Building Smart Contracts On Stellar Network Using Sorboran

Soroban is the smart contracts platform on the Stellar network. These contracts are small programs written in the Rust language and compiled as WebAssembly (Wasm) for deployment.
note

For a comprehensive introduction to Stellar smart contracts, view the Smart Contract Learn Section.

Write your first smart contract on Stellar using the Getting Started Guide.

Rust on Stellar

Stellar smart contracts have several characteristics (such as resource limits, security considerations, and more) that force contracts to use only a narrow subset of the full Rust language and must use specialized libraries for most tasks.

Learn more in the Contract Rust Dialect section.

In particular, the Rust standard library and most third-party libraries (called crates) will not be available for direct off-the-shelf use in contracts due to the abovementioned constraints. Some crates can be adapted for use in contracts, and others may be incorporated into the host environment as host objects or functions.

note

Other languages may be supported in the future, but at this time, only Rust is supported.

Soroban Rust SDK

Contracts are developed using a software development kit (SDK). The Soroban Rust SDK consists of a Rust crate and a command-line (CLI) tool.

The SDK crate acts as a substitute for the Rust standard library — providing data structures and utility functions for contracts — as well as providing access to smart-contract-specific functionality from the contract environment, like cryptographic hashing and signature verification, access to on-chain persistent storage, and location and invocation of secondary contracts via stable identifiers.

The Soroban SDK CLI tool provides a developer-focused front-end for:

- Compiling
- Testing
- Inspecting
- Versioning
- Deploying

It also includes a complete implementation of the contract host environment that is identical to the one that runs on-chain, called local testing mode. With this capability, contracts can be run locally on a developer's workstation and can be tested and debugged directly with a local debugger within a

standard IDE, as well as a native test harness for fast-feedback unit testing and high-speed fuzzing or property testing.

Host environment

The host environment is a set of Rust crates compiled into the SDK CLI tool and stellar-core. It comprises a set of host objects and functions, an interface to on-chain storage and contract invocation, a resource-accounting and fee-charging system, and a Wasm interpreter.

Most contract developers will not frequently need to interact with the host environment directly — SDK functions wrap most of its facilities and provide richer and more ergonomic types and functions — but it is helpful to understand its structure to understand the conceptual model the SDK is presenting. Some parts of the host environment will likely be visible when testing or debugging contracts compiled natively on a local workstation.

Learn more in the Environment Concepts section.

Stellar smart contracts are small programs written in the Rust programming language.

To build and develop contracts you need the following prerequisites:

- A Rust toolchain
- An editor that supports Rust
- Stellar CLI

Install Rust

- macOS/Linux
- Windows
- Other

If you use macOS, Linux, or another Unix-like OS, the simplest method to install a Rust toolchain is to install `rustup`. Install `rustup` with the following command.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Then restart the terminal.

Install the target

You'll need a "target" for which your smart contract will be compiled. For Rust `v1.84.0` or higher, install the `wasm32v1-none` target.

```
rustup target add wasm32v1-none
```

You can learn more about the finer points of what this target brings to the table, in our page all about the Stellar Rust dialect. This page describes the subset of Rust functionality that is available to you within Stellar smart contract environment.

Configure an editor

Many editors have support for Rust. Visit the following link to find out how to configure your editor: https://www.rust-lang.org/tools

Here are the tools to you need to configure your editor:

1. Visual Studio Code as code editor (or another code editor that supports Rust)
2. Rust Analyzer for Rust language support
3. CodeLLDB for step-through-debugging

Install the Stellar CLI

The Stellar CLI can execute smart contracts on futurenet, testnet, mainnet, as well as in a local sandbox.

info

The latest stable release is v23.1.4.

Install

There are a few ways to install the latest release of Stellar CLI.

- macOS
- Linux
- Windows

Install with Homebrew (macOS, Linux):

```
brew install stellar-cli
```

Install with cargo from source:

```
cargo install --locked stellar-cli@23.1.4
```

info

Report issues and share feedback about the Stellar CLI here.

Documentation

The auto-generated comprehensive reference documentation is available here.

Autocompletion

You can use `stellar completion` to generate shell completion for different shells. You should absolutely try it out. It will feel like a super power!

- Bash
- ZSH

- fish
- PowerShell
- Elvish

To enable autocomplete on the current shell session:

```
source <(stellar completion --shell bash)
```

To enable autocomplete permanently, run the following command, then restart your terminal:

```
echo "source <(stellar completion --shell bash)" >> ~/.bashrc
```

Once you've set up your development environment, you're ready to create your first smart contract.

Create a New Project

Create a new project using the `init` command to create a `soroban-hello-world` project.

```
stellar contract init soroban-hello-world
```

The `init` command will create a Rust workspace project, using the recommended structure for including Soroban contracts. Let’s take a look at the project structure:

```
.
├── Cargo.lock
├── Cargo.toml
├── README.md
└── contracts
    ├── hello_world
    │   ├── Cargo.toml
    │   ├── Makefile
    │   ├── src
    │   │   ├── lib.rs
    │   │   └── test.rs
```

Cargo.toml

The `Cargo.toml` file at the root of the project is set up as Rust Workspace, which allows us to include multiple smart contracts in one project.

Rust Workspace

The `Cargo.toml` file sets the workspace’s members as all contents of the `contracts` directory and sets the workspace’s `soroban-sdk` dependency version including the `testutils` feature, which will allow test utilities to be generated for calling the contract in tests.

Cargo.toml

```
[workspace]
resolver = "2"
members = [
```

```
  "contracts/*",
]
[workspace.dependencies]
soroban-sdk = "22"
```

info

The `testutils` are automatically enabled inside Rust unit tests inside the same crate as your contract. If you write tests from another crate, you'll need to require the `testutils` feature for those tests and enable the `testutils` feature when running your tests with `cargo test --features testutils` to be able to use those test utilities.

`release` Profile

Configuring the `release` profile to optimize the contract build is critical. Soroban contracts have a maximum size of 64KB. Rust programs, even small ones, without these configurations almost always exceed this size.

The `Cargo.toml` file has the following release profile configured.

```
[profile.release]
opt-level = "z"
overflow-checks = true
debug = 0
strip = "symbols"
debug-assertions = false
panic = "abort"
codegen-units = 1
lto = true
```

`release-with-logs` Profile

Configuring a `release-with-logs` profile can be useful if you need to build a `.wasm` file that has logs enabled for printing debug logs when using the `stellar-cli`. Note that this is not necessary to access debug logs in tests or to use a step-through-debugger.

```
[profile.release-with-logs]
inherits = "release"
debug-assertions = true
```

See the logging example for more information about how to log.

Contracts Directory

The `contracts` directory is where Soroban contracts will live, each in their own directory. There is already a `hello_world` contract in there to get you started.

Contract-specific Cargo.toml file

Each contract should have its own `Cargo.toml` file, which relies on the top-level `Cargo.toml` that we just discussed.

This is where we can specify contract-specific package information.

contracts/hello_world/Cargo.toml

```
[package]
name = "hello-world"
version = "0.0.0"
edition = "2021"
publish = false
```

The `crate-type` is configured to `cdylib` which is required for building contracts.

```
[lib]
crate-type = ["cdylib"]
doctest = false
```

We also have included the soroban-sdk dependency, configured to use the version from the workspace Cargo.toml.

```
[dependencies]
soroban-sdk = { workspace = true }
[dev-dependencies]
soroban-sdk = { workspace = true, features = ["testutils"] }
```

Contract Source Code

Creating a Soroban contract involves writing Rust code in the project's `lib.rs` file.

All contracts should begin with `#![no_std]` to ensure that the Rust standard library is not included in the build. The Rust standard library is large and not well suited to being deployed into small programs like those deployed to blockchains.

```
#![no_std]
```

The contract imports the types and macros that it needs from the `soroban-sdk` crate.

```
use soroban_sdk::{contract, contractimpl, vec, Env, String, Vec};
```

Many of the types available in typical Rust programs, such as `std::vec::Vec`, are not available, as there is no allocator and no heap memory in Soroban contracts. The `soroban-sdk` provides a variety of types like `Vec`, `Map`, `Bytes`, `BytesN`, `Symbol`, that all utilize the Soroban environment's memory and native capabilities. Primitive values like `u128`, `i128`, `u64`, `i64`, `u32`, `i32`, and `bool` can also be used. Floats and floating point math are not supported.

Contract inputs must not be references.

The `#[contract]` attribute designates the `Contract` struct as the type to which contract functions are associated. This implies that the struct will have contract functions implemented for it.

```
#[contract]
pub struct Contract;
```

Contract functions are defined within an `impl` block for the struct, which is annotated with `#[contractimpl]`. It is important to note that contract functions should have names with a maximum length of 32 characters. Additionally, if a function is intended to be invoked from outside the contract, it should be marked with the `pub` visibility modifier. It is common for the first argument of a contract function to be of type `Env`, allowing access to a copy of the Soroban environment, which is typically necessary for various operations within the contract.

```
#[contractimpl]
impl Contract {
    pub fn hello(env: Env, to: String) -> Vec<String> {
        vec![&env, String::from_str(&env, "Hello"), to]
    }
}
```

Putting those pieces together a simple contract looks like this.

contracts/hello_world/src/lib.rs

```
#![no_std]
use soroban_sdk::{contract, contractimpl, vec, Env, String, Vec};
#[contract]
pub struct Contract;
#[contractimpl]
impl Contract {
    pub fn hello(env: Env, to: String) -> Vec<String> {
        vec![&env, String::from_str(&env, "Hello"), to]
    }
}
mod test;
```

Note the `mod test` line at the bottom, this will tell Rust to compile and run the test code, which we'll take a look at next.

Contract Unit Tests

Writing tests for Soroban contracts involves writing Rust code using the test facilities and toolchain that you'd use for testing any Rust code.

Given our `Contract`, a simple test will look like this.

- contracts/hello_world/src/lib.rs
- contracts/hello_world/src/test.rs

```
#![cfg(test)]
use super::*;
```

```
use soroban_sdk::{vec, Env, String};
#[test]
fn test() {
    let env = Env::default();
    let contract_id = env.register(Contract, ());
    let client = ContractClient::new(&env, &contract_id);
    let words = client.hello(&String::from_str(&env, "Dev"));
    assert_eq!(
        words,
        vec![
            &env,
            String::from_str(&env, "Hello"),
            String::from_str(&env, "Dev"),
        ]
    );
}
```

In any test the first thing that is always required is an `Env`, which is the Soroban environment that the contract will run inside of.

```
let env = Env::default();
```

The contract is registered with the environment using the contract type. Contracts can specify a fixed contract ID as the first argument, or provide `None` and one will be generated.

```
let contract_id = env.register(Contract, ());
```

All public functions within an `impl` block that is annotated with the `#[contractimpl]` attribute have a corresponding function generated in a generated client type. The client type will be named the same as the contract type with `Client` appended. For example, in our contract the contract type is `Contract`, and the client is named `ContractClient`.

```
let client = ContractClient::new(&env, &contract_id);
let words = client.hello(&String::from_str(&env, "Dev"));
```

The values returned by functions can be asserted on:

```
    assert_eq!(
        words,
        vec![
            &env,
            String::from_str(&env, "Hello"),
            String::from_str(&env, "Dev"),
        ]
    );
```

Run the Tests

Run `cargo test` and watch the unit test run. You should see the following output:

```
cargo test

running 1 test
test test::test ... ok
```

Try changing the values in the test to see how it works.

note

The first time you run the tests you may see output in the terminal of cargo compiling all the dependencies before running the tests.

Build the contract

To build a smart contract to deploy or run, use the `stellar contract build` command.

```
stellar contract build
```

tip

If you get an error like `can't find crate for 'core'`, it means you didn't install the wasm32 target during the setup step. You can do so by running `rustup target add wasm32v1-none` (use `rustup target add wasm32v1-none` for Rust versions **older than** `v1.85.0`).

This is a small wrapper around `cargo build` that sets the target to `wasm32v1-none` and the profile to `release`. You can think of it as a shortcut for the following command:

```
cargo build --target wasm32v1-none --release
```

A `.wasm` file will be outputted in the `target` directory, at `target/wasm32v1-none/release/hello_world.wasm`. The `.wasm` file is the built contract.

The `.wasm` file contains the logic of the contract, as well as the contract's specification / interface types, which can be imported into other contracts who wish to call it. This is the only artifact needed to deploy the contract, share the interface with others, or integration test against the contract.

Optimizing Builds

Use `stellar contract optimize` to further minimize the size of the `.wasm`:

```
stellar contract optimize --wasm target/wasm32v1-none/release/hello_world.wasm
```

This will optimize and output a new `hello_world.optimized.wasm` file in the same location as the input `.wasm`.

tip

Building optimized contracts is only necessary when deploying to a network with fees or when analyzing and profiling a contract to get it as small as possible. If you're just starting out writing a contract, these steps are not necessary. See Build for details on how to build for development.

Summary

In this section, we wrote a simple contract that can be deployed to a Soroban network.

Next we'll learn to deploy the HelloWorld contract to Stellar's Testnet network and interact with it over RPC using the CLI.

2. Deploy to Testnet

To recap what we've done so far, in Setup:

- we set up our local environment to write Rust smart contracts
- installed the stellar-cli
- configured the stellar-cli to communicate with the Stellar Testnet via RPC
- and configured an identity to sign transactions

In Hello World we created a `hello-world` project, and learned how to test and build the `HelloWorld` contract. Now we are ready to deploy that contract to Testnet, and interact with it.

Configure a Source Account

When you deploy a smart contract to a network, you need to specify a source account's keypair that will be used to sign the transactions.

Let's generate a keypair called `alice`. You can use any name you want, but it might be nice to have some named keys that you can use for testing, such as `alice`, `bob`, and `carol`. Notice that the keypair's account will be funded using Friendbot.

```
stellar keys generate alice --network testnet --fund
```

You can see the public key of `alice` with:

```
stellar keys address alice
```

You can see all of the keys you generated, along with where on your filesystem their information is stored, with:

```
stellar keys ls -l
```

Deploy

To deploy your HelloWorld contract, run the following command:

- macOS/Linux

- Windows (PowerShell)

```
stellar contract deploy \
  --wasm target/wasm32v1-none/release/hello_world.wasm \
  --source-account alice \
  --network testnet \
  --alias hello_world
```

This returns the contract's id, starting with a `C`. In this example, we're going to use `CACDYF3CYMJEJTIVFESQYZTN67GO2R5D5IUABTCUG3HXQSRXCSOROBAN`, so replace it with your actual contract id.

tip

We used the `--alias` flag in this deploy command which will create a `.stellar/contract-ids/hello_world.json` file that maps the alias `hello_world` to the contract id and network. This allows us to refer to this contract as its alias instead the contract id.

Interact

Using the code we wrote in Write a Contract and the resulting `.wasm` file we built in Build, run the following command to invoke the `hello` function.

info

In the background, the CLI is making RPC calls. For information on that checkout out the RPC reference page.

- macOS/Linux
- Windows (PowerShell)

```
stellar contract invoke \
  --id CACDYF3CYMJEJTIVFESQYZTN67GO2R5D5IUABTCUG3HXQSRXCSOROBAN \
  --source-account alice \
  --network testnet \
  -- \
  hello \
  --to RPC
```

The following output should appear.

```
["Hello", "RPC"]
```

info

The `--` double-dash is required!

This is a general CLI pattern used by other commands like cargo run. Everything after the `--`, sometimes called slop, is passed to a child process. In this case, `stellar contract invoke` builds an *implicit CLI* on-the-fly for the `hello` method in your contract. It can do this

because Soroban SDK embeds your contract's schema / interface types right in the `.wasm` file that gets deployed on-chain. You can also try:

```
stellar contract invoke ... -- --help
```

and

```
stellar contract invoke ... -- hello --help
```

Summary

In this lesson, we learned how to:

- deploy a contract to Testnet
- interact with a deployed contract

Next we'll add a new contract to this project, and see how our workspace can accommodate a multi-contract project. The new contract will show off a little bit of Soroban's storage capabilities.

Now that we've built a basic Hello World example contract, we'll write a simple contract that stores and retrieves data. This will help you see the basics of Soroban's storage system.
This is going to follow along with the increment example, which has a single function that increments an internal counter and returns the value. If you want to see a working example, try it in Devcontainers.

This tutorial assumes that you've already completed the previous steps in Getting Started: Setup, Hello World, and Deploy to Testnet.

Adding the increment contract

In addition to creating a new project, the `stellar contract init` command also allows us to initialize a new contract workspace within an existing project. In this example, we're going to initialize a new contract and use the `--name` flag to specify the name of our new contract, `increment`.

This command will not overwrite existing files unless we explicitly pass in the `--overwrite` flag. From within our `soroban-hello-world` directory, run:

```
stellar contract init . --name increment
```

This creates a new `contracts/increment` directory with placeholder code in `src/lib.rs` and `src/test.rs`, which we'll replace with our new increment contract and corresponding tests.

```
└── contracts
    ├── increment
        ├── Cargo.toml
        ├── Makefile
        └── src
            ├── lib.rs
            └── test.rs
```

We will go through the contract code in more detail below, but for now, replace the placeholder code in `contracts/increment/src/lib.rs` with the following.

```
#![no_std]
use soroban_sdk::{contract, contractimpl, log, symbol_short, Env, Symbol};
const COUNTER: Symbol = symbol_short!("COUNTER");
#[contract]
pub struct IncrementContract;
#[contractimpl]
impl IncrementContract {
    /// Increment increments an internal counter, and returns the value.
    pub fn increment(env: Env) -> u32 {
        let mut count: u32 = env.storage().instance().get(&COUNTER).unwrap_or(0);
        log!(&env, "count: {}", count);
        count += 1;
        env.storage().instance().set(&COUNTER, &count);
        env.storage().instance().extend_ttl(50, 100);
        count
    }
}
mod test;
```

Imports

This contract begins similarly to our Hello World contract, with an annotation to exclude the Rust standard library, and imports of the types and macros we need from the `soroban-sdk` crate.

contracts/increment/src/lib.rs

```
#![no_std]
use soroban_sdk::{contract, contractimpl, log, symbol_short, Env, Symbol};
```

Contract Data Keys

```
const COUNTER: Symbol = symbol_short!("COUNTER");
```

Contract data is associated with a key, which can be used at a later time to look up the value.

`Symbol` is a short (up to 32 characters long) string type with limited character space (only `a-zA-Z0-9_` characters are allowed). Identifiers like contract function names and contract data keys are represented by `Symbol`s.

The `symbol_short!()` macro is a convenient way to pre-compute short symbols up to 9 characters in length at compile time using `Symbol::short`. It generates a compile-time constant that adheres to the valid character set of letters (a-zA-Z), numbers (0-9), and underscores (_). If a symbol exceeds the 9-character limit, `Symbol::new` should be utilized for creating symbols at runtime.

Contract Data Access

```
let mut count: u32 = env
    .storage()
    .instance()
    .get(&COUNTER)
    .unwrap_or(0); // If no value set, assume 0.
```

The `Env.storage()` function is used to access and update contract data. The executing contract is the only contract that can query or modify contract data that it has stored. The data stored is viewable on ledger anywhere the ledger is viewable, but contracts executing within the Soroban environment are restricted to their own data.

The `get()` function gets the current value associated with the counter key.

If no value is currently stored, the value given to `unwrap_or(...)` is returned instead.

Values stored as contract data and retrieved are transmitted from the environment and expanded into the type specified. In this case a `u32`. If the value can be expanded, the type returned will be a `u32`. Otherwise, if a developer cast it to be some other type, a panic would occur at the unwrap.

```
env.storage()
    .instance()
    .set(&COUNTER, &count);
```

The `set()` function stores the new count value against the key, replacing the existing value.

Managing Contract Data TTLs with `extend_ttl()`

```
env.storage().instance().extend_ttl(100, 100);
```

All contract data has a Time To Live (TTL), measured in ledgers, that must be periodically extended. If an entry's TTL is not periodically extended, the entry will eventually become "archived." You can learn more about this in the State Archival document.

For now, it's worth knowing that there are three kinds of storage: `Persistent`, `Temporary`, and `Instance`. This contract only uses `Instance` storage: `env.storage().instance()`. Every time the counter is incremented, this storage's TTL gets extended by 100 ledgers, or about 500 seconds.

Build the contract

From inside `soroban-hello-world`, run:

```
stellar contract build
```

Check that it built:

```
ls target/wasm32v1-none/release/*.wasm
```

You should see both `hello_world.wasm` and `increment.wasm`.

Tests

Replace the placeholder code in `contracts/increment/src/test.rs` with the following increment test code.

contracts/increment/src/test.rs

```
#![cfg(test)]
use crate::{IncrementContract, IncrementContractClient};
use soroban_sdk::Env;
#[test]
fn test() {
    let env = Env::default();
    let contract_id = env.register(IncrementContract, ());
    let client = IncrementContractClient::new(&env, &contract_id);
    assert_eq!(client.increment(), 1);
    assert_eq!(client.increment(), 2);
    assert_eq!(client.increment(), 3);
}
```

This uses the same concepts described in the Hello World example.

Make sure it passes:

```
cargo test
```

You'll see that this runs tests for the whole workspace; both the Hello World contract and the new Increment contract.

If you want to see the output of the `log!` call, run the tests with `--nocapture`:

```
cargo test -- --nocapture
```

You should see the the diagnostic log events with the count data in the output:

```
running 1 test
[Diagnostic Event]
contract:CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD2KM,
    topics:[log],
    data:["count: {}", 1]
[Diagnostic Event]
contract:CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD2KM,
    topics:[log],
    data:["count: {}", 2]
[Diagnostic Event]
contract:CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD2KM,
    topics:[log],
    data:["count: {}", 3]
test test::test ... ok
```

Take it further

Can you figure out how to add `get_current_value` function to the contract? What about `decrement` or `reset` functions?

Summary

In this section, we added a new contract to this project, that made use of Soroban's storage capabilities to store and retrieve data. We also learned about the different kinds of storage and how to manage their TTLs.

Next we'll learn a bit more about deploying contracts to Soroban's Testnet network and interact with our incrementor contract using the CLI.

4. Deploy the Increment Contract

Two-step deployment

It's worth knowing that `deploy` is actually a two-step process.

1. **Upload the contract bytes to the network.** Soroban currently refers to this as *installing* the contract—from the perspective of the blockchain itself, this is a reasonable metaphor. This uploads the bytes of the contract to the network, indexing it by its hash. This contract code can now be referenced by multiple contracts, which means they would have the exact same *behavior* but separate storage state.
2. **Instantiate the contract.** This actually creates what you probably think of as a Smart Contract. It makes a new contract ID, and associates it with the contract bytes that were uploaded in the previous step.

You can run these two steps separately. Let's try it with the Increment contract:

info

If the contract has not been build yet, run the build command `stellar contract build` from the contract's root directory.

- macOS/Linux
- Windows (PowerShell)

```
stellar contract upload \
  --network testnet \
  --source-account alice \
  --wasm target/wasm32v1-none/release/soroban_increment_contract.wasm
```

This returns the hash of the Wasm bytes, like `6ddb28e0980f643bb97350f7e3bacb0ff1fe74d846c6d4f2c625e766210fbb5b`.

Now you can use `--wasm-hash` with `deploy` rather than `--wasm. Make sure to replace the example wasm hash with your own.

- macOS/Linux
- Windows (PowerShell)

```
stellar contract deploy \
  --wasm-hash 6ddb28e0980f643bb97350f7e3bacb0ff1fe74d846c6d4f2c625e766210fbb5b \
  --source-account alice \
  --network testnet \
  --alias increment
```

This command will return the contract id (e.g. `CACDYF3CYMJEJTIVFESQYZTN67GO2R5D5IUABTCUG3HXQSRXCSOROBAN`), and you can use it to invoke the contract like we did in previous examples.

- macOS/Linux
- Windows (PowerShell)

```
stellar contract invoke \
  --id CACDYF3CYMJEJTIVFESQYZTN67GO2R5D5IUABTCUG3HXQSRXCSOROBAN \
  --source-account alice \
  --network testnet \
  -- \
  increment
```

You should see the following output:

```
1
```

Run it a few more times to watch the count change.

Run your own network/node

Sometimes you'll need to run your own node:

- Production apps! Stellar maintains public test RPC nodes for Testnet and Futurenet, but not for Mainnet. Instead, you will need to run your own node, and point your app at that. If you want to use a software-as-a-service platform for this, various providers are available.
- When you need a network that differs from the version deployed to Testnet.

The RPC team maintains Docker containers that make this as straightforward as possible. See the RPC reference for details.

Up next

Ready to turn these deployed contracts into a simple web application? Head over to the Build a Dapp Frontend section.

5. Build a Hello World Frontend

In the previous examples, we invoked the contracts using the Stellar CLI, and in this last part of the guide we'll create a web app that interacts with the Hello World contract through TypeScript bindings.

info

This example shows one way of creating a binding between a contract and a frontend. For a more comprehensive guide to Dapp frontends, see the Build a Dapp Frontend documentation. For tooling that helps you start with smart contracts integrated with a working frontend environment quickly, jump to learn more about Scaffold Stellar.

Initialize a frontend toolchain from scratch

You can build a Stellar dapp with any frontend toolchain or integrate it into any existing full-stack app. For this tutorial, we're going to use Astro. Astro works with React, Vue, Svelte, any other UI library, or no UI library at all. In this tutorial, we're not using a UI library. The smart contract-specific parts of this tutorial will be similar no matter what frontend toolchain you use.

If you're new to frontend, don't worry. We won't go too deep. But it will be useful for you to see and experience the frontend development process used by smart contract apps. We'll cover the relevant bits of JavaScript and Astro, but teaching all of frontend development and Astro is beyond the scope of this tutorial.

Let's get started.

You're going to need Node.js v20 or greater. If you haven't yet, install it now.

We want to create an Astro project with the Hello World contract from the previous lessons integrated. To do this, we install the default Astro project:

```
npm create astro@latest
```

This project has the following directory structure.

```
extra-escape
├── astro.config.mjs
├── package-lock.json
├── package.json
├── packages
├── public
├── README.md
├── src
│   ├── assets
│   │   ├── astro.svg
│   │   └── background.svg
│   ├── components
│   │   └── Welcome.astro
│   ├── layouts
│   │   └── Layout.astro
```

```
|   └── pages
|       └── index.astro
└── tsconfig.json
```

Since we already deployed these contracts with aliases, we can reuse the generated contract ID files by copying them from the `soroban-hello-world/.stellar` directory into this project:

```
cp -R ../.stellar/ .stellar
```

Generate an NPM package for the Hello World contract

Before we open the new frontend files, let's generate an NPM package for the Hello World contract. This is our suggested way to interact with contracts from frontends. These generated libraries work with any JavaScript project (not a specific UI like React), and make it easy to work with some of the trickiest bits of smart contracts on Stellar, like encoding XDR.

This is going to use the CLI command `stellar contract bindings typescript`:

```
stellar contract bindings typescript \
  --network testnet \
  --contract-id hello_world \
  --output-dir packages/hello_world
```

tip

Notice that we were able to use the contract alias, `hello_world`, in place of the contract id!

The binding will be created in as a NPM package in the directory `packages/hello_world` as specified in the CLI command. We'll need to build the bindings package, since (in its initial state) the package is mostly TypeScript types and stubs for the various contract functions.

```
cd packages/hello_world
npm install
npm run build
cd ../..
```

We attempt to keep the code in these generated libraries readable, so go ahead and look around. Open up the new `packages/hello_world` directory in your editor. If you've built or contributed to Node projects, it will all look familiar. You'll see a `package.json` file, a `src` directory, a `tsconfig.json`, and even a README.

Call the contract from the frontend

Now let's open up `src/pages/index.astro` and use the binding to call the `hello` contract function with an argument.

The default Astro project consists of a page (`pages/index.astro`) and a welcome component (`component/Welcome.astro`), and we don't need any of that code. Replace the `pages/index.astro` code with this code (the welcome component will not be needed):

src/pages/index.astro

```
---
import * as Client from './packages/hello_world';
const contract = new Client.Client({
   ...Client.networks.testnet,
   rpcUrl: 'https://soroban-testnet.stellar.org:443'
});
const { result } = await contract.hello({to: "Devs!"});
const greeting = result.join(" ");
---
<h1>{greeting}</h1>
```

First we import the binding library, and then we need to define a contract client we can use for invoking the contract function we deployed to testnet in a previous step.

The `hello()` contract function is invoked synchronously with the argument `{to: "Devs!"}` and the expected response is an array consisting of "Hello" and "Devs!". We join the result array and the constant `greeting` should now hold the text `Hello Devs!`

Jumping down to the HTML section we now want to display the `greeting` text in the browser. Let's see it in action! Start the dev server:

```
npm run dev
```

And open localhost:4321 in your browser. You should see the greeting from the contract!

You can try updating the argument to `{ to: 'Stellar' }`. When you save the file, the page will automatically update.

info

When you start up the dev server with `npm run dev`, you will see similar output in your terminal as when you ran `npm run init`. This is because the `dev` script in package.json is set up to run `npm run init` and `astro dev`, so that you can ensure that your deployed contract and your generated NPM package are always in sync. If you want to just start the dev server without the initialize.js script, you can run `npm run astro dev`.

Using Scaffold Stellar to rapidly develop dapps

Scaffold Stellar is a developer toolkit for building decentralized applications and smart contracts on the Stellar blockchain, integrated with a frontend application.

Getting a running set of smart contracts and a frontend can be done in just a few short steps with Scaffold Stellar. This tutorial will show you how to make a new Scaffold Stellar project.

If you'd like to use existing smart contracts in a Scaffold Stellar project, all you need to do is copy them to the `contracts/` folder in your project root!

1. Install the Scaffold Stellar CLI

Since you already have the Stellar CLI installed, you have everything you need to install Scaffold Stellar:

```
cargo binstall stellar-scaffold-cli
```

We recommend using binstall for faster installs, or use `cargo install --locked stellar-scaffold-cli`. Scaffold Stellar is a plugin on the Stellar CLI, meaning you'll use `stellar scaffold` from the command line.

2. Initialize a fresh Scaffold Stellar project

```
stellar scaffold init <my-project>
```

Scaffold Stellar is a plugin on the Stellar CLI, so you'll run commands as `stellar scaffold <command>`. `init` initializes a fresh Stellar Scaffold project at the provided path.

3. Install Node dependencies & set up environment variables

Make sure you already have Node.js, and run:

```
cd <my-project> # make sure you're in your new project's directory
npm install # install frontend dependencies
cp .env.example .env # copies the example frontend environment variable file to
your own copy
```

4. Start your app in development mode!

```
npm start # or npm run dev
```

This command does two tasks concurrently: it runs `stellar scaffold watch --build-clients` which compiles your smart contracts, updating them as you change them, deploying them to your local Stellar chain (make sure Docker is running!), and configures all these settings via your `environments.toml` file in your root project folder.

The second task it runs is `vite`, a popular JavaScript build tool to compile and watch your React frontend for changes.

After a little compiling time, you'll be able to see your running app at localhost:5173!

Learn more about Scaffold Stellar.

This is a continuation of the Getting Started tutorial, where you should have deployed two smart contracts to the public network. In this section, we'll create a web app that interacts with the contracts via RPC calls.
Let's get started.

Initialize a frontend toolchain

You can build a Soroban app with any frontend toolchain or integrate it into any existing full-stack app. For this tutorial, we're going to use Astro. Astro works with React, Vue, Svelte, any other UI library, or no UI library at all. In this tutorial, we're not using a UI library. The Soroban-specific parts of this tutorial will be similar no matter what frontend toolchain you use.

If you're new to frontend, don't worry. We won't go too deep. But it will be useful for you to see and experience the frontend development process used by Soroban apps. We'll cover the relevant bits of JavaScript and Astro, but teaching all of frontend development and Astro is beyond the scope of this tutorial.

Let's get started.

You're going to need Node.js v18.14.1 or greater. If you haven't yet, install it now.

We want to create an Astro project with the contracts from the previous lesson. To do this, we can clone a template. You can find Soroban templates on GitHub by searching for repositories that start with "soroban-template-". For this tutorial, we'll use stellar/soroban-template-astro. We'll also use a tool called degit to clone the template without its git history. This will allow us to set it up as our own git project.

Since you have `node` and its package manager `npm` installed, you also have `npx`.

We're going to create a new project directory with this template to make things easier in this tutorial, so make sure you're no longer in your `soroban-hello-world` directory and then run:

```
npx degit stellar/soroban-template-astro first-soroban-app
cd first-soroban-app
git init
git add .
git commit -m "first commit: initialize from stellar/soroban-template-astro"
```

This project has the following directory structure, which we'll go over in more detail below.

```
├── contracts
│   ├── hello_world
│   └── increment
├── CONTRIBUTING.md
├── Cargo.toml
├── Cargo.lock
├── initialize.js
├── package-lock.json
├── package.json
├── packages
├── public
├── src
│   ├── components
│   │   └── Card.astro
│   ├── env.d.ts
```

```
│   ├── layouts
│   │   └── Layout.astro
│   └── pages
│       └── index.astro
└── tsconfig.json
```

The `contracts` are the same ones you walked through in the previous steps of the tutorial. Since we already deployed these contracts with aliases, we can reuse the generated contract ID files by copying them from the `soroban-hello-world/.stellar` directory into this project:

```
cp -R ../soroban-hello-world/.stellar/ .stellar
```

Generate an NPM package for the Hello World contract

Before we open the new frontend files, let's generate an NPM package for the Hello World contract. This is our suggested way to interact with contracts from frontends. These generated libraries work with any JavaScript project (not a specific UI like React), and make it easy to work with some of the trickiest bits of Soroban, like encoding XDR.

This is going to use the CLI command `stellar contract bindings typescript`:

```
stellar contract bindings typescript \
  --network testnet \
  --contract-id hello_world \
  --output-dir packages/hello_world
```

tip

Notice that we were able to use the contract alias, `hello_world`, in place of the contract id!

This project is set up as an NPM Workspace, and so the `hello_world` client library was generated in the `packages` directory at `packages/hello_world`.

We attempt to keep the code in these generated libraries readable, so go ahead and look around. Open up the new `packages/hello_world` directory in your editor. If you've built or contributed to Node projects, it will all look familiar. You'll see a `package.json` file, a `src` directory, a `tsconfig.json`, and even a README.

Generate an NPM package for the Increment contract

Though we can run `stellar contract bindings typescript` for each of our contracts individually, the soroban-template-astro project that we used as our template includes a very handy `initialize.js` script that will handle this for all of the contracts in our `contracts` directory.

In addition to generating the NPM packages, `initialize.js` will also:

- Generate and fund our Stellar account

- Build all of the contracts in the `contracts` dir
- Deploy our contracts
- Create handy contract clients for each contract

We have already taken care of the first three bullet points in earlier steps of this tutorial, so those tasks will be noops when we run `initialize.js`.

Configure initialize.js

We need to make sure that `initialize.js` has all of the environment variables it needs before we do anything else. Copy the `.env.example` file over to `.env`. The environment variables set in `.env` are used by the `initialize.js` script.

```
cp .env.example .env
```

Let's take a look at the contents of the `.env` file:

```
# Prefix with "PUBLIC_" to make available in Astro frontend files
PUBLIC_STELLAR_NETWORK_PASSPHRASE="Standalone Network ; February 2017"
PUBLIC_STELLAR_RPC_URL="http://localhost:8000/soroban/rpc"
STELLAR_ACCOUNT="me"
STELLAR_NETWORK="standalone"
```

This `.env` file defaults to connecting to a locally running network, but we want to configure our project to communicate with Testnet, since that is where we deployed our contracts. To do that, let's update the `.env` file to look like this:

```
# Prefix with "PUBLIC_" to make available in Astro frontend files
-PUBLIC_STELLAR_NETWORK_PASSPHRASE="Standalone Network ; February 2017"
+PUBLIC_STELLAR_NETWORK_PASSPHRASE="Test SDF Network ; September 2015"
-PUBLIC_STELLAR_RPC_URL="http://localhost:8000/soroban/rpc"
+PUBLIC_STELLAR_RPC_URL="https://soroban-testnet.stellar.org:443"
-STELLAR_ACCOUNT="me"
+STELLAR_ACCOUNT="alice"
-STELLAR_NETWORK="standalone"
+STELLAR_NETWORK="testnet"
```

info

This `.env` file is used in the `initialize.js` script. When using the CLI, we can still use the network configuration we set up in the Setup step, or by passing the `--rpc-url` and `--network-passphrase` flags.

Run `initialize.js`

First let's install the Javascript dependencies:

```
npm install
```

And then let's run `initialize.js`:

```
npm run init
```

As mentioned above, this script attempts to build and deploy our contracts, which we have already done. The script is smart enough to check if a step has already been taken care of, and is a no-op in that case, so it is safe to run more than once.

Call the contract from the frontend

Now let's open up `src/pages/index.astro` and take a look at how the frontend code integrates with the NPM package we created for our contracts.

Here we can see that we're importing our generated `helloWorld` client from `../contracts/hello_world`. We're then invoking the `hello` method and adding the result to the page.

src/pages/index.astro

```
---
import Layout from "../layouts/Layout.astro";
import Card from "../components/Card.astro";
import helloWorld from "../contracts/hello_world";
const { result } = await helloWorld.hello({ to: "you" });
const greeting = result.join(" ");
---
 ...
<h1>{greeting}</h1>
```

Let's see it in action! Start the dev server:

```
npm run dev
```

And open localhost:4321 in your browser. You should see the greeting from the contract!

You can try updating the `{ to: 'Soroban' }` argument. When you save the file, the page will automatically update.

info

When you start up the dev server with `npm run dev`, you will see similar output in your terminal as when you ran `npm run init`. This is because the `dev` script in package.json is set up to run `npm run init` and `astro dev`, so that you can ensure that your deployed contract and your generated NPM pacakage are always in sync. If you want to just start the dev server without the initialize.js script, you can run `npm run astro dev`.

What's happening here?

If you inspect the page (right-click, inspect) and refresh, you'll see a couple interesting things:

- The "Network" tab shows that there are no Fetch/XHR requests made. But RPC calls happen via Fetch/XHR! So how is the frontend calling the contract?
- There's no JavaScript on the page. But we just wrote some JavaScript! How is it working?

This is part of Astro's philosophy: the frontend should ship with as few assets as possible. Preferably zero JavaScript. When you put JavaScript in the frontmatter, Astro will run it at build time, and then replace anything in the `{...}` curly brackets with the output.

When using the development server with `npm run dev`, it runs the frontmatter code on the server, and injects the resulting values into the page on the client.

You can try building to see this more dramatically:

```
npm run build
```

Then check the `dist` folder. You'll see that it built an HTML and CSS file, but no JavaScript. And if you look at the HTML file, you'll see a static "Hello Soroban" in the `<h1>`.

During the build, Astro made a single call to your contract, then injected the static result into the page. This is great for contract methods that don't change, but probably won't work for most contract methods. Let's integrate with the `incrementor` contract to see how to handle interactive methods in Astro. -->

Call the incrementor contract from the frontend

While `hello` is a simple view-only/read method, `increment` changes on-chain state. This means that someone needs to sign the transaction. So we'll need to add transaction-signing capabilities to the frontend.

The way signing works in a browser is with a *wallet*. Wallets can be web apps, browser extensions, standalone apps, or even separate hardware devices.

Install Freighter Extension

Right now, the wallet that best supports Soroban is Freighter. It is available as a Firefox Add-on, as well as extensions for Chrome and Brave. Go ahead and install it now.

Once it's installed, open it up by clicking the extension icon. If this is your first time using Freighter, you will need to create a new wallet. Go through the prompts to create a password and save your recovery passphrase.

Go to Settings (the gear icon) → Preferences and toggle the switch to Enable Experimental Mode. Then go back to its home screen and select "Test Net" from the top-right dropdown. Finally, if it shows the message that your Stellar address is not funded, go ahead and click the "Fund with Friendbot" button.

Now you're all set up to use Freighter as a user, and you can add it to your app.

Add the StellarWalletsKit and set it up

Even though we're using Freighter to test our app, there are more wallets that support signing smart contract transactions. To make their integration easier, we are using the `StellarWalletsKit` library which allows us support all Stellar Wallets with a single library.

To install this kit we are going to include the next package:

```
npm install @creit.tech/stellar-wallets-kit
```

With the package installed, we are going to create a new simple file where our instantiated kit and simple state will be located. Create the file `src/stellar-wallets-kit.ts` and paste this:

src/stellar-wallets-kit.ts

```
import {
  allowAllModules,
  FREIGHTER_ID,
  StellarWalletsKit,
} from "@creit.tech/stellar-wallets-kit";
const SELECTED_WALLET_ID = "selectedWalletId";
function getSelectedWalletId() {
  return localStorage.getItem(SELECTED_WALLET_ID);
}
const kit = new StellarWalletsKit({
  modules: allowAllModules(),
  network: import.meta.env.PUBLIC_STELLAR_NETWORK_PASSPHRASE,
  // StellarWalletsKit forces you to specify a wallet, even if the user didn't
  // select one yet, so we default to Freighter.
  // We'll work around this later in `getPublicKey`.
  selectedWalletId: getSelectedWalletId() ?? FREIGHTER_ID,
});
export const signTransaction = kit.signTransaction.bind(kit);
export async function getPublicKey() {
  if (!getSelectedWalletId()) return null;
  const { address } = await kit.getAddress();
  return address;
}
export async function setWallet(walletId: string) {
  localStorage.setItem(SELECTED_WALLET_ID, walletId);
  kit.setWallet(walletId);
}
export async function disconnect(callback?: () => Promise<void>) {
  localStorage.removeItem(SELECTED_WALLET_ID);
  kit.disconnect();
  if (callback) await callback();
}
export async function connect(callback?: () => Promise<void>) {
  await kit.openModal({
    onWalletSelected: async (option) => {
      try {
```

```
        await setWallet(option.id);
        if (callback) await callback();
      } catch (e) {
        console.error(e);
      }
      return option.id;
    },
  });
}
```

In the code above, we instantiate the kit with desired settings and export it. We also wrap some kit functions and add custom functionality, such as augmenting the kit by allowing it to remember which wallet options was selected between page refreshes (that's the `localStorage` bit). The kit requires a `selectedWalletId` even before the user selects one, so we also work around this limitation, as the code comment explains. You can learn more about how the kit works in the StellarWalletsKit documentation

Now we're going to add a "Connect" button to the page which will open the kit's built-in modal, and prompt the user to use their preferred wallet. Once the user picks their preferred wallet and grants permission to accept requests from the website, we will fetch the public key and the "Connect" button will be replaced with a message saying, "Signed in as [their public key]".

Now let's add a new component to the `src/components` directory called `ConnectWallet.astro` with the following content:

src/components/ConnectWallet.astro

```
<div id="connect-wrap" class="wrap" aria-live="polite">
   
  <div class="ellipsis"></div>
  <button style="display:none" data-connect aria-controls="connect-wrap">
    Connect
  </button>
  <button style="display:none" data-disconnect aria-controls="connect-wrap">
    Disconnect
  </button>
</div>
<style>
  .wrap {
    text-align: center;
    display: flex;
    width: 18em;
    margin: auto;
    justify-content: center;
    line-height: 2.7rem;
    gap: 0.5rem;
  }
  .ellipsis {
    overflow: hidden;
    text-overflow: ellipsis;
```

```
    text-align: center;
    white-space: nowrap;
  }
</style>
<script>
  import { getPublicKey, connect, disconnect } from "../stellar-wallets-kit";
  const ellipsis = document.querySelector(
    "#connect-wrap .ellipsis",
  ) as HTMLElement;
  const connectButton = document.querySelector("[data-connect]") as
HTMLButtonElement;
  const disconnectButton = document.querySelector(
    "[data-disconnect]",
  ) as HTMLButtonElement;
  async function showDisconnected() {
    ellipsis.innerHTML = "";
    ellipsis.removeAttribute("title");
    connectButton.style.removeProperty("display");
    disconnectButton.style.display = "none";
  }
  async function showConnected() {
    const publicKey = await getPublicKey();
    if (publicKey) {
      ellipsis.innerHTML = `Signed in as ${publicKey}`;
      ellipsis.title = publicKey ?? "";
      connectButton.style.display = "none";
      disconnectButton.style.removeProperty("display");
    } else {
      showDisconnected();
    }
  }
  connectButton.addEventListener("click", async () => {
    await connect(showConnected);
  });
  disconnectButton.addEventListener("click", async () => {
    disconnect(showDisconnected);
  });
  if (await getPublicKey()) {
    showConnected();
  } else {
    showDisconnected();
  }
</script>
```

Some of this may look surprising. `<style>` and `<script>` tags in the middle of the page? Uncreative class names like `wrap`? `import` statements in a `<script>`? Top-level `await`? What's going on here?

Astro automatically scopes the styles within a component to that component, so there's no reason for us to come up with a clever names for our classes.

And all the `script` declarations get bundled together and included intelligently in the page. Even if you use the same component multiple times, the script will only be included once. And yes, you can use top-level `await`.

You can read more about this in Astro's page about client-side scripts.

The code itself here is pretty self-explanatory. We import `kit` from the file we created before. Then, when the user clicks on the sign-in button, we call the `connect` function we created in our `stellar-wallets-kit.ts` file above. This will launch the built-in StellarWalletsKit modal, which allows the user to pick from the wallet options we configured (we configured all of them, with `allowAllModules`). We pass our own `setLoggedIn` function as the callback, which will be called in the `onWalletSelected` function in `stellar-wallets-kit.ts`. We end by updating the UI, based on whether the user is currently connected or not.

Now we can import the component in the frontmatter of `pages/index.astro`:

pages/index.astro

```
 ---
 import Layout from '../layouts/Layout.astro';
 import Card from '../components/Card.astro';
 import helloWorld from "../contracts/hello_world";
+import ConnectWallet from '../components/ConnectWallet.astro'
 ...
```

And add it right below the `<h1>`:

pages/index.astro

```
 <h1>{greeting}</h1>
+<ConnectWallet />
```

If you're no longer running your dev server, go ahead and restart it:

```
npm run dev
```

Then open the page and click the "Connect" button. You should see Freighter pop up and ask you to sign in. Once you do, the button should be replaced with a message saying, "Signed in as [your public key]".

Now you're ready to sign the call to `increment`!

Call `increment`

Now we can import the `increment` contract client from `contracts/increment.ts` and start using it. We'll again create a new Astro component. Create a new file at `src/components/Counter.astro` with the following contents:

src/components/Counter.astro

```
<strong>Incrementor</strong><br />
Current value: <strong id="current-value" aria-live="polite">???</strong><br />
<br />
<button data-increment aria-controls="current-value">Increment</button>
<script>
  import { getPublicKey, signTransaction } from "../stellar-wallets-kit";
  import incrementor from "../contracts/increment";
  const button = document.querySelector(
    "[data-increment]",
  ) as HTMLButtonElement;
  const currentValue = document.querySelector("#current-value") as HTMLElement;
  button.addEventListener("click", async () => {
    const publicKey = await getPublicKey();
    if (!publicKey) {
      alert("Please connect your wallet first");
      return;
    } else {
      incrementor.options.publicKey = publicKey;
      incrementor.options.signTransaction = signTransaction;
    }
    button.disabled = true;
    button.classList.add("loading");
    currentValue.innerHTML =
      currentValue.innerHTML +
      '<span class="visually-hidden"> – updating…</span>';
    try {
      const tx = await incrementor.increment();
      const { result } = await tx.signAndSend();
      // Only use `innerHTML` with contract values you trust!
      // Blindly using values from an untrusted contract opens your users to script
injection attacks!
      currentValue.innerHTML = result.toString();
    } catch (e) {
      console.error(e);
    } finally {
      button.disabled = false;
      button.classList.remove("loading");
    }
  });
</script>
```

This should be somewhat familiar by now. We have a `script` that, thanks to Astro's build system, can `import` modules directly. We use `document.querySelector` to find the elements defined above. And we add a `click` handler to the button, which calls `increment` and updates the value on the page. It also sets the button to `disabled` and adds a `loading` class while the call is in progress to prevent the user from clicking it again and visually communicate that something is happening. For people using screen readers, the loading state is communicated with the visually-hidden span, which will be announced to them thanks to the `aria` tags we saw before.

The biggest difference from the call to `greeter.hello` is that this transaction gets executed in two steps. The initial call to `increment` constructs a Soroban transaction and then makes an RPC call to *simulate* it. For read-only calls like `hello`, this is all you need, so you can get the `result` right away. For write calls like `increment`, you then need to `signAndSend` before the transaction actually gets included in the ledger. You also need to make sure you set a valid `publicKey` and a `signTransaction` method.

info

Destructuring `{ result }`: If you're new to JavaScript, you may not know what's happening with those `const { result }` lines. This is using JavaScript's *destructuring* feature. If the thing on the right of the equals sign is an object, then you can use this pattern to quickly grab specific keys from that object and assign them to variables. You can also name the variable something else, if you like. For example, try changing the code above to:

```
const { result: newValue } = ...
```

Also, notice that you don't need to manually specify Freighter as the wallet in the call to `increment`. This may change in the future, but while Freighter is the only game in town, these generated libraries automatically use it. If you want to override this behavior, you can pass a `wallet` option; check the latest `Wallet` interface in [the template source] for details.

Now let's use this component. In `pages/index.astro`, first import it:

pages/index.astro

```
 ---
 import Layout from '../layouts/Layout.astro';
 import Card from '../components/Card.astro';
 import helloWorld from "../contracts/hello_world";
 import ConnectFreighter from '../components/ConnectFreighter.astro';
+import Counter from '../components/Counter.astro';
 ...
```

Then use it. Let's replace the contents of the `instructions` paragraph with it:

pages/index.astro

```
 <p class="instructions">
-  To get started, open the directory <code>src/pages</code> in your project.<br />
-  <strong>Code Challenge:</strong> Tweak the "Welcome to Astro" message above.
+  <Counter />
 </p>
```

Check the page; if you're still running your dev server, it should have already updated. Click the "Increment" button; you should see a Freighter confirmation. Confirm, and... the value updates! 🎉

There's obviously some functionality missing, though. For example, that `???` is a bummer. But our `increment` contract doesn't give us a way to query the current value without also updating it.

Before you try to update it, let's streamline the process around building, deploying, and generating clients for contracts.

Take it further

If you want to take it a bit further and make sure you understand all the pieces here, try the following:

- Make a `src/contracts` folder with a `greeter.ts` and an `incrementor.ts`. Move the `new Contract({ ... })` logic into those files. You may also want to extract the `rpcUrl` variable to a `src/contracts/utils.ts` file.
- Add a `get_value` method to the `increment` contract, and use it to display the current value in the `Counter` component. When you run `npm run dev`, the `initialize` script will run and update the contract and the generated client.
- Add a "Decrement" button to the `Counter` component.
- Deploy your frontend. You can do this quickly and for free with GitHub. If you get stuck installing stellar-cli and deploying contracts on GitHub, check out how we did this.
- Rather than using NPM scripts for everything, try using a more elegant script runner such as just. The existing npm `scripts` can then call `just`, such as `"setup": "just setup"`.
- Update the README to explain what this project is and how to use it to potential collaborators and employers 😉

Troubleshooting

Sometimes things go wrong. As a first step when troubleshooting, you may want to clone our tutorial repository and see if the problem happens there, too. If it happens there, too, then it may be a temporary problem with the Soroban network.

Here are some common issues and how to fix them.

Call to `hello` fails

Sometimes the call to `hello` can start failing. You can obviously stub out the call and define `result` some other way to troubleshoot.

One of the common problems here is that the contract becomes archived. To check if this is the problem, you can re-run `npm run init`.

If you're still having problems, join our Discord (link above) or open an issue in GitHub.

All contract calls start throwing `403` errors

This means that Testnet is down, and you probably just need to wait a while and try again.

Wrapping up

Some of the things we did in this section:

- We learned about Astro's no-JS-by-default approach
- We added Astro components and learned how their `script` and `style` tags work
- We saw how easy it is to interact with smart contracts from JavaScript by generating client libraries using `stellar contract bindings typescript`
- We learned about wallets and Freighter

At this point, you've seen a full end-to-end example of building a contract on Stellar! What's next? You choose! You can:

- See more complex example contracts in the Example Contracts section.
- Learn more about the internal architecture and design of Soroban.
- Learn how to find other templates other than stellar/soroban-template-astro, and how to build your own: Develop contract initialization frontend templates

Smart Wallets

Smart wallets on Stellar are a growing piece of Stellar's ecosystem. Lots of development is taking place, and this documentation will grow and progress as we collaborate on an audited smart wallet solution. In the meantime, below you'll find some conceptual grounding, as well as some examples and tooling, to get you started.

Overview

Let's look at the various pieces that are at play in Stellar smart wallets.

Smart wallets

In short, a smart wallet is a smart contract that can act as an account on the network. A smart wallet can hold value and interact with other smart contracts.

They are digital wallets that leverage smart contract composability to offer enhanced functionality and security for managing digital assets. Unlike traditional wallets, smart wallets integrate features such as multi-signature support, policy signers, programmable transactions, and enhanced user experience.

Smart wallets enable users to interact seamlessly with blockchain ecosystems, providing a user-friendly interface for sending, receiving, and storing digital assets. They can incorporate signers of many types. Among the most exciting are passkey signers, which implement WebAuthn standards to ensure secure, password-less authentication, reducing the risks associated with traditional seed phrases and private keys.

By using smart contracts, smart wallets can automate transactions and implement advanced security protocols. For example: time-locked transfers, spending limits, and fraud detection. These features

make smart wallets an ideal choice for users seeking a secure, convenient, and versatile solution for managing their digital assets in the evolving landscape of blockchain technology.

WebAuthn

WebAuthn (web authentication) is a web standard for secure, password-less authentication. It uses public-key cryptography to eliminate the reliance on shared secrets in the web2 world. In the context of a blockchain, using a WebAuthn signer enhances security and usability for dapps.

WebAuthn provides decentralized authentication (no central authority manages passwords), enhanced security (secret keys remain on the user's device), improved user experience (users don't have to worry about storing or remembering their secret keys), and broader interoperability (WebAuthn is already supported by major browsers and platforms).

Secp256r1

The secp256r1 signature scheme is an elliptic curve that is often used with the Elliptic Curve Digital Signature Algorithm (ECDSA), which is a widely used public key signature scheme.

Secp256r1 is a common signature algorithm used in WebAuthn, which is the standard behind the passkeys available on browsers, computers, and phones. Enabling on-chain secp256r1 verification allows developers to design contracts that incorporate passkeys to sign smart contract transactions and access accounts instead of using seed phrases or signing keys.

The launch of Protocol 21 enabled **native** support of secp256r1 verification in Stellar smart contracts on Mainnet as outlined in CAP-0051. This integration means that secp256r1 is built directly into the protocol and applications can both sign transactions and submit them to the network using secp256r1 without relying on a third party.

Passkeys

Passkeys are an implementation of the WebAuthn standard. The ability to use passkeys to sign transactions and access accounts removes the need for users to store or remember their secret keys and/or 12- to 24-word seed phrases, something that has been a massive barrier to entry for end-users entering the blockchain space. Secret keys and seed phrases can be overwhelming, hard to remember, entered incorrectly, and prone to security breaches.

Passkeys offer a faster, more secure method of identity authentication by using encrypted data stored on a device and performing user verification with hardware tokens (like YubiKeys), biometric data (like fingerprints or facial recognition), or other cryptographic methods.

Read more about passkeys in this introduction to passkeys blog.

How they work together

That's a lot of technical buzzwords. Taken together:

1. Secp256r1 provides the cryptographic foundation for key generation and digital signatures,

2. WebAuthn offers a standardized framework for passwordless authentication using public-key cryptography, and
3. Passkeys implement these technologies to provide a seamless and secure user experience.

Bake all of that together into a Stellar smart contract, and you have the foundation for the planet's best smart wallet implementation!

Build with smart wallets

If you're building or experimenting with smart wallets on Stellar, some essential resources are:

Passkey Kit

The Passkey Kit is a TypeScript SDK for creating and managing Stellar smart wallets, primarily (though not exclusively) with passkey signers. It's intended to be used in tandem with Launchtube for submitting signed transactions on-chain, however this is not a requirement. This is both a client and a server-side library. `PasskeyKit` on the client and `PasskeyServer` on the server.

The source code also includes a smart contract implementation of a proposed smart wallet standardized interface.

- Use the app
- View the code
- Watch the discussion

Launchtube transaction submission

This is an API service where developers can send their smart contract transactions to get them funded and submitted to the Stellar network without having to worry about fees, sequence numbers, classic G addresses, or other complexities.

note

Launchtube is a prototype service. While SDF maintains a Mainnet implementation, we cannot offer any guarantees or SLAs for it. SDF's Mainnet Launchtube service may not be suitable for mission-critical production services.

- View the code
- Get an access token by asking in the `#launchtube` channel on Discord. Or, generate your own testnet token.
- Farm and use the KALE asset to acquire Launchtube tokens here

SDP backend smart wallet

The Stellar Disbursement Platform team is also experimenting with smart wallet implementations. In a feature branch, they've implemented a smart wallet contract that can also be used.

- View the code

Example projects

Explore some of the current work being done with smart wallets, passkeys, policy signers, and more on Stellar! Play around with the various examples linked below, build your own passkey-powered projects, and share your findings in the `#passkeys` channel on Discord.

note

These examples are not created or maintained officially by the SDF but by dedicated community members both internal and external to the SDF.

Zafegard

A smart wallet that demonstrates the use of stateful policy signers.

- View the code
- Watch the demo video

Do Math

A smart wallet that demonstrates the use of multi-sig and policy signers to, surprisingly, do math without needing to input a passkey for every interaction.

- View the code
- Watch the demo video

Soroban by example

An app that uses passkeys to sign Stellar smart contract transactions.

- View the demo
- View the code
- Watch the demo video
- Watch the discussion
- Read the blog

Super Peach

A passkey-powered multi-signer abstract account contract example.

- View the demo
- View the code
- Watch the demo video

Ye Olde Guestbook

A passkey-powered internet guestbook from yesteryear built with smart contracts and frontend code.

- View the demo
- View the code
- Read the tutorial

Get involved

Dev Discord

Join the discussion, ask questions, and share your own work with smart wallets in the Stellar Developer Discord in the `#passkeys` channel.

SEP discussion

Join the SEP discussion on the WebAuthn smart wallet contract interface in GitHub.