET node attached to Make Array, then from SET to the Return Node.  
Compile, and you'll get an error. Find and right-click on the error in the Hex Code node, then select Create local variable. Recompile and the error will resolve.  
You also need to input the string you want converted. Select the Hex String to Color node and click the + button by Inputs located in the panel on the right. Name it hexString and give it a string type. Hex String will now appear as a value in the Hex String to Color node. Connect it to the Source String input in the Replace node.  
Compile one last time, then save and close ColorUtils.  
Return to Canvas\_HUD and open the Graph. Drag out of the SetText node that adds the color to the HUD and add a Hex String to Color node. The function expects alpha values in the hex code. To add this connect a second output of the string array GET to an Append function and append ff in the B input. Connect the Return Value to the Hex String input in Hex String to Color.  
Finally, add a Set Vector Parameter Value. Select NFT\_MPS for the collection and Vector for the Parameter Name. Connect the Liner Color output of Hex String to Color to the Parameter Value input.  
Compile, save, and close Canvas\_HUD. Run the game. Your car will start red, but after the response from the server, it will turn the color of your last NFT! Drive and collect the NFT collectible, and it will change colors!  
Conclusion  
In this tutorial, you've learned how to set up Thirdweb's engine and use it to connect an Unreal Engine game to Base. You've also learned how to use their platform to deploy and manager your contracts. Finally, you've learned how to build game elements to allow players to collect new NFTs and use them to personalize their game items.  
Tags:nftsmart contracts  
ON THIS PAGE  
Prerequisites  
Reviewing the Contract  
ERC-721 Tokens  
Unreal Engine  
Onchain Apps  
Getting Started with Unreal  
Setting up the Client and Server  
Client  
Setting Up the Game  
Client  
Server  
Adding the Color Changer  
Conclusion  
Deploying the Contract  
Adding the Contract to the Server  
Collecting the NFT from the Game  
Tinting the Car  
Fetching the NFT Colors  
Getting the Colors into the Game  
Changing the Color of the Car  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all

URL: https://docs.base.org/tutorials/verify-smart-contract-using-basescan  
  
Ecosystem  
Bridge  
Builders  
About  
Socials  
Home  
Get Started  
Docs  
Learn  
Search  
K  
CONNECT  
Verify a Smart Contract using Basescan API  
Basescan is a block explorer specifically designed for Base, offering developers a way to interact with and verify the smart contracts deployed on Base. Smart contract verification is a critical step in ensuring the transparency and security of onchain applications, as it allows others to review and validate the source code of deployed contracts. There are multiple ways to verify contracts and by the end of this tutorial you will learn how to verify a contract using the Solidity single file verification method using the Basescan API.  
Objectives  
By the end of this tutorial, you should be able to:  
Deploy a smart contract using Foundry  
Interact with the Basescan API to verify your deployed contract  
Obtain and configure a (free) Base RPC Node from Coinbase Developer Platform (CDP)  
Prerequisites  
Familiarity with smart contract development and the Solidity programming language  
Solidity is the primary programming language for writing smart contracts on Ethereum and Ethereum-compatible blockchains like Base. You should be comfortable with writing, compiling, and deploying basic smart contracts using Solidity. If not, check out Base Learn.  
Basic understanding of Foundry for Ethereum development  
Foundry is a fast and portable toolkit for Ethereum application development. It simplifies the process of deploying, testing, and interacting with smart contracts. This tutorial assumes you have experience using Foundry to compile and deploy smart contracts.  
Access to a Coinbase Developer Platform (CDP) account  
The Coinbase Developer Platform provides access to tools and services necessary for blockchain development, including RPC nodes for different networks. You'll need to sign up for a CDP account to obtain a Base RPC Node, which will be essential for deploying and interacting with your smart contracts on the Base blockchain.  
Node + Basic API requests  
Jump Right In  
For this turotial, you will deploy a simple contract that is included in the Foundry quickstart. To do so, ensure that you have Foundry installed.  
If you don't have Foundry install it:  
curl -L https://foundry.paradigm.xyz | bash  
Once installed, create a Foundry project:  
forge init verify\_contracts  
then change into the newly made directory:  
cd verify\_contracts  
You should have a folder structure similar to this:  
├── lib  
├── script  
├── src  
└── test  
The src folder will contain a Counter.sol file which will serve as the country you want to deploy.  
YOU WILL NEED ETH ON BASE TO DEPLOY  
You (the deployer wallet) will need some ETH in order to broadcast the transaction to the Base network. Fortunately, transactions are usually < 1 cent on Base mainnet.  
If using a [Coinbase Wallet] use the "Buy" button to onramp crypto from your Coinbase account.  
You will need a private key of the wallet that you want to deploy the smart contract from. Obtain it and store it as an env variable in your terminal.  
Once you have the private key to the wallet of your choice, open your terminal and store it in an environment variable:  
export PRIVATE\_KEY="<YOUR\_PRIVATE\_KEY>"  
To deploy our contract you will need a RPC URL to a Base node in order to broadcast our transactions to the network. CDP provides us with a free node for interacting with Base mainnet and testnet.  
Obtain a rpc url from the Node product and store the url as an environment variable similar to the private key in the previous step.  
Then store it as a environment variable in your terminal:  
export BASE\_RPC\_URL="your\_base\_rpc\_url"  
It's deployment time! Deploy the Counter.sol contract using forge create --rpc-url $BASE\_RPC\_URL --private-key $PRIVATE\_KEY src/Counter.sol:Counter  
Once deployed, it should return something like this:  
[⠊] Compiling...  
[⠢] Compiling 1 files with Solc 0.8.26  
[⠆] Solc 0.8.26 finished in 1.23s  
Compiler run successful!  
Deployer: 0xLo69e5523D33FBDbF133E81C91639e9d3C6cb369  
Deployed to: 0xEF5fe818Cb814E5c8277C5F12B57106B4EC3DdaA  
Transaction hash: 0xb191f9679a1fee253cf430ac09a6838f6806cfb2a250757fef407880f5546836  
Congrats! You've now deployed a contract to Base. The output of the deployment command contiains a contract address (e.g Deployed to: 0xEF5fe818Cb814E5c8277C5F12B57106B4EC3DdaA). Copy this address as you will need it in the next step.  
Verify the contract  
You will now interact with the Basescan API. For this, you need API Keys. Create an account using an email or login to Basescan.  
After signing in, navigate to your Basescan account then select API Keys on the left navigation bar.  
From the API Key page, click the blue "Add" button to create a new API Key then copy your API Key Token  
Save this to your clipboard for the next step.  
Create a .js file to create a function to that will call the Basescan's contract verification endpoint.  
In your terminal create a new file: touch verifyContractBasescan.js then open this file in your IDE of choice.  
At the top of the file create a variable that contains the Counter.sol that was created from your foundry project.  
const sourceCode = `pragma solidity ^0.8.13;  
contract Counter {  
 uint256 public number;  
 function setNumber(uint256 newNumber) public {  
 number = newNumber;  
}  
 function increment() public {  
 number++;  
 }  
}`;  
Create an async function to call the basescan api. Basescan offers a few endpoints to interact with their api with the base url being: https://api.basescan.org/api  
To verify a contract you will use the verifysourcecode route, with the contract module, and your apiKey as query parameters.  
UNSURE WHAT DATA TO INPUT?  
In every foundry project you will have a .json file that contains the conrtacts metadata and ABI. For this particular project, this information is located in the /verify\_contracts/out/Counter.sol/Counter.json  
Under the Metadata object you will find the compiler version under evmversion  
Putting everything together, our function will look like this:  
async function verifySourceCode() {  
 const url = 'https://api.basescan.org/api';  
  
 const params = new URLSearchParams({  
 module: 'contract',  
 action: 'verifysourcecode',  
 apikey: 'DK8M329VYXDSKTF633ABTK3SAEZ2U9P8FK', //remove hardcode  
 });  
  
 const data = new URLSearchParams({  
 chainId: '8453',  
 codeformat: 'solidity-single-file',  
 sourceCode: sourceCode,  
 contractaddress: '0x8aB096ea9886ACe363f81068d2439033F67F62E4',  
 contractname: 'Counter',  
 compilerversion: 'v0.8.26+commit.8a97fa7a',  
 optimizationUsed: 0,  
 evmversion: 'paris',  
 });  
 }  
Now add a try-catch block to send the request and log any errors to the console.  
Your final file should look something like this:  
const sourceCode = `pragma solidity ^0.8.13;  
  
contract Counter {  
 uint256 public number;  
  
 function setNumber(uint256 newNumber) public {  
 number = newNumber;  
 }  
  
 function increment() public {  
 number++;  
 }  
}`;  
  
async function verifySourceCode() {  
 const url = 'https://api.basescan.org/api';  
  
 const params = new URLSearchParams({  
 module: 'contract',  
 action: 'verifysourcecode',  
 apikey: 'YOUR\_API\_KEY',  
 });  
  
 const data = new URLSearchParams({  
 chainId: '8453',  
 codeformat: 'solidity-single-file',  
 sourceCode: sourceCode,  
 contractaddress: '0x8aB096ea9886ACe363f81068d2439033F67F62E4',  
 contractname: 'Counter',  
 compilerversion: 'v0.8.26+commit.8a97fa7a',  
 optimizationUsed: 0,  
 evmversion: 'paris',  
 });  
  
 try {  
 const response = await fetch(`${url}?${params}`, {  
 method: 'POST',  
 headers: {  
 'Content-Type': 'application/x-www-form-urlencoded',  
 },  
 body: data,  
 });  
  
 if (!response.ok) {  
 throw new Error(`HTTP error! status: ${response.status}`);  
 }  
  
 const result = await response.json();  
 console.log(result);  
 return result;  
 } catch (error) {  
 console.error('Error:', error);  
 throw error;  
 }  
}  
  
verifySourceCode().catch((error) => console.error('Unhandled error:', error));  
Save your file and then run node verifyContractBasescan.js in your terminal  
If successful, your terminal will output JSON text with three properties status, message and result like below:  
{  
 status: '1',  
 message: 'OK',  
 result: 'cqjzzvppgswqw5adq4v6iq4xkmf519pj1higvcxsdiwcvwxemd'  
}  
Result is the GUID and is a unique identifier for checking the status of your contracts verification.  
To verify the contract, let's create a curl request with the following paramters  
curl "https://api.basescan.org/api?module=contract&action=checkverifystatus&guid=cqjzzvppgswqw5adq4v6iq4xkmf519pj1higvcxsdiwcvwxemd&apikey=DK8M329VYXDSKTF633ABTK3SAEZ2U9P8FK"  
Run the command and you will see a that the contract should already be verified based on the result field  
{ "status": "0", "message": "NOTOK", "result": "Already Verified" }  
Conclusion  
Congratulations! You’ve successfully deployed and verified a smart contract using the Basescan API. Now, your users don’t have to rely solely on your word—they can verify the contract’s functionality through the code itself.  
Tags:smart contracts  
ON THIS PAGE  
Prerequisites  
Jump Right In  
Conclusion  
Verify the contract  
Conclusion  
All tutorials  
We use cookies and similar technologies on our websites to enhance and tailor your experience, analyze our traffic, and for security and marketing. You can choose not to allow some type of cookies by clicking Manage Settings. For more information see our Cookie Policy.  
Manage settings  
Accept all