Compile and Deploy on Secret testnet (best for Javascript devs)

Get started developing on Secret Network using the public testnet and secret.js

What is Secret.js?

In the previous section , we learned how to upload, execute, and query a Secret Network smart contract using SecretCLI and LocalSecret. Now we are going to repeat this process, but we will be uploading, executing, and querying our smart contracton a public testnet using Secret.js.

Secret.js is a JavaScript SDK for writing applications that interact with the Secret Network blockchain.

Key features include:

- Written in TypeScript and provided with type definitions.
- Provides simple abstractions over core data structures.
- Supports every possible message and transaction type.
- Exposes every possible query type.
- Handles input/output encryption/decryption for Secret Contracts.
- Works in Node.js, modern web browsers and React Native. * By the end of this tutorial, you will learn how to:
- Add the Secret Testnet to your keplr wallet (and fund your wallet with testnet tokens)
- Optimize and compile your Secret Network smart contract
- Upload and Instantiate your contract using Secret.js
- Execute and Query your contract using Secret.js
-

Let's get started!

Environment Configuration

To follow along with the guide, we will be usingnpm, git, make, rust, anddocker . Follow theSetting Up Your Environment guide here if you need any assistance.

Additionally, you will need to have the Secret Testnet configured with your keplr wallet and also fund it with testnet tokens.Learn how to configure and fund your keplr wallet here.

Generate your new counter contract

We will be working with a basic counter contract, which allows users to increment a counter variable by 1 and also reset the counter. If you've never worked with smart contracts written in Rust before that is perfectly fine.

The first thing you need to do is clone the counter contract from theSecret Network github repo . Secret Network developed this counter contract template so that developers have a simple structure to work with when developing new smart contracts, but we're going to use the contract exactly as it is for learning purposes.

Go to the folder in which you want to save your counter smart contract and run:

```

Copy cargo generate --git https://github.com/scrtlabs/secret-template.git --name my-counter-contract

```

When you run the above code, it will name your contract folder directory "my-counter-contract". But you can change the name by altering the text that follows the--name flag. Start by opening themy-counter-contract project folder in your text editor. If you navigate tomy-counter-contract/src you will seecontract.rs, msg.rs, lib.rs, and state.rs —these are the files that make up our counter smart contract. If you've never worked with a Rust smart contract before, perhaps take some time to familiarize yourself with the code, although in this tutorial we will not be going into detail discussing the contract logic.

1. In your root project folder, iemy-counter-contract
2. , create a new folder. For this tutorial I am choosing to call the foldernode
3. .
4. In yourmy-counter-contract/node
5. folder, create a new javascript file—I chose to name mineindex.js
6. .
7. Runnpm init -y
8. to create a package.json file.
9. Add"type" : "module"
10. to your package.json file.
11. Install secret.js and dotenv:npm i secretjs dotenv
12. Create a.env
13. file in yournode
14. folder, and add the variableMNEMONIC
15. along with your wallet address seed phrase, like so:
16.

```

Copy MNEMONIC=grant rice replace explain federal release fix clever romance raise often wild taxi quarter soccer fiber love must tape steak together observe swap guitar

```

Never use a wallet with actual funds when working with the testnet. If your seed phrase were pushed to github you could lose all of your funds. Create a new wallet that you use solely for working with the testnet. Congrats! You now have your environment c develop with secret.js.

Compile the Code

Since we are not making any changes to the contract code, we are going to compile it exactly as it is. To compile the code, runmake build in your terminal. This will take our Rust code and build a Web Assembly file that we can deploy to Secret Network. Basically, it just takes the smart contract that we've written and translates the code into a language that the blockchain can understand.

Linux/WSL/MacOS Windows Secret IDE ```

Copy make build

Copy env:RUSTFLAGS='-C link-arg=-s' cargo build --release --target wasm32-unknown-unknown cp ./target/wasm32-unknown-unknown/release/*.wasm ./contract.wasm

``` Runmake build from the terminal, or just GUI it up - This will create acontract.wasm andcontract.wasm.gz file in the root directory.

While we could upload this contract wasm file to the blockchain exactly as it is, instead we are going to follow best practices andoptimize the wasm file. This just means we are going to reduce the size of the file so that it costs less gas to upload, which is critical when you eventually upload contracts to mainnet.Make sure you have docker installed and then run the following code:

Optimize compiled wasm

```

Copy dockerrun--rm-v"(pwd)":/contract\ --mounttype=volume,source="(basename"(pwd)")_cache",target=/code/target\ --mounttype=volume,source=registry_cache,target=/usr/local/cargo/registry\ enigmampc/secret-contract-optimizer

```

You should now have an optimizedcontract.wasm.gz file in your root directory, which is ready to be uploaded to the blockchain! Also note that the optimizer should have removed thecontract.wasm file from your root directory

Uploading the Contract

Now that we have a working contract and an optimized wasm file, we can upload it to the blockchain and see it in action. This is calledstoring the contract code . We are using a public testnet environment, but the same commands apply no matter which network you want to use - local, public testnet, or mainnet.

Start by configuring yourindex.js file like so:

```

Copy import{ SecretNetworkClient,Wallet }from"secretjs"; import*asfsfrom"fs"; importdotenvfrom"dotenv"; dotenv.config();

constwallet=newWallet(process.env.MNEMONIC);

constcontract_wasm=fs.readFileSync("../contract.wasm.gz");

```

You now have secret.js imported, awallet variable that points to your wallet address, and acontract_wasm variable that points to the smart contract wasm file that we are going to upload to the testnet. The next step is to configure your Secret Network Client, which is used tobroadcast transactions, send queries and receive chain information.

Secret Network Client

Note thechainId and theurl that we are using. This chainId and url are for theSecret Network testnet . If you want to upload to LocalSecret or Mainnet instead, all you would need to do is swap out the chainId and url.A list of alternate API endpoints can be found here.

When connecting to a local node, you must indicate a port, which is1317 ```

Copy constsecretjs=newSecretNetworkClient({ chainId:"pulsar-3", url:"https://api.pulsar.scrttestnet.com", wallet:wallet, walletAddress:wallet.address, });

```

Nowconsole.log(secretjs) and runnode index.js in the terminal of yourmy-counter-contract/node folder to see that you have successfully connected to the Secret Network Client:

Secret Network Client configuration

Uploading with secret.js

To upload a compiled contract to Secret Network, you can use the following code:

```

Copy letupload_contract=async()=>{ lettx=awaitsecretjs.tx.compute.storeCode( { sender:wallet.address, wasm_byte_code:contract_wasm, source:"", builder:"", }, { gasLimit:4_000_000, } );

constcodeId=Number( tx.arrayLog.find((log)=>log.type==="message"&&log.key==="code_id") .value );

console.log("codeId: ",codeId);

constcontractCodeHash=( awaitsecretjs.query.compute.codeHashByCodeId({ code_id:codeId }) ).code_hash; console.log(Contract hash: {contractCodeHash});

};

upload_contract();

```

Runnode index.js in your terminal to call theupload_contract() function. Upon successful upload, acodeId and acontractCodeHash will be logged in your terminal:

```

Copy codeId: 19904 Contract hash: d350eb20b4bf93ce2a060168a1fe6faf58dafa84989bc22d3e83ac665f8c119f

```

Be sure to save the codeId and contractCodeHash as variables so you can access them in additional function calls.

Instantiating the Contract

In the previous step, we stored the contract code on the blockchain. To actually use it, we need to instantiate a new instance of it. Comment outupload_contract() and then addinstantiate_contract() .

Note that there is aninitMsg which containscount:0 . You can make the starting count whatever you'd like (as well as the contractlabel).

```

Copy letinstantiate_contract=async()=>{ // Create an instance of the Counter contract, providing a starting count constinitMsg={ count:0}; lettx=awaitsecretjs.tx.compute.instantiateContract( { code_id:codeId, sender:wallet.address, code_hash:contractCodeHash, init_msg:initMsg, label:"My Counter"+Math.ceil(Math.random()*10000), }, { gasLimit:400_000, } );

//Find the contract_address in the logs constcontractAddress=tx.arrayLog.find( (log)=>log.type==="message"&&log.key==="contract_address" ).value;

console.log(contractAddress); };

instantiate_contract();

```

Runnode index.js in your terminal to call theinstantiate_contract() function. Upon successful instantiation, a contractAddress will be logged in your terminal:

```

Copy secret1ez0nchvy6awpnnmzqqzvmqz62dcgke80pzaqc7

```

Be sure to save thecontractAddress as a variable so you can access it in additional function calls. Congrats ! You just fini and instantiating your first contract on a public Secret Network testnet! Now it's time to see it in action!

Executing and Querying the contract

The way you interact with contracts on a blockchain is by sendingcontract messages . A message contains the JSON description of a specific action that should be taken on behalf of the sender, and in most Rust smart contracts they are defined in themsg.rs file.

In ourmsg.rs file, there are two enums:ExecuteMsg , andQueryMsg . They are enums because each variant of them represents a different message which can be sent. For example, theExecuteMsg::Increment corresponds to thetry_increment message in ourcontract.rs file.

In the previous section, we compiled, uploaded and instantiated our counter smart contract. Now we are going to query the contract and also execute messages to update the contract state. Let's start by querying the counter contract we instantiated.

Query Message

AQuery Message is used to request information from a contract; unlikeexecute messages , query messages do not change contract state,

and are used just like database queries. You can think of queries as questions you can ask a smart contract.

Let's query our counter smart contract to return the current count. It should be 0, because that was the count we instantiated in the previous section. We query the count by calling the Query Messageget_count {} , which is defined in our msg.rs file. Comment outinstantiate_contract() and then addtry_query_count() .

```

Copy lettry_query_count=async()=>{ constmy_query=awaitsecretjs.query.compute.queryContract({ contract_address:contract_address, code_hash:contractCodeHash, query:{ get_count:{} }, });

console.log(my_query); };

try_query_count();

```

The query returns:

```

Copy {"count":"0"}

```

Great! Now that we have queried the contract's starting count, let's execute anincrement{} message to modify the contract state.

Execute Message

AnExecute Message is used for handling messages which modify contract state. They are used to perform contract actions.

Did you know? Messages aren't free! Each transaction costs a small fee, which represents how many resources were required to complete it. This cost is measured ingas units. The counter contract consists of two execute messages:increment{} , which increments the count by 1, andreset{} , which resets the count to anyi32 you want. The current count is 0, let's call the Execute Messageincrement{} to increase the contract count by 1.

```

Copy lettry_increment_count=async()=>{ lettx=awaitsecretjs.tx.compute.executeContract( { sender:wallet.address, contract_address:contract_address, code_hash:contractCodeHash,// optional but way faster msg:{ increment:{}, }, sentFunds:[],// optional }, { gasLimit:100_000, } ); console.log("incrementing..."); };

try_increment_count();

```

Nice work! Now we can query the contract once again to see if the contract state was successfully incremented by 1. The query returns:

```

Copy {"count":"1"}

```

Way to go! You have now successfully interacted with a Secret Network smart contract on the public testnet!

Next Steps

Congratulations! You completed the tutorial and successfully compiled, uploaded, deployed and executed a Secret Contract! The contract is the business logic that powers a blockchain application, but a full application contains other components as well. If you want to learn more about Secret Contracts, or explore what you just did more in depth, feel free to explore these awesome resources:

- Secret University counter contract breakdown
  - explains the counter contract in depth
- Millionaire's problem breakdown
  - explains how a Secret Contract works
- CosmWasm Documentation
  - everything you want to know about CosmWasm
- Secret.JS
  - Building a web UI for a Secret Contract

Last updated1 month ago On this page *What is Secret.js? * Environment Configuration * Generate your new counter contract * Compile the Code * Uploading the Contract * Instantiating the Contract * Executing and Querying the contract * Query Message * Execute Message * Next Steps

Was this helpful? Edit on GitHub Export as PDF