Getting Started

Dive into smart contract development with our "Getting Started" tutorial.

Setup

Install and configure Rust to deploy smart contracts.

1. Hello World

Create your first smart contract in Rust.

2. Deploy to Testnet

Deploy a smart contract to a live test network.

3. Storing Data

Write a smart contract that stores and retrieves data.

📄 4. Deploy the Increment Contract

Deploy the Increment contract to Testnet.

📄 5. Create an App

Make a frontend web app that interacts with your smart contracts.

Setup

Soroban contracts are small programs written in the Rust programming language.

To build and develop contracts you need only a couple prerequisites:

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

Install Rust

Linux, macOS, or other Unix-like OS

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
```

Windows

On Windows, download and run rustup-init.exe. You can continue with the default settings by pressing Enter.

> TIP
>
> The soroban CLI uses emojis in its output. To properly render them on Windows, it is recommended to use the Windows Terminal. See how to install Windows Terminal on Microsoft Learn. If the CLI is used in the built

> in Windows Command Prompt or Windows PowerShell the CLI will function as expected but the emojis will appear as question marks.

If you're already using WSL, you can also follow the instructions for Linux.

Other

For other methods of installing Rust, see: https://www.rust-lang.org/tools/install

Install the target

Install the `wasm32-unknown-unknown` target.

```
rustup target add wasm32-unknown-unknown
```

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

A popular editor is Visual Studio Code:

- Visual Studio Code editor.
- Rust Analyzer for Rust language support.
- CodeLLDB for step-through-debugging.

Install the Soroban CLI

The Soroban CLI can execute Soroban contracts in the same environment the contract will execute on network, however in a local sandbox.

Install the latest released version of Soroban CLI using `cargo install`.

```
cargo install --locked soroban-cli
```

> INFO
>
> Report issues and share feedback about the Soroban CLI here.

Usage

Run the `soroban` command and you should see output like below.

```
soroban
```

```
$ soroban
Build, deploy, & interact with contracts; set identities to sign with;
configure networks; generate keys; and more.

Intro: https://soroban.stellar.org
CLI Reference: https://github.com/stellar/soroban-tools/tree/main/docs/
soroban-cli-full-docs.md

Usage: soroban [OPTIONS] <COMMAND>

Commands:
  completion  Print shell completion code for the specified shell
```

TIP

You can use `soroban completion` to generate shell completion for `bash`, `elvish`, `fish`, `powershell`, and `zsh`. You should absolutely try it out. It will feel like a super power!

To enable autocomplete in the current bash shell, run:

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

To enable autocomplete permanently in future bash shells, run:

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

Users of non-bash shells may need to adapt the above commands to suit their needs.

Configuring the CLI for Testnet

Soroban has a test network called Testnet that you can use to deploy and test your smart contracts. It's a live network, but it's not the same as the Stellar public network. It's a separate network that is used for development and testing, so you can't use it for production apps. But it's a great place to test your contracts before you deploy them to the public network.

To configure your CLI to interact with Testnet, run the following command:

macOS/Linux **Windows (PowerShell)**

```
soroban network add \
  --global testnet \
  --rpc-url https://soroban-testnet.stellar.org:443 \
  --network-passphrase "Test SDF Network ; September 2015"
```

```
soroban network add `
  --global testnet `
  --rpc-url https://soroban-testnet.stellar.org:443 `
  --network-passphrase "Test SDF Network ; September 2015"
```

Note the `--global` flag. This creates a file in your home folder's `~/.config/soroban/network/testnet.toml` with the settings you specified. This means that you can use the `--network testnet` flag in any Soroban CLI command to use this network from any directory or filepath on your system.

If you want project-specific network configurations, you can omit the `--global` flag, and the networks will be added to your working directory's `.soroban/network` folder instead.

Configure an Identity

When you deploy a smart contract to a network, you need to specify an identity that will be used to sign the transactions.

Let's configure an identity called `alice`. You can use any name you want, but it might be nice to have some named identities that you can use for testing, such as `alice`, `bob`, and `carol`.

```
soroban keys generate --global alice --network testnet
```

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

```
soroban keys address alice
```

Like the Network configs, the `--global` means that the identity gets stored in `~/.config/soroban/identity/alice.toml`. You can omit the `--global` flag to store the identity in your project's `.soroban/identity` folder instead.

By default, `soroban keys generate` will fund the account using Friendbot. To disable this behavior, append `--no-fund` to the command when running it.

1. Hello World

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

Create a New Project

Create a new project using the `init` command to create a `getting-started-tutorial` project.

```
soroban 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
        └── 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 Soroban 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 = "20.3.2"
```

> 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.

```toml
[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 `soroban-cli`. Note that this is not necessary to access debug logs in tests or to use a step-through-debugger.

```toml
[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

```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.

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

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

```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.

```rust
#![no_std]
```

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

```
use soroban_sdk::{contract, contractimpl, symbol_short, vec, Env, Symbol,
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 HelloContract;
```

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]
```

Putting those pieces together a simple contract looks like this.

contracts/hello_world/src/lib.rs

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

#[contract]
pub struct HelloContract;

#[contractimpl]
impl HelloContract {
    pub fn hello(env: Env, to: Symbol) -> Vec<Symbol> {
        vec![&env, symbol_short!("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 HelloContract, a simple test will look like this.

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

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

```rust
#![cfg(test)]

use super::*;
use soroban_sdk::{symbol_short, vec, Env};

#[test]
fn test() {
    let env = Env::default();
    let contract_id = env.register_contract(None, HelloContract);
    let client = HelloContractClient::new(&env, &contract_id);

    let words = client.hello(&symbol_short!("Dev"));
    assert_eq!(
        words,
        vec![&env, symbol_short!("Hello"), symbol_short!("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.

```rust
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.

```rust
let contract_id = env.register_contract(None, 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(&symbol_short!("Dev"));
```

The values returned by functions can be asserted on:

```
assert_eq!(
    words,
    vec![&env, symbol_short!("Hello"), symbol_short!("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 Soroban contract to deploy or run, use the `soroban contract build` command.

```
soroban contract build
```

This is a small wrapper around `cargo build` that sets the target to `wasm32-unknown-unknown` and the profile to `release`. You can think of it as a shortcut for the following command:

```
cargo build --target wasm32-unknown-unknown --release
```

A `.wasm` file will be outputted in the `target` directory, at `target/wasm32-unknown-unknown/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 `soroban contract optimize` to further minimize the size of the `.wasm`. First, re-install soroban-cli with the `opt` feature:

```
cargo install --locked soroban-cli --features opt
```

Then build an optimized `.wasm` file:

```
soroban contract optimize --wasm target/wasm32-unknown-unknown/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 soroban-cli
- configured the soroban-cli to communicate with the Soroban 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.

Deploy

To deploy your HelloWorld contract, run the following command:

macOS/Linux Windows (PowerShell)

```
soroban contract deploy \
  --wasm target/wasm32-unknown-unknown/release/hello_world.wasm \
  --source alice \
  --network testnet
```

```
soroban contract deploy `
  --wasm target/wasm32-unknown-unknown/release/hello_world.wasm `
  --source alice `
  --network testnet
```

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.

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)**

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

```
soroban contract invoke `
  --id CACDYF3CYMJEJTIVFESQYZTN67GO2R5D5IUABTCUG3HXQSRXCSOROBAN `
  --source 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, `soroban 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:
>
> ```
> soroban contract invoke ... -- --help
> ```
>
> and
>
> ```
> soroban 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.

3. Storing Data

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 GitPod.

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

The `soroban contract init` command allows us to initialize a new project with any of the example contracts from the soroban-examples repo, using the `--with-example` (or `-w`) flag.

It will not overwrite existing files, so we can also use this command to add a new contract to an existing project. Run the command again with a `--with-example` flag to add an `increment` contract to our project. From inside our `getting-started-tutorial` directory, run:

```
soroban contract init ./ --with-example increment
```

This will create a new `contracts/increment` directory with the following files:

```
└── contracts
```

The following code was added to `contracts/increment/src/lib.rs`. We'll go over it in more detail below.

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

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

#[contract]
pub struct IncrementorContract;

#[contractimpl]
impl IncrementorContract {
    /// Increment an internal counter; return the new value.
    pub fn increment(env: Env) -> u32 {
        let mut count: u32 =
env.storage().instance().get(&COUNTER).unwrap_or(0);

        count += 1;

        log!(&env, "count: {}", count);

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

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

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

Contract Data Keys

```rust
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

```rust
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 caused 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 `getting-started-tutorial`, run:

```
soroban contract build
```

Check that it built:

```
ls target/wasm32-unknown-unknown/release/*.wasm
```

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

Tests

The following test has been added to the `contracts/increment/src/test.rs` file.

contracts/incrementor/src/test.rs

```
use crate::{IncrementorContract, IncrementorContractClient};
use soroban_sdk::Env;

#[test]
fn increment() {
    let env = Env::default();
    let contract_id = env.register_contract(None, IncrementorContract);
    let client = IncrementorContractClient::new(&env, &contract_id);

    assert_eq!(client.increment(), 1);
```

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 output:

```
running 1 test
count: U32(0)
count: U32(1)
count: U32(2)
test test::incrementor ... 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:

macOS/Linux Windows (PowerShell)

```
soroban contract install \
  --network testnet \
  --source alice \
  --wasm target/wasm32-unknown-unknown/release/
soroban_increment_contract.wasm
```

```
soroban contract install `
```

This returns the hash of the Wasm bytes, like `6ddb28e0980f643bb97350f7e3bacb0ff1fe74d846c6d4f2c625e766210fbb5b`. Now you can use `--wasm-hash` with `deploy` rather than `--wasm`:

macOS/Linux **Windows (PowerShell)**

```
soroban contract deploy \
  --wasm-hash
6ddb28e0980f643bb97350f7e3bacb0ff1fe74d846c6d4f2c625e766210fbb5b \
  --source alice \
  --network testnet
```

```
soroban contract deploy `
  --wasm-hash
6ddb28e0980f643bb97350f7e3bacb0ff1fe74d846c6d4f2c625e766210fbb5b `
  --source alice `
  --network testnet
```

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)**

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

```
soroban contract invoke `
  --id CACDYF3CYMJEJTIVFESQYZTN67GO2R5D5IUABTCUG3HXQSRXCSOROBAN `
```

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

Up next, we'll use the deployed contracts to build a simple web app.

5. Create an App

With two smart contracts deployed to a public network, you can now create a web app that interacts with them 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 initialize our current project as an Astro project. To do this, we can again turn to the `soroban contract init` command, which has a `--frontend-template` flag that allows us to pass the url of a frontend template repository. As we learned in Storing Data, `soroban contract init` will not overwrite existing files, and is safe to use to add to an existing project.

From our `getting-started-tutorial` directory, run the following command to add the Astro template files.

```
soroban contract init ./ \
  --frontend-template https://github.com/stellar/soroban-astro-template
```

This will add the following to your project, which we'll go over in more detail below.

```
├── CONTRIBUTING.md
├── initialize.js
├── package-lock.json
├── package.json
├── packages
├── public
│   └── favicon.svg
├── src
│   ├── components
│   │   └── Card.astro
│   ├── env.d.ts
│   ├── layouts
│   │   └── Layout.astro
│   └── pages
│       └── index.astro
└── tsconfig.json
```

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 `soroban contract bindings typescript`:

```
soroban contract bindings typescript \
  --network testnet \
  --contract-id $(cat .soroban/contract-ids/hello_world.txt) \
  --output-dir packages/hello_world
```

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 `soroban contract bindings typescript` for each of our contracts individually, the soroban-astro-template 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_SOROBAN_NETWORK_PASSPHRASE="Standalone Network ; February 2017"
PUBLIC_SOROBAN_RPC_URL="http://localhost:8000/soroban/rpc"

SOROBAN_ACCOUNT="me"
SOROBAN_NETWORK="standalone"

# env vars that begin with PUBLIC_ will be available to the client
PUBLIC_SOROBAN_RPC_URL=$SOROBAN_RPC_URL
```

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_SOROBAN_NETWORK_PASSPHRASE="Standalone Network ; February 2017"
+PUBLIC_SOROBAN_NETWORK_PASSPHRASE="Test SDF Network ; September 2015"
-PUBLIC_SOROBAN_RPC_URL="http://localhost:8000/soroban/rpc"
+PUBLIC_SOROBAN_RPC_URL="https://soroban-testnet.stellar.org:443"

-SOROBAN_ACCOUNT="me"
+SOROBAN_ACCOUNT="alice"
-SOROBAN_NETWORK="standalone"
+SOROBAN_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 http://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 Freighter

We're going to add a "Connect" button to the page that opens Freighter and prompts the user to give your web page permission to use Freighter. Once they grant this permission, the "Connect" button will be replaced with a message saying, "Signed in as [their public key]".

First, add @stellar/freighter-api as a dependency:

```
npm install @stellar/freighter-api
```

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

src/components/ConnectFreighter.astro

```
<div id="freighter-wrap" class="wrap" aria-live="polite">
  <div class="ellipsis">
    <button data-connect aria-controls="freighter-wrap">Connect</button>
  </div>
</div>

<style>
  .wrap {
    text-align: center;
  }

  .ellipsis {
    line-height: 2.7rem;
    margin: auto;
    max-width: 12rem;
    overflow: hidden;
    text-overflow: ellipsis;
    text-align: center;
    white-space: nowrap;
  }
</style>

<script>
  import { isAllowed, setAllowed, getUserInfo } from '@stellar/freighter-
api';
```

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 a few methods from `@stellar/freighter-api` to check if the user is logged in. If they already are, then `isAllowed` returns `true`. If it's been more than a day since they've used the Freighter extension, then the `publicKey` will be blank, so we tell them to unlock Freighter and refresh the page. If `isAllowed` and the `publicKey` both look good, we replace the contents of the `div` with the signed-in message, replacing the button. Otherwise, we add a click handler to the button to prompt the user to connect Freighter with `setAllowed`. Once they do, we again replace the contents of the `div` with the signed-in message. The `aria` stuff ensures that screen readers will read the new contents when they're updated.

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 ConnectFreighter from '../components/ConnectFreighter.astro';
 ...
```

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

pages/index.astro

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

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 `soroban_increment_contract` 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 incrementor from "../contracts/soroban_increment_contract";
```

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.

> 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 soroban-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 Soroban contracts from JavaScript by generating client libraries using `soroban contract bindings typescript`
- We learned about wallets and Freighter

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

- See more complex example contracts in the Tutorials section.
- Learn more about the internal architecture and design of Soroban.
- Check out a more full-featured example app, which uses React rather than vanilla JavaScript and Next.js rather than Astro. This app also has a more complex setup & initialization process, with the option of using a locally-hosted RPC node.

Example Contracts

The Soroban team has put together a large collection of example contracts to demonstrate use of the Soroban smart contracts platform. For many of these example contracts, we've written an accompanying tutorial that will walk you through the example contract and describe a bit more about its design.

The examples listed below are provided in a sequential manner. The first listed example contracts create a solid foundation of concepts that will be required during the later examples. While you are absolutely free to choose, read, and use any of the example contracts you like, please keep in mind that the order you see is intentional.

Events - Publish events from a smart contract.

Custom Types - Define your own data structures in a smart contract.

Errors - Define and generate errors in a smart contract.

Logging - Debug a smart contract with logs.

Auth - Implement authentication and authorization.

Cross Contract Calls - Call a smart contract from another smart contract.

Deployer - Deploy and initialize a smart contract using another smart contract.

Allocator - Use the allocator feature to emulate heap memory in a smart contract.

Atomic Swap - Swap tokens atomically between authorized users.

Batched Atomic Swaps - Swap a token pair among groups of authorized users.

Timelock - Lockup some token to be claimed by another user under set conditions.

Single Offer Sale - Make a standing offer to sell a token in exchange for another token.

Liquidity Pool - Write a constant-product liquidity pool contract.

Tokens - Write a CAP-46-6 compliant token contract.

Custom Account - Implement an account contract supporting multisig and custom authorization policies.

Fuzz Testing - Increase confidence in a contract's correctness with fuzz testing.

Events

The events example demonstrates how to publish events from a contract. This example is an extension of the storing data example.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `events` directory, and use `cargo test`.

```
cd events
cargo test
```

You should see the output:

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

Code

events/src/lib.rs

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

        // Increment the count.
        count += 1;

        // Save the count.
        env.storage().instance().set(&COUNTER, &count);

        // Publish an event about the increment occuring.
        // The event has two topics:
        //   - The "COUNTER" symbol.
        //   - The "increment" symbol.
        // The event data is the count.
        env.events()
            .publish((COUNTER, symbol_short!("increment")), count);

        // Return the count to the caller.
        count
    }
}
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/events

How it Works

This example contract extends the increment example by publishing an event each time the counter is incremented.

Contract events let contracts emit information about what their contract is doing.

Contracts can publish events using the environments events publish function.

```
env.events().publish(topics, data);
```

Event Topics

An event may contain up to four topics.

Topics are conveniently defined using a tuple. In the sample code two topics of `Symbol` type are used.

```
env.events().publish((COUNTER, symbol_short!("increment")), ...);
```

> TIP
>
> The topics don't have to be made of the same type. You can mix different types as long as the total topic count stays below the limit.

Event Data

An event also contains a data object of any value or type including types defined by contracts using `#[contracttype]`. In the example the data is the `u32` count.

```
env.events().publish(..., count);
```

Publishing

Publishing an event is done by calling the `publish` function and giving it the topics and data. The function returns nothing on success, and panics on failure. Possible failure reasons can include malformed inputs (e.g. topic count exceeds limit) and running over the resource budget (TBD). Once successfully published, the new event will be available to applications consuming the events.

```
env.events().publish((COUNTER, symbol_short!("increment")), count);
```

> CAUTION
>
> Published events are discarded if a contract invocation fails due to a panic, budget exhaustion, or when the contract returns an error.

Tests

Open the `events/src/test.rs` file to follow along.

events/src/test.rs

```
#[test]
fn test() {
    let env = Env::default();
```

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

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

The contract is registered with the environment using the contract type.

```
let contract_id = env.register_contract(None, IncrementContract);
```

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 `IncrementContract`, and the client is named `IncrementContractClient`.

```
let client = IncrementContractClient::new(&env, &contract_id);
```

The example invokes the contract several times.

```
assert_eq!(client.increment(), 1);
```

The example asserts that the events were published.

```
assert_eq!(
    env.events().all(),
    vec![
        &env,
        (
            contract_id.clone(),
            (symbol_short!("COUNTER"), symbol_short!("increment")).into_val(&env),
            1u32.into_val(&env)
        ),
        // ...
    ]
);
```

Build the Contract

To build the contract, use the `soroban contract build` command.

```
soroban contract build
```

A `.wasm` file should be outputted in the `target` directory:

```
target/wasm32-unknown-unknown/release/soroban_events_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke contract functions in the using it.

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke \
    --wasm target/wasm32-unknown-unknown/release/soroban_events_contract.wasm \
    --id 1 \
    -- \
```

```
soroban contract invoke `
    --wasm target/wasm32-unknown-unknown/release/soroban_events_contract.wasm `
    --id 1 `
    -- `
    increment
```

The following output should occur using the code above.

```
1
#0: event:
{"ext":"v0","contractId":[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],"type":"contract","body":{"v0":{"topics":[{"symbol":[67,79,85,78,84,
```

A single event `#0` is outputted, which is the contract event the contract published. The event contains the two topics, each a `symbol` (displayed as bytes), and the data object containing the `u32`.

Custom Types

The custom types example demonstrates how to define your own data structures that can be stored on the ledger, or used as inputs and outputs to contract invocations. This example is an extension of the storing data example.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `custom_types` directory, and use `cargo test`.

```
cd custom_types
cargo test
```

You should see the output:

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

Code

custom_types/src/lib.rs

```rust
#[contracttype]
#[derive(Clone, Debug, Eq, PartialEq)]
pub struct State {
    pub count: u32,
    pub last_incr: u32,
}

const STATE: Symbol = symbol_short!("STATE");

#[contract]
pub struct IncrementContract;

#[contractimpl]
impl IncrementContract {
    /// Increment increments an internal counter, and returns the value.
    pub fn increment(env: Env, incr: u32) -> u32 {
        // Get the current count.
        let mut state = Self::get_state(env.clone());

        // Increment the count.
        state.count += incr;
        state.last_incr = incr;

        // Save the count.
        env.storage().instance().set(&STATE, &state);

        // Return the count to the caller.
        state.count
    }
    /// Return the current state.
    pub fn get_state(env: Env) -> State {
        env.storage().instance().get(&STATE).unwrap_or(State {
            count: 0,
            last_incr: 0,
        }) // If no value set, assume 0.
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/custom_types

How it Works

Custom types are defined using the `#[contracttype]` attribute on either a `struct` or an `enum`.

Open the `custom_types/src/lib.rs` file to follow along.

Custom Type: Struct

Structs are stored on ledger as a map of key-value pairs, where the key is up to a 32 character string representing the field name, and the value is the value encoded.

Field names must be no more than 32 characters.

```rust
#[contracttype]
#[derive(Clone, Debug, Eq, PartialEq)]
pub struct State {
    pub count: u32,
    pub last_incr: u32,
}
```

Custom Type: Enum

The example does not contain enums, but enums may also be contract types.

Enums containing unit and tuple variants are stored on ledger as a two element vector, where the first element is the name of the enum variant as a string up to 32 characters in length, and the value is the value if the variant has one.

Only unit variants and single value variants, like `A` and `B` below, are supported.

```
#[contracttype]
#[derive(Clone, Debug, Eq, PartialEq)]
pub enum Enum {
    A,
    B(...),
}
```

Enums containing integer values are stored on ledger as the `u32` value.

```
#[contracttype]
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
#[repr(u32)]
pub enum Enum {
    A = 1,
    B = 2,
}
```

Using Types in Functions

Types that have been annotated with `#[contracttype]` can be stored as contract data and retrieved later.

Types can also be used as inputs and outputs on contract functions.

```
pub fn increment(env: Env, incr: u32) -> u32 {
        let mut state = Self::get_state(env.clone());
        state.count += incr;
        state.last_incr = incr;
        env.storage().instance().set(&STATE, &state);
        state.count
}

pub fn get_state(env: Env) -> State {
    env.storage().instance().get(&STATE).unwrap_or(State {
```

Tests

Open the `custom_types/src/test.rs` file to follow along.

custom_types/src/test.rs

```rust
#[test]
fn test() {
    let env = Env::default();
    let contract_id = env.register_contract(None, IncrementContract);
    let client = IncrementContractClient::new(&env, &contract_id);

    assert_eq!(client.increment(&1), 1);
    assert_eq!(client.increment(&10), 11);
    assert_eq!(
        client.get_state(),
        State {
            count: 11,
            last_incr: 10
        }
    );
}
```

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

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

The contract is registered with the environment using the contract type.

```rust
let contract_id = env.register_contract(None, IncrementContract);
```

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 `IncrementContract`, and the client is named `IncrementContractClient`.

```
let client = IncrementContractClient::new(&env, &contract_id);
```

The test invokes the `increment` function on the registered contract that causes the `State` type to be stored and updated a couple times.

```
assert_eq!(client.increment(&1), 1);
assert_eq!(client.increment(&10), 11);
```

The test then invokes the `get_state` function to get the `State` value that was stored, and can assert on its values.

```
assert_eq!(
    client.get_state(),
    State {
        count: 11,
        last_incr: 10
    }
);
```

Build the Contract

To build the contract, use the `soroban contract build` command.

```
soroban contract build
```

A `.wasm` file should be outputted in the `target` directory:

```
target/wasm32-unknown-unknown/release/soroban_custom_types_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke contract functions in the Wasm using it.

macOS/Linux | Windows (PowerShell)

```
soroban contract invoke \
    --wasm target/wasm32-unknown-unknown/release/
soroban_custom_types_contract.wasm \
    --id 1 \
    -- \
    increment \
    --incr 5
```

```
soroban contract invoke `
    --wasm target/wasm32-unknown-unknown/release/
soroban_custom_types_contract.wasm `
    --id 1 `
    -- `
    increment `
    --incr 5
```

The following output should occur using the code above.

```
5
```

Run it a few more times with different increment amounts to watch the count change.

Use the `soroban` to inspect what the counter is after a few runs.

```
soroban contract read --id 1 --key STATE
```

```
STATE,"{""count"":25,""last_incr"":15}"
```

Errors

The errors example demonstrates how to define and generate errors in a contract that invokers of the contract can understand and handle. This example is an extension of the storing data example.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `errors` directory, and use `cargo test`.

```
cd errors
cargo test
```

You should see output that begins like this:

```
running 2 tests

count: U32(0)
count: U32(1)
```

Code

errors/src/lib.rs

```
#[contracterror]
#[derive(Copy, Clone, Debug, Eq, PartialEq, PartialOrd, Ord)]
#[repr(u32)]
pub enum Error {
    LimitReached = 1,
}

const COUNTER: Symbol = symbol_short!("COUNTER");
const MAX: u32 = 5;

#[contract]
pub struct IncrementContract;

#[contractimpl]
impl IncrementContract {
    /// Increment increments an internal counter, and returns the value.
Errors
    /// if the value is attempted to be incremented past 5.
    pub fn increment(env: Env) -> Result<u32, Error> {
        // Get the current count.
        let mut count: u32 =
env.storage().instance().get(&COUNTER).unwrap_or(0); // If no value set,
assume 0.
        log!(&env, "count: {}", count);

        // Increment the count.
        count += 1;

        // Check if the count exceeds the max.
        if count <= MAX {
            // Save the count.
            env.storage().instance().set(&COUNTER, &count);

            // Return the count to the caller.
            Ok(count)
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/errors

How it Works

Open the `errors/src/lib.rs` file to follow along.

Defining an Error

Contract errors are Rust u32 enums where every variant of the enum is assigned an integer. The `#[contracterror]` attribute is used to set the error up so it can be used in the return value of contract functions.

The enum has some constraints:

- It must have the `#[repr(u32)]` attribute.
- It must have the `#[derive(Copy)]` attribute.
- Every variant must have an explicit integer value assigned.

```
#[contracterror]
#[derive(Copy, Clone, Debug, Eq, PartialEq, PartialOrd, Ord)]
#[repr(u32)]
pub enum Error {
    LimitReached = 1,
}
```

Contract errors cannot be stored as contract data, and therefore cannot be used as types on fields of contract types.

> TIP
>
> If an error is returned from a function anything the function has done is rolled back. If ledger entries have been altered, or contract data stored, all

> those changes are reverted and will not be persisted.

Returning an Error

Errors can be returned from contract functions by returning `Result<_, E>`.

The increment function returns a `Result<u32, Error>`, which means it returns `Ok(u32)` in the successful case, and `Err(Error)` in the error case.

```
pub fn increment(env: Env) -> Result<u32, Error> {
    // ...
    if count <= MAX {
        // ...
        Ok(count)
    } else {
        // ...
        Err(Error::LimitReached)
    }
}
```

Panicking with an Error

Errors can also be panicked instead of being returned from the function.

The increment function could also be written as follows with a `u32` return value. The error can be passed to the environment using the `panic_with_error!` macro.

```
pub fn increment(env: Env) -> u32 {
    // ...
    if count <= MAX {
        // ...
        count
    } else {
        // ...
```

> ⚠ CAUTION
>
> Functions that do not return a `Result<_, E>` type do not include in their specification what the possible error values are. This makes it more difficult for other contracts and clients to integrate with the contract. However, this might be ideal if the errors are diagnostic and debugging, and not intended to be handled.

Tests

Open the `errors/src/test.rs` file to follow along.

errors/src/test.rs

```rust
#[test]
fn test() {
    let env = Env::default();
    let contract_id = env.register_contract(None, IncrementContract);
    let client = IncrementContractClient::new(&env, &contract_id);

    assert_eq!(client.try_increment(), Ok(Ok(1)));
    assert_eq!(client.try_increment(), Ok(Ok(2)));
    assert_eq!(client.try_increment(), Ok(Ok(3)));
    assert_eq!(client.try_increment(), Ok(Ok(4)));
    assert_eq!(client.try_increment(), Ok(Ok(5)));
    assert_eq!(client.try_increment(), Err(Ok(Error::LimitReached)));

    std::println!("{}", env.logs().all().join("\n"));
}

#[test]
#[should_panic(expected = "Status(ContractError(1))")]
#E3256B
fn test_panic() {
    let env = Env::default();
    let contract_id = env.register_contract(None, IncrementContract);
```

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

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

The contract is registered with the environment using the contract type.

```
let contract_id = env.register_contract(None, IncrementContract);
```

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 `IncrementContract`, and the client is named `IncrementContractClient`.

```
let client = IncrementContractClient::new(&env, &contract_id);
```

Two functions are generated for every contract function, one that returns a `Result<>`, and the other that does not handle errors and panicks if an error occurs.

`try_increment`

In the first test the `try_increment` function is called and returns `Result<Result<u32, _>, Result<Error, Status>>`.

```
assert_eq!(client.try_increment(), Ok(Ok(5)));
assert_eq!(client.try_increment(), Err(Ok(Error::LimitReached)));
```

- If the function call is successful, `Ok(Ok(u32))` is returned.

- If the function call is successful but returns a value that is not a `u32`, `Ok(Err(_))` is returned.
- If the function call is unsuccessful, `Err(Ok(Error))` is returned.
- If the function call is unsuccessful but returns an error code not in the `Error` enum, or returns a system error code, `Err(Err(Status))` is returned and the `Status` can be inspected.

`increment`

In the second test the `increment` function is called and returns `u32`. When the last call is made the function panicks.

```
assert_eq!(client.increment(), 5);
client.increment();
```

- If the function call is successful, `u32` is returned.
- If the function call is successful but returns a value that is not a `u32`, a panic occurs.
- If the function call is unsuccessful, a panic occurs.

Build the Contract

To build the contract, use the `soroban contract build` command.

```
soroban contract build
```

A `.wasm` file should be outputted in the `target` directory:

```
target/wasm32-unknown-unknown/release/soroban_errors_contract.wasm
```

Run the Contract

Let's deploy the contract to Testnet so we can run it. The value provided as `--source` was set up in our Getting Started guide; please change accordingly if you created a different identity.

macOS/Linux **Windows (PowerShell)**

```
soroban contract deploy \
  --wasm target/wasm32-unknown-unknown/release/soroban_errors_contract.wasm \
  --source alice \
  --network testnet
```

```
soroban contract deploy `
  --wasm target/wasm32-unknown-unknown/release/soroban_errors_contract.wasm `
  --source alice `
  --network testnet
```

The command above will output the contract id, which in our case is `CC3UMHVTIEH6GGDBW7MM72Q545HBDCXGU3GMIXP23PQVSBFKNZRWT37X`.

Now that we've deployed the contract, we can invoke it.

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke \
    --id CC3UMHVTIEH6GGDBW7MM72Q545HBDCXGU3GMIXP23PQVSBFKNZRWT37X \
    --network testnet \
    --source alice \
    -- \
    increment
```

```
soroban contract invoke `
    --id CC3UMHVTIEH6GGDBW7MM72Q545HBDCXGU3GMIXP23PQVSBFKNZRWT37X `
    --network testnet `
    --source alice `
    -- `
    increment
```

Run the command a few times and on the 6th invocation you should see an error like this:

```
...
error: transaction simulation failed: host invocation failed

Caused by:
    HostError: Error(Contract, #1)

    Event log (newest first):
       0: [Diagnostic Event] contract:<your contract id>, topics:[error,
Error(Contract, #1)], data:"escalating Ok(ScErrorType::Contract) frame-exit
to Err"
       1: [Diagnostic Event] topics:[fn_call,
Bytes(b7461eb3410fe31861b7d8cfea1de74e118ae6a6ccc45dfadbe15904aa6e6369),
increment], data:Void
...
```

To retrieve the current counter value, use the command `soroban contract read`.

macOS/Linux Windows (PowerShell)

```
soroban contract read \
  --id CC3UMHVTIEH6GGDBW7MM72Q545HBDCXGU3GMIXP23PQVSBFKNZRWT37X \
  --network testnet \
  --source alice \
  --durability persistent \
  --output json
```

```
soroban contract read `
  --id CC3UMHVTIEH6GGDBW7MM72Q545HBDCXGU3GMIXP23PQVSBFKNZRWT37X `
  --network testnet `
  --source alice `
  --durability persistent `
  --output json
```

Logging

The logging example demonstrates how to log for the purpose of debugging.

Open in Gitpod

Logs in contracts are only visible in tests, or when executing contracts using `soroban-cli`. Logs are only compiled into the contract if the `debug-assertions` Rust compiler option is enabled.

TIP

Logs are not a substitute for step-through debugging. Rust tests for Soroban can be step-through debugged in your Rust-enabled IDE. See testing for more details.

CAUTION

Logs are not accessible by dapps and other applications. See the events example for how to produce structured events.

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `logging` directory, and use `cargo test`.

```
cd logging
cargo test -- --nocapture
```

You should see the output:

```
running 1 test
Hello Symbol(Dev)
test test::test ... ok
```

Code

Cargo.toml

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

logging/src/lib.rs

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

#[contract]
pub struct Contract;

#[contractimpl]
impl Contract {
    pub fn hello(env: Env, value: Symbol) {
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/logging

How it Works

The `log!` macro logs a string. Any logs that occur during execution are outputted to stdout in `soroban-cli` and available for tests to assert on or print.

Logs are only outputted if the contract is built with the `debug-assertions` compiler option enabled. This makes them efficient to leave in code permanently since a regular `release` build will omit them.

Logs are only recorded in Soroban environments that have logging enabled. The only Soroban environments where logging is enabled is in Rust tests, and in the `soroban-cli`.

Open the files above to follow along.

`Cargo.toml` Profile

Logs are only outputted if the contract is built with the `debug-assertions` compiler option enabled.

The `test` profile that is activated when running `cargo test` has `debug-assertions` enabled, so when running tests logs are enabled by default.

A new `release-with-logs` profile is added to `Cargo.toml` that inherits from the `release` profile, and enables `debug-assertions`. It can be used to build a `.wasm` file that has logs enabled.

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

To build without logs use the `--release` or `--profile release` option.

To build with logs use the `--profile release-with-logs` option.

Using the `log!` Macro

The `log!` macro builds a string from the format string, and a list of arguments. Arguments are substituted wherever the `{}` value appears in the format string.

```
log!(&env, "Hello {}", value);
```

The above log will render as follows if `value` is a `Symbol` containing `"Dev"`.

```
Hello Symbol(Dev)
```

> **CAUTION**
>
> The values outputted are currently relatively limited. While primitive values like `u32`, `u64`, `bool`, and `Symbol`s will render clearly in the log output, `Bytes`, `Vec`, `Map`, and custom types will render only their handle number. Logging capabilities are in early development.

Tests

Open the `logging/src/test.rs` file to follow along.

logging/src/test.rs

```
extern crate std;
```

The `std` crate, which contains the Rust standard library, is imported so that the test can use the `std::vec!` and `std::println!` macros. Since contracts are required to use `#![no_std]`, tests in contracts must manually import `std` to use std functionality like printing to stdout.

```
extern crate std;
```

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

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

The contract is registered with the environment using the contract type.

```
let contract_id = env.register_contract(None, HelloContract);
```

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 `HelloContract`, and the client is named `HelloContractClient`.

```
let client = HelloContractClient::new(&env, &contract_id);
let words = client.hello(&symbol_short!("Dev"));
```

Logs are available in tests via the environment.

```
let logs = env.logs().all();
```

They can asserted on like any other value.

```
assert_eq!(logs, std::vec!["Hello Symbol(Dev)"]);
```

They can be printed to stdout.

```
std::println!("{}", logs.join("\n"));
```

Build the Contract

To build the contract, use the `soroban contract build` command.

Without Logs

To build the contract without logs, use the `--release` option.

```
soroban contract build
```

A `.wasm` file should be outputted in the `target` directory, in the `release` subdirectory:

```
target/wasm32-unknown-unknown/release/soroban_logging_contract.wasm
```

With Logs

To build the contract with logs, use the `--profile release-with-logs` option.

```
soroban contract build --profile release-with-logs
```

A `.wasm` file should be outputted in the `target` directory, in the `release-with-logs` subdirectory:

```
target/wasm32-unknown-unknown/release-with-logs/soroban_logging_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke contract functions in the using it. Specify the `-v` option to enable verbose logs.

macOS/Linux **Windows (PowerShell)**

```
soroban -v contract invoke \
    --wasm target/wasm32-unknown-unknown/release-with-logs/
soroban_logging_contract.wasm \
    --id 1 \
    -- \
    hello \
    --value friend
```

```
soroban -v contract invoke `
    --wasm target/wasm32-unknown-unknown/release-with-logs/
soroban_logging_contract.wasm `
    --id 1 `
    -- `
    hello `
    --value friend
```

The output should include the following line.

```
soroban_cli::log::event::contract_log: log="Hello Symbol(me)"
```

Auth

The auth example demonstrates how to implement authentication and authorization using the Soroban Host-managed auth framework.

This example is an extension of the storing data example.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `auth` directory, and use `cargo test`.

```
cd auth
cargo test
```

You should see the output:

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

Code

auth/src/lib.rs

```
#[contracttype]
pub enum DataKey {
    Counter(Address),
}

#[contract]
pub struct IncrementContract;

#[contractimpl]
impl IncrementContract {
    /// Increment increments a counter for the user, and returns the value.
    pub fn increment(env: Env, user: Address, value: u32) -> u32 {
        // Requires `user` to have authorized call of the `increment` of this
        // contract with all the arguments passed to `increment`, i.e. `user`
        // and `value`. This will panic if auth fails for any reason.
        // When this is called, Soroban host performs the necessary
        // authentication, manages replay prevention and enforces the user's
        // authorization policies.
        // The contracts normally shouldn't worry about these details and just
        // write code in generic fashion using `Address` and `require_auth` (or
        // `require_auth_for_args`).
        user.require_auth();

        // This call is equilvalent to the above:
        // user.require_auth_for_args((&user, value).into_val(&env));

        // The following has less arguments but is equivalent in authorization
        // scope to the above calls (the user address doesn't have to be
        // included in args as it's guaranteed to be authenticated).
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/auth

How it Works

The example contract stores a per-`Address` counter that can only be incremented by the owner of that `Address`.

Open the `auth/src/lib.rs` file or see the code above to follow along.

`Address`

```
#[contracttype]
pub enum DataKey {
    Counter(Address),
}
```

`Address` is a universal Soroban identifier that may represent a Stellar account, a contract or an 'account contract' (a contract that defines a custom authentication scheme and authorization policies). Contracts don't need to distinguish between these internal representations though. `Address` can be used any time some network identity needs to be represented, like to distinguish between counters for different users in this example.

> ENUM KEYS LIKE `DataKey` ARE USEFUL FOR ORGANIZING CONTRACT STORAGE.
>
> Different enum values create different key 'namespaces'.
>
> In the example the counter for each address is stored against `DataKey::Counter(Address)`. If the contract needs to start storing other types of data, it can do so by adding additional variants to the enum.

`require_auth`

```
impl IncrementContract {
    pub fn increment(env: Env, user: Address, value: u32) -> u32 {
        user.require_auth();
```

`require_auth` method can be called for any `Address`. Semantically `user.require_auth()` here means 'require `user` to have authorized calling `increment` function of the current `IncrementContract` instance with the current call arguments, i.e. the current `user` and `value` argument values'. In simpler terms, this ensures that the `user` has allowed incrementing their counter value and nobody else can increment it.

When using `require_auth` the contract implementation doesn't need to worry about the signatures, authentication, and replay prevention. All these features are implemented by the Soroban host and happen automatically as long as the `Address` type is used.

`Address` has another method called `require_auth_for_args`. It works in the same fashion as `require_auth`, but allows customizing the arguments that need to be authorized. Note though, this should be used with care to ensure that there is a deterministic mapping between the contract invocation arguments and the `require_auth_for_args` arguments.

The following two calls are functionally equivalent to `user.require_auth`:

```
// Completely equivalent
user.require_auth_for_args((&user, value).into_val(&env));
// The following has less arguments but is equivalent in authorization
// scope to the above call (the user address doesn't have to be
// included in args as it's guaranteed to be authenticated).
user.require_auth_for_args((value,).into_val(&env));
```

Tests

Open the `auth/src/test.rs` file to follow along.

auth/src/test.rs

```rust
fn test() {
    let env = Env::default();
    env.mock_all_auths();

    let contract_id = env.register_contract(None, IncrementContract);
    let client = IncrementContractClient::new(&env, &contract_id);

    let user_1 = Address::random(&env);
    let user_2 = Address::random(&env);

    assert_eq!(client.increment(&user_1, &5), 5);
    // Verify that the user indeed had to authorize a call of `increment` with
    // the expected arguments:
    assert_eq!(
        env.auths(),
        [(
            // Address for which auth is performed
            user_1.clone(),
            // Identifier of the called contract
            contract_id.clone(),
            // Name of the called function
            symbol_short!("increment"),
            // Arguments used to call `increment` (converted to the env-managed vector via `into_val`)
            (user_1.clone(), 5_u32).into_val(&env)
        )]
    );

    // Do more `increment` calls. It's not necessary to verify authorizations
    // for every one of them as we don't expect the auth logic to change from
    // call to call.
    assert_eq!(client.increment(&user_1, &2), 7);
    assert_eq!(client.increment(&user_2, &1), 1);
    assert_eq!(client.increment(&user_1, &3), 10);
    assert_eq!(client.increment(&user_2, &4), 5);
}
```

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

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

The test instructs the environment to mock all auths. All calls to `require_auth` or `require_auth_for_args` will succeed.

```rust
env.mock_all_auths();
```

The contract is registered with the environment using the contract type.

```rust
let contract_id = env.register_contract(None, IncrementContract);
```

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 `IncrementContract`, and the client is named `IncrementContractClient`.

```rust
let client = IncrementContractClient::new(&env, &contract_id);
```

Generate `Address`es for two users. Normally the exact value of the `Address` shouldn't matter for testing, so they're simply generated randomly.

```rust
let user_1 = Address::random(&env);
let user_2 = Address::random(&env);
```

Invoke `increment` function for `user_1`.

```rust
assert_eq!(client.increment(&user_1, &5), 5);
```

In order to verify that the `require_auth` call(s) have indeed happened, use `auths` function that returns a vector of tuples containing the authorizations from the most recent contract invocation.

```rust
assert_eq!(
    env.auths(),
    [(
        // Address for which auth is performed
        user_1.clone(),
        // Identifier of the called contract
        contract_id.clone(),
        // Name of the called function
        symbol_short!("increment"),
        // Arguments used to call `increment` (converted to the env-managed vector via `into_val`)
        (user_1.clone(), 5_u32).into_val(&env)
    )]
);
```

Invoke `increment` function several more times for both users. Notice, that the values are tracked separately for each users.

```rust
assert_eq!(client.increment(&user_1, &2), 7);
assert_eq!(client.increment(&user_2, &1), 1);
assert_eq!(client.increment(&user_1, &3), 10);
assert_eq!(client.increment(&user_2, &4), 5);
```

Build the Contract

To build the contract into a `.wasm` file, use the `soroban contract build` command.

```
soroban contract build
```

The `.wasm` file should be found in the `target` directory after building:

```
target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke functions on the contract.

But since we are dealing with authorization and signatures, we need to set up some identities to use for testing and get their public keys:

```
soroban keys generate acc1
soroban keys generate acc2
soroban keys address acc1
soroban keys address acc2
```

Example output with two public keys of identities:

```
GA6S566FD3EQDUNQ4IGSLXKW3TGVSTQW3TPHPGS7NWMCEIPBOKTNCSRU
GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B
```

Now the contract itself can be invoked. Notice the `--source` must be the identity name matching the address passed to the `--user` argument. This allows `soroban` tool to automatically sign the necessary payload for the invocation.

macOS/Linux Windows (PowerShell)

```
soroban contract invoke \
    --source acc1 \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
    -- \
    increment \
    --user GA6S566FD3EQDUNQ4IGSLXKW3TGVSTQW3TPHPGS7NWMCEIPBOKTNCSRU \
    --value 2
```

```
soroban contract invoke `
    --source acc1 `
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm `
    --id 1 `
    -- `
    increment `
    --user GA6S566FD3EQDUNQ4IGSLXKW3TGVSTQW3TPHPGS7NWMCEIPBOKTNCSRU `
    --value 2
```

Run a few more increments for both accounts.

macOS/Linux Windows (PowerShell)

```
soroban contract invoke \
    --source acc2 \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
    -- \
    increment \
    --user GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B \
    --value 5
```

```
soroban contract invoke \
    --source acc1 \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
    -- \
    increment \
    --user GA6S566FD3EQDUNQ4IGSLXKW3TGVSTQW3TPHPGS7NWMCEIPBOKTNCSRU \
    --value 3
```

```
soroban contract invoke \
    --source acc2 \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
    -- \
    increment \
    --user GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B \
    --value 10
```

```
soroban contract invoke \
    --source acc2 \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
    -- \
    increment \
    --user GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B \
    --value 5
```

```
soroban contract invoke \
    --source acc1 \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
```

```
soroban contract invoke \
    --source acc2 \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
    -- \
    increment \
    --user GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B \
    --value 10
```

View the data that has been stored against each user with `soroban contract read`.

```
soroban contract read --id 1
```

```
"[""Counter"",""GA6S566FD3EQDUNQ4IGSLXKW3TGVSTQW3TPHPGS7NWMCEIPBOKTNCSRU""]",5
"[""Counter"",""GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B""]",15
```

It is also possible to preview the authorization payload that is being signed by providing `--auth` flag to the invocation:

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke \
    --source acc2 \
    --auth \
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm \
    --id 1 \
    -- \
    increment \
    --user GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B \
    --value 123
```

```
soroban contract invoke `
    --source acc2 `
    --auth `
    --wasm target/wasm32-unknown-unknown/release/soroban_auth_contract.wasm `
    --id 1 `
    -- `
    increment `
    --user GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B `
    --value 123
```

```
Contract auth:
[{"address_with_nonce":null,"root_invocation":{"contract_id":"0000000000000000000000000000000000000000000000000000000000000001","function_name":"increment","ar
```

Further reading

Authorization documentation provides more details on how Soroban auth framework works.

Timelock and Single Offer examples demonstrate authorizing token operations on behalf of the user, which can be extended to any nested contract invocations.

Atomic Swap example demonstrates multi-party authorization where multiple users sign their parts of the contract invocation.

Custom Account example for demonstrates an account contract that defines a custom authentication scheme and user-defined authorization policies.

Cross Contract Calls

The cross contract call example demonstrates how to call a contract from another contract.

Open in Gitpod

> ⓘ INFO
>
> In this example there are two contracts that are compiled separately, deployed separately, and then tested together. There are a variety of ways to develop and test contracts with dependencies on other contracts, and the Soroban SDK and tooling is still building out the tools to support these workflows. Feedback appreciated here.

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `cross_contract/contract_b` directory, and use `cargo test`.

```
cd cross_contract/contract_b
```

You should see the output:

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

Code

cross_contract/contract_a/src/lib.rs

```rust
#[contract]
pub struct ContractA;

#[contractimpl]
impl ContractA {
    pub fn add(x: u32, y: u32) -> u32 {
        x.checked_add(y).expect("no overflow")
    }
}
```

cross_contract/contract_b/src/lib.rs

```rust
mod contract_a {
    soroban_sdk::contractimport!(
        file = "../contract_a/target/wasm32-unknown-unknown/release/
soroban_cross_contract_a_contract.wasm"
    );
}

#[contract]
pub struct ContractB;

#[contractimpl]
impl ContractB {
    pub fn add_with(env: Env, contract: Address, x: u32, y: u32) -> u32 {
        let client = contract_a::Client::new(&env, &contract);
        client.add(&x, &y)
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/cross_contract

How it Works

Cross contract calls are made by invoking another contract by its contract ID.

Contracts to invoke can be imported into your contract with the use of `contractimport!(file = "...")`. The import will code generate:

- A `ContractClient` type that can be used to invoke functions on the contract.
- Any types in the contract that were annotated with `#[contracttype]`.

> TIP
>
> The `contractimport!` macro will generate the types in the module it is used, so it's a good idea to use the macro inside a `mod { ... }` block, or inside its own file, so that the names of generated types don't collide with names of types in your own contract.

Open the files above to follow along.

Contract A: The Contract to be Called

The contract to be called is Contract A. It is a simple contract that accepts `x` and `y` parameters, adds them together and returns the result.

cross_contract/contract_a/src/lib.rs

```
#[contract]
pub struct ContractA;

#[contractimpl]
```

> TIP
>
> The contract uses the `checked_add` method to ensure that there is no overflow, and if there is overflow, panics rather than returning an overflowed value. Rust's primitive integer types all have checked operations available as functions with the prefix `checked_`.

Contract B: The Contract doing the Calling

The contract that does the calling is Contract B. It accepts a contract ID that it will call, as well as the same parameters to pass through. In many contracts the contract to call might have been stored as contract data and be retrieved, but in this simple example it is being passed in as a parameter each time.

The contract imports Contract A into the `contract_a` module.

The `contract_a::Client` is constructed pointing at the contract ID passed in.

The client is used to execute the `add` function with the `x` and `y` parameters on Contract A.

cross_contract_calls/src/a.rs

```
mod contract_a {
    soroban_sdk::contractimport!(
        file = "../contract_a/target/wasm32-unknown-unknown/release/
soroban_cross_contract_a_contract.wasm"
    );
}

#[contract]
pub struct ContractB;

#[contractimpl]
impl ContractB {
```

Tests

Open the `cross_contract/contract_b/src/test.rs` file to follow along.

cross_contract/contract_b/src/test.rs

```rust
#[test]
fn test() {
    let env = Env::default();

    // Register contract A using the imported Wasm.
    let contract_a_id = env.register_contract_wasm(None, contract_a::Wasm);

    // Register contract B defined in this crate.
    let contract_b_id = env.register_contract(None, ContractB);

    // Create a client for calling contract B.
    let client = ContractBClient::new(&env, &contract_b_id);

    // Invoke contract B via its client. Contract B will invoke contract A.
    let sum = client.add_with(&contract_a_id, &5, &7);
    assert_eq!(sum, 12);
}
```

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

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

Contract A is registered with the environment using the imported Wasm.

```rust
let contract_a_id = env.register_contract_wasm(None, contract_a::Wasm);
```

Contract B is registered with the environment using the contract type.

```
let contract_b_id = env.register_contract(None, ContractB);
```

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 `ContractB`, and the client is named `ContractBClient`. The client can be constructed and used in the same way that client generated for Contract A can be.

```
let client = ContractBClient::new(&env, &contract_b_id);
```

The client is used to invoke the `add_with` function on Contract B. Contract B will invoke Contract A, and the result will be returned.

```
let sum = client.add_with(&contract_a_id, &5, &7);
```

The test asserts that the result that is returned is as we expect.

```
assert_eq!(sum, 12);
```

Build the Contracts

To build the contract into a `.wasm` file, use the `soroban contract build` command. Both `contract_call/contract_a` and `contract_call/contract_b` must be built, with `contract_a` being built first.

```
soroban contract build
```

Both `.wasm` files should be found in both contract `target` directories after building both contracts:

```
target/wasm32-unknown-unknown/release/soroban_cross_contract_a_contract.wasm
```

```
target/wasm32-unknown-unknown/release/soroban_cross_contract_b_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke contract functions. Both contracts must be deployed.

macOS/Linux **Windows (PowerShell)**

```
soroban contract deploy \
    --wasm target/wasm32-unknown-unknown/release/
soroban_cross_contract_a_contract.wasm \
    --id a
```

```
soroban contract deploy \
    --wasm target/wasm32-unknown-unknown/release/
soroban_cross_contract_b_contract.wasm \
    --id b
```

```
soroban contract deploy `
    --wasm target/wasm32-unknown-unknown/release/
soroban_cross_contract_a_contract.wasm `
    --id a
```

```
soroban contract deploy `
    --wasm target/wasm32-unknown-unknown/release/
soroban_cross_contract_b_contract.wasm `
    --id b
```

Invoke Contract B's `add_with` function, passing in values for `x` and `y` (e.g. as `5` and `7`), and then pass in the contract ID of Contract A.

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke \
    --id b \
    -- \
    add_with \
    --contract_id a \
    --x 5 \
    --y 7
```

```
soroban contract invoke `
    --id b `
    -- `
    add_with `
    --contract_id a `
    --x 5 `
    --y 7
```

The following output should occur using the code above.

```
12
```

Contract B's `add_with` function invoked Contract A's `add` function to do the addition.

Deployer

The deployer example demonstrates how to deploy contracts using a contract.

Here we deploy a contract on behalf of any address and initialize it atomically.

Open in Gitpod

> INFO
>
> In this example there are two contracts that are compiled separately, and the tests deploy one with the other.

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `deployer/deployer` directory, and use `cargo test`.

```
cd deployer/deployer
cargo test
```

You should see the output:

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

Code

deployer/deployer/src/lib.rs

```
#[contract]
pub struct Deployer;

#[contractimpl]
impl Deployer {
    /// Deploy the contract Wasm and after deployment invoke the init
function
    /// of the contract with the given arguments.
    ///
    /// This has to be authorized by `deployer` (unless the `Deployer`
instance
    /// itself is used as deployer). This way the whole operation is atomic
    /// and it's not possible to frontrun the contract initialization.
    ///
    /// Returns the contract address and result of the init function.
    pub fn deploy(
        env: Env,
        deployer: Address,
        wasm_hash: BytesN<32>,
        salt: BytesN<32>,
        init_fn: Symbol,
        init_args: Vec<Val>,
    ) -> (Address, Val) {
        // Skip authorization if deployer is the current contract.
        if deployer != env.current_contract_address() {
            deployer.require_auth();
        }

        // Deploy the contract using the uploaded Wasm with given hash.
        let deployed_address = env
            .deployer()
            .with_address(deployer, salt)
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/deployer

How it Works

Contracts can deploy other contracts using the SDK `deployer()` method.

The contract address of the deployed contract is deterministic and is derived from the address of the deployer. The deployment also has to be authorized by the deployer.

Open the `deployer/deployer/src/lib.rs` file to follow along.

Contract Wasm Upload

Before deploying the new contract instances, the Wasm code needs to be uploaded on-chain. Then it can be used to deploy an arbitrary number of contract instances. The upload should typically happen outside of the deployer contract, as it needs to happen just once. However, it is possible to use `env.deployer().upload_contract_wasm()` function to upload Wasm from a contract as well.

See the tests for an example of uploading the contract code programmatically. For the actual on-chain installation see the general deployment tutorial.

Authorization

> INFO
>
> This section can be skipped for factory contracts that deploy another contract from their own address (`deployer == env.current_contract_address()``).

> INFO
>
> For introduction to Soroban authorization see the auth tutorial.

We start with verifying authorization of the `deployer`, unless its the current contract (at which point the authorization is implied).

```
if deployer != env.current_contract_address() {
    deployer.require_auth();
}
```

While `deployer().with_address()` performs authorization as well, we want to make sure that `deployer` has also authorized the whole operation, as besides deployment it also performs atomic contract initialization. If we didn't require deployer authorization here, then it would be possible to frontrun the deployment operation performed by `deployer` and initialize it differently, thus breaking the promise of atomic initialization.

See more details on the actual authorization payloads in tests.

`deployer()`

The `deployer()` SDK function comes with a few deployment-related utilities. Here we use the most generic deployer kind, `with_address(deployer_address, salt)`.

```
let deployed_address = env
    .deployer()
    .with_address(deployer, salt)
    .deploy(wasm_hash);
```

`with_address()` accepts the `deployer` address and salt. Both are used to derive the address of the deployed contract deterministically. It is not possible to re-deploy an already existing contract.

`deploy()` function performs the actual deployment using the provided `wasm_hash`. The implementation of the new contract is defined by the Wasm file uploaded under `wasm_hash`.

> TIP
>
> Only the `wasm_hash` itself is stored per contract ID thus saving the ledger space and fees.

When only deploying the contract on behalf of the current contract, i.e. when `deployer` address is always `env.current_contract_address()` it is possible to use `deployer().with_current_contract(salt)` function for brevity.

Initialization

The contract can be called immediately after deployment, which is useful for initialization.

```
let res: Val = env.invoke_contract(&deployed_address, &init_fn, init_args);
```

`invoke_contract` can call any defined contract function with any arguments. We pass the actual function to call and the arguments from `deploy` inputs. The result can be any value, depending on the `init_fn`'s return value.

If the initialization fails, then the whole `deploy` call falls and thus the contract won't be deployed. This behavior is required for the atomic initialization guarantee as well.

The contract returns the deployed contract's address and the result of executing the initialization function.

```
(deployed_address, res)
```

Tests

Open the `deployer/deployer/src/test.rs` file to follow along.

Import the test contract Wasm to be deployed.

```
// The contract that will be deployed by the deployer contract.
mod contract {
    soroban_sdk::contractimport!(
        file =
            "../contract/target/wasm32-unknown-unknown/release/
soroban_deployer_test_contract.wasm"
    );
}
```

That contract contains the following code that exports two functions: initialization function that takes a value and a getter function for the stored initialized value.

deployer/contract/src/lib.rs

```
#[contract]
pub struct Contract;

const KEY: Symbol = symbol_short!("value");

#[contractimpl]
impl Contract {
    pub fn init(env: Env, value: u32) {
        env.storage().instance().set(&KEY, &value);
    }
    pub fn value(env: Env) -> u32 {
        env.storage().instance().get(&KEY).unwrap()
    }
}
```

This test contract will be used when testing the deployer. The deployer contract will deploys the test contract and invoke its `init` function.

There are two tests: deployment from the current contract without authorization and deployment from an arbitrary address with authorization. Besides authorization, these tests are very similar.

Curent contract deployer

In the first test we deploy contract from the `Deployer` contract instance itself.

```
#[test]
fn test_deploy_from_contract() {
    let env = Env::default();
    let deployer_client = DeployerClient::new(&env,
&env.register_contract(None, Deployer));

    // Upload the Wasm to be deployed from the deployer contract.
    // This can also be called from within a contract if needed.
    let wasm_hash = env.deployer().upload_contract_wasm(contract::WASM);

    // Deploy contract using deployer, and include an init function to call.
    let salt = BytesN::from_array(&env, &[0; 32]);
    let init_fn = symbol_short!("init");
    let init_fn_args: Vec<Val> = (5u32,).into_val(&env);
    let (contract_id, init_result) = deployer_client.deploy(
        &deployer_client.address,
        &wasm_hash,
        &salt,
        &init_fn,
        &init_fn_args,
    );

    assert!(init_result.is_void());
    // No authorizations needed - the contract acts as a factory.
    assert_eq!(env.auths(), vec![]);

    // Invoke contract to check that it is initialized.
    let client = contract::Client::new(&env, &contract_id);
    let sum = client.value();
```

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

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

Register the deployer contract with the environment and create a client to for it.

```
let deployer_client = DeployerClient::new(&env, &env.register_contract(None,
Deployer));
```

Upload the code of the test contract that we have imported above via `contractimport!` and get the hash of the uploaded Wasm code.

```
let wasm_hash = env.deployer().upload_contract_wasm(contract::WASM);
```

The client is used to invoke the `deploy` function. The contract will deploy the test contract using the hash of its Wasm code, call the `init` function, and pass in a single `5u32` argument. The expected return value of `init` function is just `void` (i.e. no value).

```
let salt = BytesN::from_array(&env, &[0; 32]);
let init_fn = symbol_short!("init");
let init_fn_args: Vec<Val> = (5u32,).into_val(&env);
let (contract_id, init_result) = deployer_client.deploy(
    &deployer_client.address,
    &wasm_hash,
    &salt,
    &init_fn,
    &init_fn_args,
);
```

The test checks that the test contract was deployed by using its client to invoke it and get back the value set during initialization.

```rust
let client = contract::Client::new(&env, &contract_id);
let sum = client.value();
assert_eq!(sum, 5);
```

External deployer

The second test is very similar to the first one.

```rust
#[test]
fn test_deploy_from_address() {
    let env = Env::default();
    let deployer_client = DeployerClient::new(&env,
&env.register_contract(None, Deployer));

    // Upload the Wasm to be deployed from the deployer contract.
    // This can also be called from within a contract if needed.
    let wasm_hash = env.deployer().upload_contract_wasm(contract::WASM);

    // Define a deployer address that needs to authorize the deployment.
    let deployer = Address::random(&env);

    // Deploy contract using deployer, and include an init function to call.
    let salt = BytesN::from_array(&env, &[0; 32]);
    let init_fn = symbol_short!("init");
    let init_fn_args: Vec<Val> = (5u32,).into_val(&env);
    env.mock_all_auths();
    let (contract_id, init_result) =
        deployer_client.deploy(&deployer, &wasm_hash, &salt, &init_fn,
&init_fn_args);

    assert!(init_result.is_void());

    let expected_auth = AuthorizedInvocation {
        // Top-level authorized function is `deploy` with all the arguments.
        function: AuthorizedFunction::Contract((
            deployer_client.address,
            symbol_short!("deploy"),
            (
                deployer.clone(),
                wasm_hash.clone(),
                salt,
```

The main difference is that the contract is deployed on behalf of the arbitrary address.

```
// Define a deployer address that needs to authorize the deployment.
let deployer = Address::random(&env);
```

Before invoking the contract we need to enable mock authorization in order to get the recorded authorization payload that we can verify.

```
env.mock_all_auths();
let (contract_id, init_result) =
        deployer_client.deploy(&deployer, &wasm_hash, &salt, &init_fn,
&init_fn_args);
```

The expected authorization tree for the `deployer` looks as follows.

```
let expected_auth = AuthorizedInvocation {
    // Top-level authorized function is `deploy` with all the arguments.
    function: AuthorizedFunction::Contract((
        deployer_client.address,
        symbol_short!("deploy"),
        (
            deployer.clone(),
            wasm_hash.clone(),
            salt,
            init_fn,
            init_fn_args,
        )
            .into_val(&env),
    )),
    // From `deploy` function the 'create contract' host function has to be
    // authorized.
    sub_invocations: vec![AuthorizedInvocation {
        function:
AuthorizedFunction::CreateContractHostFn(CreateContractArgs {
            contract_id_preimage:
ContractIdPreimage::Address(ContractIdPreimageFromAddress {
```

At the top level we have the `deploy` function itself with all the arguments that we've passed to it. From the `deploy` function the `CreateContractHostFn` has to be authorized. This is the authorization payload that has to be authorized by any deployer in any context. It contains the deployer address, salt and executable.

This authorization tree proves that the deployment and initialization are authorized atomically: actual deployment happens within the context of `deploy` and all of salt, executable, and initialization arguments are authorized together (i.e. there is one signature to authorizes this exact combination).

Then we make sure that deployer has authorized the expected tree and that expected value has been stored.

```
assert_eq!(env.auths(), vec![(deployer, expected_auth)]);

let client = contract::Client::new(&env, &contract_id);
let sum = client.value();
assert_eq!(sum, 5);
```

Build the Contracts

To build the contract into a `.wasm` file, use the `soroban contract build` command. Build both the deployer contract and the test contract.

```
soroban contract build
```

Both `.wasm` files should be found in both contract `target` directories after building both contracts:

```
target/wasm32-unknown-unknown/release/soroban_deployer_contract.wasm
```

```
target/wasm32-unknown-unknown/release/soroban_deployer_test_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke the contract function to deploy the test contract.

Before deploying the test contract with the deployer, install the test contract Wasm using the `install` command. The `install` command will print out the hash derived from the Wasm file (it's not just the hash of the Wasm file itself though) which should be used by the deployer.

```
soroban contract install --wasm contract/target/wasm32-unknown-unknown/
release/soroban_deployer_test_contract.wasm
```

The command prints out the hash as hex. It will look something like `7792a624b562b3d9414792f5fb5d72f53b9838fef2ed9a901471253970bc3b15`.

We also need to deploy the `Deployer` contract:

```
soroban contract deploy --wasm deployer/target/wasm32-unknown-unknown/
release/soroban_deployer_contract.wasm --id 1
```

This will return the deployer address: `CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD2KM`.

Then the deployer contract may be invoked with the Wasm hash value above.

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke --id 1 -- deploy \
    --deployer CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD2KM
    --salt 123 \
    --wasm_hash
7792a624b562b3d9414792f5fb5d72f53b9838fef2ed9a901471253970bc3b15 \
    --init_fn init \
    --init_args '[{"u32":5}]'
```

```
soroban contract invoke --id 1 -- deploy `
    --deployer CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD2KM
    --salt 123 `
    --wasm_hash
7792a624b562b3d9414792f5fb5d72f53b9838fef2ed9a901471253970bc3b15 `
    --init_fn init `
    --init_args '[{"u32":5}]'
```

And then invoke the deployed test contract using the identifier returned from the previous command.

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke \
    --id ead19f55aec09bfcb555e09f230149ba7f72744a5fd639804ce1e934e8fe9c5d \
    -- \
    value
```

```
soroban contract invoke `
    --id ead19f55aec09bfcb555e09f230149ba7f72744a5fd639804ce1e934e8fe9c5d `
    -- `
    value
```

The following output should occur using the code above.

```
5
```

Allocator

The allocator example demonstrates how to utilize the allocator feature when writing a contract.

Open in Gitpod

The `soroban-sdk` crate provides a lightweight bump-pointer allocator which can be used to emulate heap memory allocation in a Wasm smart contract.

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `alloc` directory, and use `cargo test`.

```
cd alloc
cargo test
```

You should see the output:

```
running 1 test
```

Dependencies

This example depends on the `alloc` feature in `soroban-sdk`. To include it, add "alloc" to the "features" list of `soroban-sdk` in the `Cargo.toml` file:

alloc/Cargo.toml

```
[dependencies]
soroban-sdk = { version = "20.0.0", features = ["alloc"] }

[dev_dependencies]
soroban-sdk = { version = "20.0.0", features = ["testutils", "alloc"] }
```

Code

alloc/src/lib.rs

```
#![no_std]
use soroban_sdk::{contractimpl, Env};

extern crate alloc;

#[contract]
pub struct AllocContract;

#[contractimpl]
impl AllocContract {
    /// Allocates a temporary vector holding values (0..count), then computes
and returns their sum.
    pub fn sum(_env: Env, count: u32) -> u32 {
        let mut v1 = alloc::vec![];
        (0..count).for_each(|i| v1.push(i));

        let mut sum = 0;
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/alloc

How it Works

```
extern crate alloc;
```

Imports the `alloc` crate, which is required in order to support allocation under `no_std`. See Contract Rust dialect for more info about `no_std`.

```
let mut v1 = alloc::vec![];
```

Creates a contiguous growable array `v1` with contents allocated on the heap memory.

> INFO
>
> The heap memory in the context of a smart contract actually refers to the Wasm linear memory. The `alloc` will use the global allocator provided by the soroban sdk to interact with the linear memory.

> CAUTION
>
> Using heap allocated array is typically slow and computationally expensive. Try to avoid it and instead use a fixed-sized array or `soroban_sdk::vec!` whenever possible.
>
> This is especially the case for a large-size array. Whenever the array size grows beyond the current linear memory size, which is multiple of the page size (64KB), the `wasm32::memory_grow` is invoked to grow the linear memory by more pages as necessary, which is very computationally expensive.

The remaining code pushes values `(0..count)` to `v1`, then computes and returns their sum. This is the simplest example to illustrate how to use the allocator.

Atomic Swap

The atomic swap example swaps two tokens between two authorized parties atomically while following the limits they set.

This is example demonstrates advanced usage of Soroban auth framework and assumes the reader is familiar with the auth example and with Soroban token usage.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example use `cargo test`.

```
cargo test -p soroban-atomic-swap-contract
```

You should see the output:

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

Code

atomic_swap/src/lib.rs

```
#[contract]
pub struct AtomicSwapContract;

#[contractimpl]
impl AtomicSwapContract {
    // Swap token A for token B atomically. Settle for the minimum requested
price
    // for each party (this is an arbitrary choice to demonstrate the usage
of
    // allowance; full amounts could be swapped as well).
    pub fn swap(
        env: Env,
        a: Address,
        b: Address,
        token_a: Address,
        token_b: Address,
        amount_a: i128,
        min_b_for_a: i128,
        amount_b: i128,
        min_a_for_b: i128,
    ) {
        // Verify preconditions on the minimum price for both parties.
        if amount_b < min_b_for_a {
            panic!("not enough token B for token A");
        }
        if amount_a < min_a_for_b {
            panic!("not enough token A for token B");
        }
        // Require authorization for a subset of arguments specific to a
party.
        // Notice, that arguments are symmetric - there is no difference
between
        // `a` and `b` in the call and hence their signatures can be used
        // either for `a` or for `b` role.
        a.require_auth_for_args(
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/atomic_swap

How it Works

The example contract requires two `Address`-es to authorize their parts of the swap operation: one `Address` wants to sell a given amount of token A for token B at a given price and another `Address` wants to sell token B for token A at a given price. The contract swaps the tokens atomically, but only if the requested minimum price is respected for both parties.

Open the `atomic_swap/src/lib.rs` file or see the code above to follow along.

Swap authorization

```
...
a.require_auth_for_args(
    (token_a.clone(), token_b.clone(), amount_a, min_b_for_a).into_val(&env),
);
b.require_auth_for_args(
    (token_b.clone(), token_a.clone(), amount_b, min_a_for_b).into_val(&env),
);
...
```

Authorization of `swap` function leverages `require_auth_for_args` Soroban host function. Both `a` and `b` need to authorize symmetric arguments: token they sell, token they buy, amount of token they sell, minimum amount of token they want to receive. This means that `a` and `b` can be freely exchanged in the invocation arguments (as long as the respective arguments are changed too).

Moving the tokens

```
...
// Perform the swap via two token transfers.
move_token(&env, token_a, &a, &b, amount_a, min_a_for_b);
move_token(&env, token_b, &b, &a, amount_b, min_b_for_a);
...
fn move_token(
    env: &Env,
    token: &Address,
    from: &Address,
    to: &Address,
    max_spend_amount: i128,
    transfer_amount: i128,
) {
    let token = token::Client::new(env, token);
    let contract_address = env.current_contract_address();
    // This call needs to be authorized by `from` address. It transfers the
    // maximum spend amount to the swap contract's address in order to
decouple
    // the signature from `to` address (so that parties don't need to know
each
    // other).
    token.transfer(from, &contract_address, &max_spend_amount);
    // Transfer the necessary amount to `to`.
    token.transfer(&contract_address, to, &transfer_amount);
    // Refund the remaining balance to `from`.
    token.transfer(
        &contract_address,
        from,
        &(&max_spend_amount - &transfer_amount),
    );
}
```

The swap itself is implemented via two token moves: from `a` to `b` and from `b` to `a`. The token move is implemented via allowance: the users don't need to know each other in order to perform the swap, and instead they authorize the swap contract to spend the necessary amount of token on their behalf via `incr_allow`. Soroban auth framework makes sure that the `incr_allow` signatures would have the proper context, and they won't be usable outside the `swap` contract invocation.

Tests

Open the `atomic_swap/src/test.rs` file to follow along.

Refer to another examples for the general information on the test setup.

The interesting part for this example is verification of `swap` authorization:

```
contract.swap(
    &a,
    &b,
    &token_a.address,
    &token_b.address,
    &1000,
    &4500,
    &5000,
    &950,
);

assert_eq!(
    env.auths(),
    std::vec![
        (
            a.clone(),
            AuthorizedInvocation {
                function: AuthorizedFunction::Contract((
                    contract.address.clone(),
                    symbol_short!("swap"),
                    (
                        token_a.address.clone(),
                        token_b.address.clone(),
                        1000_i128,
                        4500_i128
                    )
                        .into_val(&env),
                )),
                sub_invocations: std::vec![AuthorizedInvocation {
                    function: AuthorizedFunction::Contract((
```

`env.auths()` returns all the authorizations. In the case of `swap` four authorizations are expected. Two for each address authorizing, because each address authorizes not only the swap, but the `approve` all on the token being sent.

Batched Atomic Swaps

The atomic swap batching example swaps a pair of tokens between the two groups of users that authorized the `swap` operation from the Atomic Swap example.

This contract basically batches the multiple swaps while following some simple rules to match the swap participants.

Follow the comments in the code for more information.

Open in Gitpod

Timelock

The timelock example demonstrates how to write a timelock and implements a greatly simplified claimable balance similar to the claimable balance feature available on Stellar.

Open in Gitpod

The contract accepts deposits of an amount of a token, and allows other users to claim it before or after a time point.

Single Offer Sale

The single offer sale example demonstrates how to write a contract that allows a seller to set up an offer to sell token A for token B to multiple buyers. The comments in the source code explain how the contract should be used.

Open in Gitpod

Liquidity Pool

The liquidity pool example demonstrates how to write a constant product liquidity pool contract. A liquidity pool is an automated way to add liquidity for a set of tokens that will facilitate asset conversion between them. Users can deposit some amount of each token into the pool, receiving a proportional number of "token shares." The user will then receive a portion of the accrued conversion fees when they ultimately "trade in" their token shares to receive their original tokens back.

Soroban liquidity pools are exclusive to Soroban and cannot interact with built-in Stellar AMM liquidity pools.

> CAUTION
>
> Implementing a custom liquidity pool should be done cautiously. User funds are involved, so great care should be taken to ensure safety and transparency. The example here should *not* be considered a ready-to-go contract. Please use it as a reference only.
>
> The Stellar network already has liquidity pool functionality built right in to the core protocol. Learn more here.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `liquidity_pool` directory, and use `cargo test`.

```
cd liquidity_pool
cargo test
```

You should see the output:

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

Code

> INFO
>
> Since our liquidity pool will be issuing its own token to establish the nuber of shares in the pool the address has, we have created a `token.rs` module in this project to hold the logic controlling the token contract for those shares.

lib.rs **token.rs**

liquidity_pool/src/lib.rs

```rust
#![no_std]

mod test;
mod token;

use num_integer::Roots;
use soroban_sdk::{
    contract, contractimpl, contractmeta, Address, BytesN, ConversionError,
Env, IntoVal,
    TryFromVal, Val,
};
use token::create_contract;

#[derive(Clone, Copy)]
#[repr(u32)]
pub enum DataKey {
    TokenA = 0,
    TokenB = 1,
    TokenShare = 2,
    TotalShares = 3,
    ReserveA = 4,
    ReserveB = 5,
}

impl TryFromVal<Env, DataKey> for Val {
    type Error = ConversionError;

    fn try_from_val(_env: &Env, v: &DataKey) -> Result<Self, Self::Error> {
        Ok((*v as u32).into())
    }
}

fn get_token_a(e: &Env) -> Address {
    e.storage().instance().get(&DataKey::TokenA).unwrap()
}

fn get_token_b(e: &Env) -> Address {
    e.storage().instance().get(&DataKey::TokenB).unwrap()
}

fn get_token_share(e: &Env) -> Address {
    e.storage().instance().get(&DataKey::TokenShare).unwrap()
```

liquidity_pool/src/token.rs

```rust
#![allow(unused)]
use soroban_sdk::{xdr::ToXdr, Address, Bytes, BytesN, Env};

soroban_sdk::contractimport!(
    file = "../token/target/wasm32-unknown-unknown/release/
soroban_token_contract.wasm"
);

pub fn create_contract(
    e: &Env,
    token_wasm_hash: BytesN<32>,
    token_a: &Address,
    token_b: &Address,
) -> Address {
    let mut salt = Bytes::new(e);
    salt.append(&token_a.to_xdr(e));
    salt.append(&token_b.to_xdr(e));
    let salt = e.crypto().sha256(&salt);
    e.deployer()
        .with_current_contract(salt)
        .deploy(token_wasm_hash)
}
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/liquidity_pool

How it Works

Every asset created on Stellar starts with zero liquidity. The same is true of tokens created on Soroban (unless a Stellar asset with existing liquidity token is "wrapped" for use in Soroban). In simple terms, "liquidity" means how much of an asset in a market is available to be bough or sold. In the "old days," you could generate liquidity in a market by creating buy/sell orders on an order book.

Liquidity pools automate this process by substituting the orders with math.

Depositors into the liquidity pool earn fees from `swap` transactions. No orders required!

Open the `liquidity_pool/src/lib.rs` file or see the code above to follow along.

Initialize the Contract

When this contract is first deployed, it could create a liquidity pool for *any* pair of tokens available on Soroban. It must first be initialized with the following information:

- `token_wasm_hash`**:** The contract will end up creating its own `POOL` token as well as interacting with contracts for `token_a` and `token_b`. The way this example works is by using the `token` example contract for both of these jobs. When our liquidity pool contract is initialized it wants us to pass the wasm hash of the **already installed** token contract. It will then deploy a contract that will run the WASM bytecode stored at that hash as a new token contract for the `POOL` tokens.
- `token_a`**:** The contract `Address` for an **already deployed** (or wrapped) token that will be held in reserve by the liquidity pool.
- `token_b`**:** The contract `Address` for an **already deployed** (or wrapped) token that will be held in reserve by the liquidity pool.

Bear in mind that which token is `token_a` and which is `token_b` is **not** an arbitrary distinction. In line with the Built-in Stellar liquidity pools, this contract can only make a single liquidity pool for a given set of tokens. So, the token addresses must be provided in lexicographical order at the time of initialization.

liquidity_pool/src/lib.rs

```
fn initialize(e: Env, token_wasm_hash: BytesN<32>, taken_a: Address,
token_b: Address) {
```

A "Constant Product" Liquidity Pool

The *type* of liquidity pool this example contract implements is called a "constant product" liquidity pool. While this isn't the only type of liquidity pool out there, it is the most common variety. These liquidity pools are designed to keep the *total* value of each asset in *relative* equilibrium. The "product" in the constant product (also called an "invariant") will change every time the liquidity pool is interacted with (deposit, withdraw, or token swaps). However, the invariant **must** only increase with every interaction.

During a swap, what must be kept in mind is that for every withdrawal from the `token_a` side, you must "refill" the `token_b` side with a sufficient amount to keep the liquidity pool's price balanced. The math is predictable, but it is not linear. The more you take from one side, the more you must give on the opposite site *exponentially*.

Inside the `swap` function, the math is done like this (this is a simplified version, however):

liquidity_pool/src/lib.rs

```rust
fn swap(e: Env, to: Address, buy_a: bool, out: i128, in_max: i128) {
    // Get the current balances of both tokens in the liquidity pool
    let (reserve_sell, reserve_buy) = (get_reserve_a(&e), get_reserve_b(&e));

    // Calculate how much needs to be
    let n = reserve_sell * out * 1000;
    let d = (reserve_buy - out) * 997;
    let sell_amount = (n / d) + 1;
}
```

We have much more in-depth information about how this kind of liquidity pool works is available in Stellar Quest: Series 3, Quest 5. This is a really useful, interactive way to learn more about how the built-in Stellar liquidity pools work. Much of the knowledge you might gain from there will easily translate to this example contract.

Interacting with Token Contracts in Another Contract

This liquidity pool contract will operate with a total of three different Soroban tokens:

- `POOL`: This token is a unique token that is given to asset depositors in exchange for their deposit. These tokens are "traded in" by the user when they withdraw some amount of their original deposit (plus any earned swap fees). This example contract implements the same `token` example contract for this token.
- `token_a` and `token_b`: Will be the two "reserve tokens" that users will deposit into the pool. These could be "wrapped" tokens from pre-existing Stellar assets, or they could be Soroban-native tokens. This contract doesn't really care, as long as the functions it needs from the common Token Interface are available in the token contract.

Creating a Custom `POOL` Token for LP Shares

We are utilizing the compiled `token` example contract as our asset contract for the `POOL` token. This means it follows all the conventions of the Token Interface, and can be treated just like any other token. They could be transferred, burned, minted, etc. It also means the LP developer *could* take advantage of the administrative features such as clawbacks, authorization, and more.

The `token.rs` file contains a `create_contract` function that we will use to deploy this particular token contract.

src/token.rs

```rust
pub fn create_contract(
    e: &Env,
    token_wasm_hash: BytesN<32>,
    token_a: &Address,
    token_b: &Address,
) -> Address {
    let mut salt = Bytes::new(e);
    salt.append(&token_a.to_xdr(e));
    salt.append(&token_b.to_xdr(e));
    let salt = e.crypto().sha256(&salt);
    e.deployer()
        .with_current_contract(salt)
        .deploy(token_wasm_hash)
}
```

This `POOL` token contract is then created within the `initialize` function.

liquidity_pool/src/lib.rs

```rust
fn initialize(e: Env, token_wasm_hash: BytesN<32>, token_a: Address,
token_b: Address) {
    let share_contract = create_contract(&e, token_wasm_hash, &token_a,
&token_b);
    token::Client::new(&e, &share_contract).initialize(
        &e.current_contract_address(),
        &7u32,
        &"Pool Share Token".into_val(&e),
        &"POOL".into_val(&e),
    );
}
```

Then, during a `deposit`, a calculated amount of `POOL` tokens are `mint`ed to the depositing address.

liquidity_pool/src/lib.rs

```rust
fn mint_shares(e: &Env, to: Address, amount: i128) {
```

How is that number of shares calculated, you ask? Excellent question! If it's the very first deposit (see above), it's just the square root of the product of the quantities of `token_a` and `token_b` deposited. Very simple.

However, if there have already been deposits into the liquidity pool, and the user is just adding more tokens into the pool, there's a bit more math. However, the main point is that each depositor receives the same ratio of `POOL` tokens for their deposit as every other depositor.

liquidity_pool/src/lib.rs

```
fn deposit(e: Env, to: Address, desired_a: i128, min_a: i128, desired_b: 
i128, min_b: i128) {
    let zero = 0;
    let new_total_shares = if reserve_a > zero && reserve_b > zero {
        // Note balance_a and balance_b at this point in the function include
        // the tokens the user is currently depositing, whereas reserve_a and
        // reserve_b do not yet.
        let shares_a = (balance_a * total_shares) / reserve_a;
        let shares_b = (balance_b * total_shares) / reserve_b;
        shares_a.min(shares_b)
    } else {
        (balance_a * balance_b).sqrt()
    };
}
```

Token Transfers to/from the LP Contract

As we've already discussed, the liquidity pool contract will make use of the Token Interface available in the token contracts that were supplied as `token_a` and `token_b` arguments at the time of initialization. Throughout the rest of the contract, the liquidity pool will make use of that interface to make transfers of those tokens to/from itself.

What's happening is that as a user deposits tokens into the pool, and the contract invokes the `transfer` function to move the tokens from the `to` address (the depositor) to be held by the contract address. `POOL` tokens are then minted to depositor (see previous section). Pretty simple, right!?

liquidity_pool/src/lib.rs

```
fn deposit(e: Env, to: Address, desired_a: i128, min_a: i128, desired_b:
i128, min_b: i128) {
    // Depositor needs to authorize the deposit
    to.require_auth();

    let token_a_client = token::Client::new(&e, &get_token_a(&e));
    let token_b_client = token::Client::new(&e, &get_token_b(&e));

    token_a_client.transfer(&to, &e.current_contract_address(), &amounts.0);
    token_b_client.transfer(&to, &e.current_contract_address(), &amounts.1);

    mint_shares(&e, to, new_total_shares - total_shares);
}
```

In contrast, when a user withdraws their deposited tokens, It's about more involved, and the following procedure happens.

1. Some amount of the `POOL` token is transferred from the depositor to the contract address. This is a temporary way to track how many `POOL` tokens are being redeemed. The contract will not hold this balance of `POOL` for long.
2. The withdraw amounts for the reserve tokens are calculated based on the contract's current balance of `POOL` tokens.
3. The `POOL` tokens are burned now that the withdraw amounts have been calculated, and they are no longer needed.
4. The respective amounts of `token_a` and `token_b` are transferred *from* the contract address into the `to` address (the depositor).

liquidity_pool/src/lib.rs

```
fn withdraw(e: Env, to: Address, share_amount: i128, min_a: i128, min_b:
i128) -> (i128, i128) {
    to.require_auth();

    // First transfer the pool shares that need to be redeemed
    let share_token_client = token::Client::new(&e, &get_token_share(&e));
    share_token_client.transfer(&to, &e.current_contract_address(),
&share_amount);

    // Now calculate the withdraw amounts
    let out_a = (balance_a * balance_shares) / total_shares;
    let out_b = (balance_b * balance_shares) / total_shares;

    burn_shares(&e, balance_shares);
    transfer_a(&e, to.clone(), out_a);
    transfer_b(&e, to, out_b);
}
```

You'll notice that by holding the balance of `token_a` and `token_b` on the liquidity pool contract itself it makes, it very easy for us to perform any of the Token Interface actions inside the contract. As a bonus, any outside observer could query the balances of `token_a` or `token_b` held by the contract to verify the reserves are actually in line with the values the contract reports when its own `get_rsvs` function is invoked.

Tests

Open the `liquidity_pool/src/test.rs` file to follow along.

liquidity_pool/src/test.rs

```
#![cfg(test)]
extern crate std;

use crate::{token, LiquidityPoolClient};
```

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

liquidity_pool/src/test.rs

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

We mock authentication checks in the tests, which allows the tests to proceed as if all users/addresses/contracts/etc. had successfully authenticated.

liquidity_pool/src/test.rs

```
e.mock_all_auths();
```

We have abstracted into a few functions the tasks of creating token contracts, deploying a liquidity pool contract, and installing the token example WASM bytecode into our test environment. Each are then used within the test.

liquidity_pool/src/test.rs

```
fn create_token_contract<'a>(e: &Env, admin: &Address) -> token::Client<'a> {
    token::Client::new(e, &e.register_stellar_asset_contract(admin.clone()))
}

fn create_liqpool_contract<'a>(
    e: &Env,
    token_wasm_hash: &BytesN<32>,
    token_a: &Address,
    token_b: &Address,
) -> LiquidityPoolClient<'a> {
    let liqpool = LiquidityPoolClient::new(e, &e.register_contract(None,
crate::LiquidityPool {}));
    liqpool.initialize(token_wasm_hash, token_a, token_b);
    liqpool
}
```

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 `LiquidityPool`, and the client is named `LiquidityPoolClient`.

These tests examine the "typical" use-case of a liquidity pool, ensuring that the balances, returns, etc. are appropriate at various points during the test.

1. First, the test sets everything up with an `Env`, two admin addresses, two reserve tokens, a randomly generated address to act as the user of the liquidity pool, the liquidity pool itself, a pool token shares contract, and mints the reserve assets to the user address.
2. The user then deposits some of each asset into the liquidity pool. At this time, the following checks are done:
    - appropriate authorizations for deposits and transfers exist,
    - balances are checked for each token (`token_a`, `token_b`, and `POOL`) from both the user's perspective and the `liqpool` contract's perspective
3. The user performs a swap, buying `token_b` in exchange for `token_a`. The same checks as the previous step are made now, excepting the balances of `POOL`, since a swap has no effect on `POOL` tokens.
4. The user then withdraws all of the deposits it made, trading all of its `POOL` tokens in the process. The same checks are made here as were made in the `deposit` step.

Build the Contract

To build the contract, use the `soroban contract build` command.

```
soroban contract build
```

A `.wasm` file should be outputted in the `target` directory:

```
target/wasm32-unknown-unknown/release/soroban_liquidity_pool_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke contract functions using it.

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke \
    --wasm target/wasm32-unknown-unknown/release/
soroban_liquidity_pool_contract.wasm \
    --id 1 \
    -- \
    deposit \
    --to GBZV3NONYSUDVTEHATQO4BCJVFXJO3XQU5K32X3XREVZKSMMOZFO4ZXR \
    --desired_a 100 \
    --min_a 98 \
    --desired_be 200 \
    --min_b 196
```

```
soroban contract invoke `
    --wasm target/wasm32-unknown-unknown/release/
soroban_liquidity_pool_contract.wasm `
    --id 1 `
    -- `
    deposit `
    --to GBZV3NONYSUDVTEHATQO4BCJVFXJO3XQU5K32X3XREVZKSMMOZFO4ZXR `
    --desired_a 100 `
    --min_a 98 `
    --desired_be 200 `
    --min_b 196
```

Tokens

The token example demonstrates how to write a token contract that implements the Token Interface.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example, navigate to the `hello_world` directory, and use `cargo test`.

```
cd token
cargo test
```

You should see the output:

```
running 8 tests
test test::initialize_already_initialized - should panic ... ok
test test::transfer_spend_deauthorized - should panic ... ok
test test::decimal_is_over_max - should panic ... ok
test test::test_burn ... ok
```

Code

> NOTE
>
> The source code for this token example is broken into several smaller modules. This is a common design pattern for more complex smart contracts.

lib **admin** **allowance** **balance** **contract** **event** **metadata**

storage_types

token/src/lib.rs

```rust
#![no_std]

mod admin;
mod allowance;
mod balance;
mod contract;
mod event;
mod metadata;
mod storage_types;
mod test;

pub use crate::contract::TokenClient;
```

token/src/admin.rs

```rust
use crate::storage_types::DataKey;
use soroban_sdk::{Address, Env, symbol_short};
```

token/src/allowance.rs

```rust
use crate::storage_types::{AllowanceDataKey, AllowanceValue, DataKey};
use soroban_sdk::{Address, Env};

pub fn read_allowance(e: &Env, from: Address, spender: Address) ->
AllowanceValue {
    let key = DataKey::Allowance(AllowanceDataKey { from, spender });
    if let Some(allowance) = e.storage().temporary().get::<_,
AllowanceValue>(&key) {
        if allowance.expiration_ledger < e.ledger().sequence() {
            AllowanceValue {
                amount: 0,
                expiration_ledger: allowance.expiration_ledger,
            }
        } else {
            allowance
        }
    } else {
        AllowanceValue {
            amount: 0,
            expiration_ledger: 0,
        }
    }
}

pub fn write_allowance(
    e: &Env,
    from: Address,
    spender: Address,
    amount: i128,
    expiration_ledger: u32,
) {
    let allowance = AllowanceValue {
        amount,
        expiration_ledger,
    };

    if amount > 0 && expiration_ledger < e.ledger().sequence() {
        panic!("expiration_ledger is less than ledger seq when amount > 0")
    }
```

token/src/balance.rs

```rust
use crate::storage_types::DataKey;
use soroban_sdk::{Address, Env};

pub fn read_balance(e: &Env, addr: Address) -> i128 {
    let key = DataKey::Balance(addr);
    if let Some(balance) = e.storage().persistent().get::<DataKey,
i128>(&key) {
        balance
    } else {
        0
    }
}

fn write_balance(e: &Env, addr: Address, amount: i128) {
    let key = DataKey::Balance(addr);
    e.storage().persistent().set(&key, &amount);
}

pub fn receive_balance(e: &Env, addr: Address, amount: i128) {
    let balance = read_balance(e, addr.clone());
    if !is_authorized(e, addr.clone()) {
        panic!("can't receive when deauthorized");
    }
    write_balance(e, addr, balance + amount);
}

pub fn spend_balance(e: &Env, addr: Address, amount: i128) {
    let balance = read_balance(e, addr.clone());
    if !is_authorized(e, addr.clone()) {
        panic!("can't spend when deauthorized");
    }
    if balance < amount {
        panic!("insufficient balance");
    }
    write_balance(e, addr, balance - amount);
}

pub fn is_authorized(e: &Env, addr: Address) -> bool {
    let key = DataKey::State(addr);
    if let Some(state) = e.storage().persistent().get::<DataKey, bool>(&key)
```

token/src/contract.rs

```rust
//! This contract demonstrates a sample implementation of the Soroban token
//! interface.
use crate::admin::{has_administrator, read_administrator,
write_administrator};
use crate::allowance::{read_allowance, spend_allowance, write_allowance};
use crate::balance::{is_authorized, write_authorization};
use crate::balance::{read_balance, receive_balance, spend_balance};
use crate::event;
use crate::metadata::{read_decimal, read_name, read_symbol, write_metadata};
use soroban_sdk::{contractimpl, Address, String, Env};
use soroban_token_sdk::TokenMetadata;

pub trait TokenTrait {
    fn initialize(e: Env, admin: Address, decimal: u32, name: String,
symbol: String);

    fn allowance(e: Env, from: Address, spender: Address) -> i128;

    fn approve(e: Env, from: Address, spender: Address, amount: i128,
expiration_ledger: u32);

    fn balance(e: Env, id: Address) -> i128;

    fn spendable_balance(e: Env, id: Address) -> i128;

    fn authorized(e: Env, id: Address) -> bool;

    fn transfer(e: Env, from: Address, to: Address, amount: i128);

    fn transfer_from(e: Env, spender: Address, from: Address, to: Address,
amount: i128);

    fn burn(e: Env, from: Address, amount: i128);

    fn burn_from(e: Env, spender: Address, from: Address, amount: i128);

    fn clawback(e: Env, from: Address, amount: i128);

    fn set_authorized(e: Env, id: Address, authorize: bool);
```

token/src/event.rs

```rust
use soroban_sdk::{Address, Env, Symbol, symbol_short};

pub(crate) fn approve(e: &Env, from: Address, to: Address, amount: i128,
expiration_ledger: u32) {
    let topics = (Symbol::new(e, "approve"), from, to);
    e.events().publish(topics, (amount, expiration_ledger));
}

pub(crate) fn transfer(e: &Env, from: Address, to: Address, amount: i128) {
    let topics = (symbol_short!("transfer"), from, to);
    e.events().publish(topics, amount);
}

pub(crate) fn mint(e: &Env, admin: Address, to: Address, amount: i128) {
    let topics = (symbol_short!("mint"), admin, to);
    e.events().publish(topics, amount);
}

pub(crate) fn clawback(e: &Env, admin: Address, from: Address, amount: i128)
{
    let topics = (symbol_short!("clawback"), admin, from);
    e.events().publish(topics, amount);
}

pub(crate) fn set_authorized(e: &Env, admin: Address, id: Address,
authorize: bool) {
    let topics = (Symbol::new(e, "set_authorized"), admin, id);
    e.events().publish(topics, authorize);
}

pub(crate) fn set_admin(e: &Env, admin: Address, new_admin: Address) {
    let topics = (symbol_short!("set_admin"), admin);
    e.events().publish(topics, new_admin);
}

pub(crate) fn burn(e: &Env, from: Address, amount: i128) {
    let topics = (symbol_short!("burn"), from);
    e.events().publish(topics, amount);
}
```

token/src/metadata.rs

```rust
use soroban_sdk::{Bytes, Env};
use soroban_token_sdk::{TokenMetadata, TokenUtils};

pub fn read_decimal(e: &Env) -> u32 {
    let util = TokenUtils::new(e);
    util.get_metadata_unchecked().unwrap().decimal
}

pub fn read_name(e: &Env) -> Bytes {
    let util = TokenUtils::new(e);
    util.get_metadata_unchecked().unwrap().name
}

pub fn read_symbol(e: &Env) -> Bytes {
    let util = TokenUtils::new(e);
    util.get_metadata_unchecked().unwrap().symbol
}

pub fn write_metadata(e: &Env, metadata: TokenMetadata) {
    let util = TokenUtils::new(e);
    util.set_metadata(&metadata);
}
```

token/src/storage_types.rs

```rust
use soroban_sdk::{contracttype, Address};

#[derive(Clone)]
#[contracttype]
pub struct AllowanceDataKey {
    pub from: Address,
    pub spender: Address,
}

#[contracttype]
pub struct AllowanceValue {
    pub amount: i128,
    pub expiration_ledger: u32,
}
```

Ref: https://github.com/stellar/soroban-examples/tree/v20.0.0/token

How it Works

Tokens created on a smart contract platform can take many different forms, include a variety of different functionalities, and meet very different needs or use-cases. While each token can fulfill a unique niche, there are some "normal" features that almost all tokens will need to make use of (e.g., payments, transfers, balance queries, etc.). In an effort to minimize repetition and streamline token deployments, Soroban implements the Token Interface, which provides a uniform, predictable interface for developers and users.

Creating a Soroban token compatible contract from an existing Stellar asset is very easy, it requires deploying the built-in Stellar Asset Contract.

This example contract, however, demonstrates how a smart contract token might be constructed that doesn't take advantage of the Stellar Asset Contract, but does still satisfy the commonly used Token Interface to maximize interoperability.

Separation of Functionality

You have likely noticed that this example contract is broken into discrete modules, with each one responsible for a siloed set of functionality. This common practice helps to organize the code and make it more maintainable.

For example, most of the token logic exists in the `contract.rs` module. Functions like `mint`, `burn`, `transfer`, etc. are written and programmed in that file. The Token Interface describes how some of these functions should emit events when they occur. However, keeping all that event-emitting logic bundled in with the rest of the contract code could make it harder to track what is happening in the code, and that confusion could ultimately lead to errors.

Instead, we have a separate `events.rs` module that takes away all the headache of emitting events when other functions run. Here is the function to emit an event whenever the token is minted:

```
pub(crate) fn mint(e: &Env, admin: Address, to: Address, amount: i128) {
    let topics = (symbol_short!("mint"), admin, to);
    e.events().publish(topics, amount);
}
```

Admittedly, this is a simple example, but constructing the contract this way makes it very clear to the developer what is happening and where. This function is then used by the `contract.rs` module whenever the `mint` function is invoked:

```
// earlier in `contract.rs`
use crate::event;

fn mint(e: Env, to: Address, amount: i128) {
    check_nonnegative_amount(amount);
    let admin = read_administrator(&e);
    admin.require_auth();
    receive_balance(&e, to.clone(), amount);
    event::mint(&e, admin, to, amount);
}
```

This same convention is used to separate from the "main" contract code the metadata for the token, the storage type definitions, etc.

Standardized Interface, Customized Behavior

This example contract follows the standardized Token Interface, implementing all of the same functions as the Stellar Asset Contract. This gives wallets, users, developers, etc. a predictable interface to interact with the token. Even though we are implementing the same *interface* of functions, that doesn't mean we have to implement the same *behavior* inside those functions. While this example contract doesn't actually modify any of the functions that would be present in a deployed instance of the Stellar Asset Contract, that possibility remains open to the contract developer.

By way of example, perhaps you have an NFT project, and the artist wants to have a small royalty paid every time their token transfers hands:

```
// This is mainly the `transfer` function from `src/contract.rs`
fn transfer(e: Env, from: Address, to: Address, amount: i128) {
    from.require_auth();

    check_nonnegative_amount(amount);
    spend_balance(&e, from.clone(), amount);
    // We calculate some new amounts for payment and royalty
    let payment = (amount * 997) / 1000;
    let royalty = amount - payment
    receive_balance(&e, artist.clone(), royalty);
    receive_balance(&e, to.clone(), payment);
    event::transfer(&e, from, to, amount);
}
```

The `transfer` interface is still in use, and is still the same as other tokens, but we've customized the behavior to address a specific need. Another use-case might be a tightly controlled token that requires authentication from an admin before any `transfer`, `allowance`, etc. function could be invoked.

> TIP
>
> Of course, you will want your token to behave in an *intuitive* and *transparent* manner. If a user is invoking a `transfer`, they will expect tokens to move. If an asset issuer needs to invoke a `clawback` they will likely *require* the right kind of behavior to take place.

Tests

Open the `token/src/test.rs` file to follow along.

token/src/test.rs

```rust
#![cfg(test)]
extern crate std;

use crate::{contract::Token, TokenClient};
use soroban_sdk::{testutils::Address as _, Address, Env, IntoVal, Symbol};

fn create_token<'a>(e: &Env, admin: &Address) -> TokenClient<'a> {
    let token = TokenClient::new(e, &e.register_contract(None, Token {}));
    token.initialize(admin, &7, &"name".into_val(e), &"symbol".into_val(e));
    token
}

#[test]
fn test() {
    let e = Env::default();
    e.mock_all_auths();

    let admin1 = Address::random(&e);
    let admin2 = Address::random(&e);
    let user1 = Address::random(&e);
    let user2 = Address::random(&e);
    let user3 = Address::random(&e);
    let token = create_token(&e, &admin1);

    token.mint(&user1, &1000);
    assert_eq!(
        e.auths(),
        std::vec![(
            admin1.clone(),
            AuthorizedInvocation {
                function: AuthorizedFunction::Contract((
                    token.address.clone(),
                    symbol_short!("mint"),
```

The token example implements eight different tests to cover a wide array of potential behaviors and problems. However, all of the tests start with a few common pieces. In any test, the first thing that is always required is an `Env`, which is the Soroban environment that the contract will run in.

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

We mock authentication checks in the tests, which allows the tests to proceed as if all users/addresses/contracts/etc. had successfully authenticated.

```
e.mock_all_auths();
```

We're also using a `create_token` function to ease the repetition of having to register and initialize our token contract. The resulting `token` client is then used to invoke the contract during each test.

```
// It is defined at the top of the file...
fn create_token<'a>(e: &Env, admin: &Address) -> TokenClient<'a> {
    let token = TokenClient::new(e, &e.register_contract(None, Token {}));
    token.initialize(admin, &7, &"name".into_val(e), &"symbol".into_val(e));
    token
}

// ... and it is used inside each test
let token = create_token(&e, &admin);
```

All public functions within an `impl` block that has been annotated with the `#[contractimpl]` attribute will have a corresponding function in the test's 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 named `Token`, and the client type is named `TokenClient`.

The eight tests created for this example contract test a range of possible conditions and ensure the contract responds appropriately to each one:

- `test()` - This function makes use of a variety of the built-in token functions to test the "predictable" way an asset might be interacted with by a user, as well as an administrator.
- `test_burn()` - This function ensures a `burn()` invocation decreases a user's balance, and that a `burn_from()` invocation decreases a user's balance as well as consuming another user's allowance of that balance.
- `transfer_insufficient_balance()` - This function ensures a `transfer()` invocation panics when the `from` user doesn't have the balance to cover it.
- `transfer_receive_deauthorized()` - This function ensures a user who is specifically de-authorized to hold the token cannot be the beneficiary of a `transfer()` invocation.
- `transfer_spend_deauthorized()` - This function ensures a user with a token balance, who is subsequently de-authorized cannot be the source of a `transfer()` invocation.
- `transfer_from_insufficient_allowance()` - This function ensures a user with an existing allowance for someone else's balance cannot make a `transfer()` greater than that allowance.
- `initialize_already_initialized()` - This function checks that the contract cannot have it's `initialize()` function invoked a second time.
- `decimal_is_over_max()` - This function tests that invoking `initialize()` with too high of a decimal precision will not succeed.

Build the Contract

To build the contract, use the `soroban contract build` command.

```
soroban contract build
```

A `.wasm` file should be outputted in the `target` directory:

```
target/wasm32-unknown-unknown/release/soroban_token_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke contract functions using it.

macOS/Linux **Windows (PowerShell)**

```
soroban contract invoke \
    --wasm target/wasm32-unknown-unknown/release/soroban_token_contract.wasm
\
    --id 1 \
    -- \
    balance \
    --id GBZV3NONYSUDVTEHATQO4BCJVFXJO3XQU5K32X3XREVZKSMMOZFO4ZXR
```

```
soroban contract invoke `
    --wasm target/wasm32-unknown-unknown/release/soroban_token_contract.wasm
`
    --id 1 `
    -- `
    balance `
    --id GBZV3NONYSUDVTEHATQO4BCJVFXJO3XQU5K32X3XREVZKSMMOZFO4ZXR
```

Custom Account

The custom account example demonstrates how to implement a simple account contract that supports multisig and customizable authorization policies. This account contract can be used with the Soroban auth framework, so that any time an `Address` pointing at this contract instance is used, the custom logic implemented here is applied.

Custom accounts are exclusive to Soroban and can't be used to perform other Stellar operations.

> DANGER
>
> Implementing a custom account contract requires a very good understanding of authentication and authorization and requires rigorous testing and review. The example here is *not* a full-fledged account contract - use it as an API reference only.

> CAUTION
>
> While custom accounts are supported by the Stellar protocol and Soroban SDK, the full client support (such as transaction simulation) is still under development.

Open in Gitpod

Run the Example

First go through the Setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

Or, skip the development environment setup and open this example in Gitpod.

To run the tests for the example use `cargo test`.

```
cargo test -p soroban-account-contract
```

You should see the output:

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

How it Works

Open the `account/src/lib.rs` file to follow along.

Account contracts implement a special function `__check_auth` that takes the signature payload, signatures and authorization context. The function should error if auth is declined, otherwise auth will be approved.

This example contract uses ed25519 keys for signature verification and supports multiple equally weighted signers. It also implements a policy that allows setting per-token limits on transfers. The token can be spent beyond the limit only if every signature is provided.

For example, the user may initialize this contract with 2 keys and introduce 100 USDC spend limit. This way they can use a single key to sign their contract invocations and be sure that even if they sign a malicious transaction they won't spend more than 100 USDC.

Initialization

```
#[contracttype]
#[derive(Clone)]
enum DataKey {
    SignerCnt,
    Signer(BytesN<32>),
    SpendLimit(BytesN<32>),
}
...
// Initialize the contract with a list of ed25519 public key ('signers').
pub fn init(env: Env, signers: Vec<BytesN<32>>) {
    // In reality this would need some additional validation on signers
    // (deduplication etc.).
    for signer in signers.iter() {
        env.storage().instance().set(&DataKey::Signer(signer), &());
    }
    env.storage()
        .instance()
        .set(&DataKey::SignerCnt, &signers.len());
}
```

This account contract needs to work with the public keys explicitly. Here we initialize the contract with ed25519 keys.

Policy modification

```
// Adds a limit on any token transfers that aren't signed by every signer.
pub fn add_limit(env: Env, token: BytesN<32>, limit: i128) {
    // The current contract address is the account contract address and has
    // the same semantics for `require_auth` call as any other account
    // contract address.
    // Note, that if a contract *invokes* another contract, then it would
    // authorize the call on its own behalf and that wouldn't require any
```

This function allows users to set and modify the per-token spend limit described above. The neat trick here is that `require_auth` can be used for the `current_contract_address()`, i.e. the account contract may be used to verify authorization for its own administrative functions. This way there is no need to write duplicate authorization and authentication logic.

`__check_auth`

```
pub fn __check_auth(
    env: Env,
    signature_payload: BytesN<32>,
    signatures: Vec<Signature>,
    auth_context: Vec<Context>,
) -> Result<(), AccError> {
    // Perform authentication.
    authenticate(&env, &signature_payload, &signatures)?;

    let tot_signers: u32 = env
        .storage()
        .instance()
        .get::<_, u32>(&DataKey::SignerCnt)
        .unwrap();
    let all_signed = tot_signers == signatures.len();

    let curr_contract = env.current_contract_address();

    // This is a map for tracking the token spend limits per token. This
    // makes sure that if e.g. multiple `transfer` calls are being authorized
    // for the same token we still respect the limit for the total
    // transferred amount (and not the 'per-call' limits).
    let mut spend_left_per_token = Map::<Address, i128>::new(&env);
    // Verify the authorization policy.
    for context in auth_context.iter() {
        verify_authorization_policy(
            &env,
            &context,
            &curr_contract,
            all_signed,
            &mut spend_left_per_token,
        )?;
```

`__check_auth` is a special function that account contracts implement. It will be called by the Soroban environment every time `require_auth` or `require_auth_for_args` is called for the address of the account contract.

Here it is implemented in two steps. First, authentication is performed using the signature payload and a vector of signatures. Second, authorization policy is enforced using the `auth_context` vector. This vector contains all the contract calls that are being authorized by the provided signatures.

`__check_auth` is a reserved function and can only be called by the Soroban environment in response to a call to `require_auth`. Any direct call to `__check_auth` will fail. This makes it safe to write to the account contract storage from `__check_auth`, as it's guaranteed to not be called in unexpected context. In this example it's possible to persist the spend limits without worrying that they'll be exhausted via a bad actor calling `__check_auth` directly.

Authentication

```rust
fn authenticate(
    env: &Env,
    signature_payload: &BytesN<32>,
    signatures: &Vec<Signature>,
) -> Result<(), AccError> {
    for i in 0..signatures.len() {
        let signature = signatures.get_unchecked(i);
        if i > 0 {
            let prev_signature = signatures.get_unchecked(i - 1);
            if prev_signature.public_key >= signature.public_key {
                return Err(AccError::BadSignatureOrder);
            }
        }
        if !env
            .storage()
            .instance()
            .has(&DataKey::Signer(signature.public_key.clone()))
        {
```

Authentication here simply checks that the provided signatures are valid given the payload and also that they belong to the signers of this account contract.

Authorization policy

```
fn verify_authorization_policy(
    env: &Env,
    context: &Context,
    curr_contract: &Address,
    all_signed: bool,
    spend_left_per_token: &mut Map<Address, i128>,
) -> Result<(), AccError> {
    // For the account control every signer must sign the invocation.
    let contract_context = match context {
        Context::Contract(c) => {
            if &c.contract == curr_contract {
                if !all_signed {
                    return Err(AccError::NotEnoughSigners);
                }
            }
            c
        }
        Context::CreateContractHostFn(_) => return
Err(AccError::InvalidContext),
    };
```

We verify the policy per `Context`. i.e. Per one `require_auth` call. The policy for the account contract itself enforces every signer to have signed the method call.

```
// Otherwise, we're only interested in functions that spend tokens.
if contract_context.fn_name != TRANSFER_FN
    && contract_context.fn_name != Symbol::new(env, "approve")
{
    return Ok(());
}

let spend_left: Option<i128> =
    if let Some(spend_left) =
```

Then we check for the standard token function names and verify that for these function we don't exceed the spending limits.

Tests

Open the `account/src/test.rs` file to follow along.

Refer to another examples for the general information on the test setup.

Here we only look at some points specific to the account contracts.

```
fn sign(e: &Env, signer: &Keypair, payload: &BytesN<32>) -> RawVal {
    Signature {
        public_key: signer_public_key(e, signer),
        signature: signer
            .sign(payload.to_array().as_slice())
            .to_bytes()
            .into_val(e),
    }
    .into_val(e)
}
```

Unlike most of the contracts that may simply use `Address`, account contracts deal with the signature verification and hence need to actually sign the payloads.

```
let payload = BytesN::random(&env);
let token = BytesN::random(&env);
env.try_invoke_contract_check_auth::<AccError>(
    &account_contract.address.contract_id(),
    &payload,
    &vec![&env, sign(&env, &signers[0], &payload)],
    &vec![
        &env,
        token_auth_context(&env, &token, Symbol::new(&env, "transfer"),
1000),
    ],
)
```

`__check_auth` can't be called directly as regular contract functions, hence we need to use `try_invoke_contract_check_auth` testing utility that emulates being called by the Soroban host during a `require_auth` call.

```
// Add a spend limit of 1000 per 1 signer.
account_contract.add_limit(&token, &1000);
// Verify that this call needs to be authorized.
assert_eq!(
    env.auths(),
    std::vec![(
        account_contract.address.clone(),
        AuthorizedInvocation {
            function: AuthorizedFunction::Contract((
                account_contract.address.clone(),
                symbol_short!("add_limit"),
                (token.clone(), 1000_i128).into_val(&env),
            )),
            sub_invocations: std::vec![]
        }
    )]
);
```

Asserting the contract-specific error to `try_invoke_contract_check_auth` allows verifying the exact error code and makes sure that the verification has failed due to not having enough signers and not for any other reason.

It's a good idea for the account contract to have detailed error codes and verify that they are returned when they are expected.

```
assert_eq!(
    env.try_invoke_contract_check_auth::<AccError>(
        &account_contract.address.contract_id(),
        &payload,
        &vec![&env, sign(&env, &signers[0], &payload)],
        &vec![
            &env,
            token_auth_context(&env, &token, Symbol::new(&env, "transfer"),
1001)
        ],
```

Fuzz Testing

The fuzzing example demonstrates how to fuzz test Soroban contracts with `cargo-fuzz` and customize the input to fuzz tests with the `arbitrary` crate. It also demonstrates how to adapt fuzz tests into reusable property tests with the `proptest` and `proptest-arbitrary-interop` crates. It builds on the timelock example.

Open in Gitpod

Run the Example

First go through the setup process to get your development environment configured, then clone the `v20.0.0` tag of `soroban-examples` repository:

```
git clone -b v20.0.0 https://github.com/stellar/soroban-examples
```

You will also need the `cargo-fuzz` tool, and to run `cargo-fuzz` you will need a nightly Rust toolchain:

```
cargo install cargo-fuzz
rustup install nightly
```

To run one of the fuzz tests, navigate to the `fuzzing` directory and run the `cargo fuzz` subcommand with the `nightly` toolchain:

```
cd fuzzing
cargo +nightly fuzz run fuzz_target_1
```

> INFO
>
> If you're developing on MacOS you may need to add the `--sanitizer=thread` flag in order to fix some known linking errors.

You should see output that begins like this:

```
$ cargo +nightly fuzz run fuzz_target_1
   Compiling soroban-fuzzing-contract v0.0.0 (/home/azureuser/data/stellar/soroban-examples/fuzzing)
   Compiling soroban-fuzzing-contract-fuzzer v0.0.0 (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz)
    Finished release [optimized + debuginfo] target(s) in 23.74s
    Finished release [optimized + debuginfo] target(s) in 0.07s
     Running `fuzz/target/x86_64-unknown-linux-gnu/release/fuzz_target_1 ...`
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 886588732
INFO: Loaded 1 modules   (1093478 inline 8-bit counters): 1093478 [0x55eb8e2c7620, 0x55eb8e3d2586),
INFO: Loaded 1 PC tables (1093478 PCs): 1093478 [0x55eb8e3d2588,0x55eb8f481be8),
INFO:      105 files found in /home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/corpus/fuzz_target_1
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: seed corpus: files: 105 min: 32b max: 61b total: 3558b rss: 86Mb
#2      pulse  ft: 8355 exec/s: 1 rss: 307Mb
#4      pulse  cov: 8354 ft: 11014 corp: 1/32b exec/s: 2 rss: 313Mb
#8      pulse  cov: 8495 ft: 12420 corp: 4/128b exec/s: 4 rss: 315Mb
```

The rest of this tutorial will explain how to set up this fuzz test, interpret this output, and remedy fuzzing failures.

Background: Fuzz Testing and Rust

Fuzzing is a kind of testing where new inputs are repeatedly fed into a program in hopes of finding unexpected bugs. This style of testing is commonly employed to increase confidence in the correctness of security-sensitive software.

In Rust, fuzzing is most often performed with the `cargo-fuzz` tool, which drives LLVM's `libfuzzer`, though other fuzzing tools are available.

Soroban has built-in support for fuzzing Soroban contracts with `cargo-fuzz`.

`cargo-fuzz` is a mutation-based fuzzer: it runs a test program, passing it generated input; while the program is executing, the fuzzer monitors which branches the program takes, and which functions it executes; after execution the fuzzer uses this information to make decisions about how to *mutate* the previously-used input to create new input that might discover more branches and functions; it then runs the test again with new input, repeating this process for potentially millions of iterations. In this way `cargo-fuzz` is able to automatically explore execution paths through the program that may never be seen by other types of tests.

If a fuzz tests panics or hard-crashes, `cargo-fuzz` considers it a failure and provides instructions for repeating the test with the failing inputs.

Fuzz testing is typically an exploratory and interactive process, with the programmer devising schemes for producing input that will stress the program in interesting ways, observing the behavior of the fuzz test, and iterating on the test itself.

Resolving a fuzz testing failure typically involves capturing the problematic input in a unit test. The fuzz test itself may or may not be kept, depending on determinations about the cost of maintaining the fuzzer vs the likelihood of it continuing to find bugs in the future.

While fuzzing non-memory-safe software tends to be more lucrative than fuzzing Rust software, it is still relatively common to find panics and other logic errors in Rust through fuzzing.

In Rust, multiple fuzzers are maintained by the `rust-fuzz` GitHub organization, which also maintains a "trophy case" of Rust bugs found through fuzzing.

About the Example

The example used for this tutorial is based on the `timelock` example program, with some changes to demonstrate fuzzing.

The contract, `ClaimableBalanceContract`, allows one party to deposit an arbitrary quantity of a token to the contract, specifying additionally: the `claimants`, addresses that may withdraw from the contract; and the `time_bound`, a specification of when those claimants may withdraw from the account.

The `TimeBound` type looks like

```
#[derive(Clone)]
#[contracttype]
pub struct TimeBound {
    pub kind: TimeBoundKind,
    pub timestamp: u64,
}

#[derive(Clone)]
#[contracttype]
pub enum TimeBoundKind {
    Before,
    After,
}
```

`ClaimableBalanceContract` has two methods, `deposit` and `claim`:

```
    pub fn deposit(
        env: Env,
        from: Address,
        token: Address,
        amount: i128,
        claimants: Vec<Address>,
        time_bound: TimeBound,
    );

    pub fn claim(
        env: Env,
```

`deposit` may only be successfully called once, after which `claim` may be called multiple times until the balance is completely drained, at which point the contract becomes dormant and may no longer be used.

Fuzz Testing Setup

For these examples, the fuzz tests have been created for you, but normally you would use the `cargo fuzz init` command to create a fuzzing project as a subdirectory of the contract under test.

To do that you would navigate to the contract directory, in this case, `soroban-examples/fuzzing`, and execute

```
cargo fuzz init
```

A `cargo-fuzz` project is its own crate, which lives in the `fuzz` subdirectory of the crate being tested. This crate has its own `Cargo.toml` and `Cargo.lock`, and another subdirectory, `fuzz_targets`, which contains Rust programs, each its own fuzz test.

Our `soroban-examples/fuzzing` directory looks like

- `Cargo.toml` - this is the contract's manifest
- `Cargo.lock`
- `src`
  - `lib.rs` - this is the contract code
- `fuzz` - this is the fuzzing crate
  - `Cargo.toml` - this is fuzzing crate's manifest
  - `Cargo.lock`
  - `fuzz_targets`
    - `fuzz_target_1.rs` - this is a single fuzz test
    - `fuzz_target_2.rs`

There are special considerations to note in the configuration of both the contract's manifest and the fuzzing crate's manifest.

Within the contract's manifest one must specificy the crate type as both "cdylib" and "rlib":

```
[package]
name = "soroban-fuzzing-contract"
version = "0.0.0"
authors = ["Stellar Development Foundation <[email protected]>"]
license = "Apache-2.0"
edition = "2021"
publish = false

[lib]
crate-type = ["cdylib", "rlib"]
doctest = false

[features]
testutils = []
```

In most examples, a Soroban contract will only be a "cdylib", a Rust crate that is compiled to a dynamically loadable wasm module. For fuzzing though, the fuzzing crate needs to be able to link to the contract crate as a Rust library, an "rlib".

> NOTE
>
> Note that cargo has a feature/bug that inhibits LTO of cdylibs when a crate is both a "cdylib" and "rlib". This can be worked around by building the contract with either `soroban contract build` or `cargo rustc --crate-type cdylib` instead of the typical `cargo build`.

The contract crate must also provide the "testutils" feature. When "testutils" is activated, the Soroban SDK's `contracttype` macro emits additional code needed for running fuzz tests.

Within the fuzzing crate's manifest one must turn on the "testutils" features in both the contract crate and the `soroban-sdk` crate:

```
[package]
name = "soroban-fuzzing-contract-fuzzer"
version = "0.0.0"
publish = false
edition = "2021"

[package.metadata]
cargo-fuzz = true

[dependencies]
libfuzzer-sys = "0.4"
soroban-sdk = { version = "20.0.0", features = ["testutils"] }

[dependencies.soroban-fuzzing-contract]
path = ".."
features = ["testutils"]
```

A Simple Fuzz Test

First let's look at `fuzz_target_1.rs`. This fuzz test does two things: it first deposits an arbitrary amount, then it claims an arbitrary amount.

Again, you can run this fuzzer from the `soroban-examples/fuzzing` directory with the following command:

```
cargo +nightly fuzz run fuzz_target_1
```

The entry point and setup code for Soroban contract fuzz tests will typically look like:

```
#[derive(Arbitrary, Debug)]
struct Input {
    deposit_amount: i128,
    claim_amount: i128,
}

fuzz_target!(|input: Input| {
    let env = Env::default();

    env.mock_all_auths();

    env.ledger().set(LedgerInfo {
        timestamp: 12345,
        protocol_version: 1,
        sequence_number: 10,
        network_id: Default::default(),
        base_reserve: 10,
    });

    // Turn off the CPU/memory budget for testing.
    env.budget().reset_unlimited();

    // ... do fuzzing here ...
}
```

Instead of a `main` function, `cargo-fuzz` uses a special entry point defined by the `fuzz_target!` macro. This macro accepts a Rust closure that accepts `input`, any Rust type that implements the `Arbitrary` trait. Here we have defined a struct, `Input`, that derives `Arbitrary`.

`cargo-fuzz` will be responsible for generating `input` and repeatedly calling this closure.

To test a Soroban contract, we must set up an `Env`. Note that we have disabled the CPU and memory budget: this will allow us to fuzz arbitrarily complex code paths without worrying about running out of budget; we can assume that running out of budget during a transaction always correctly fails, canceling the transaction; it is not something we need to fuzz.

Refer to the `fuzz_target_1.rs` source code for additional setup for this contract.

This fuzzer performs two steps: deposit, then claim:

```
    // Deposit, then assert invariants.
    {
        let _ = fuzz_catch_panic(|| {
            timelock_client.deposit(
                &depositor_address,
                &token_contract_id,
                &input.deposit_amount,
                &vec![
                    &env,
                    claimant_address.clone(),
                ],
                &TimeBound {
                    kind: TimeBoundKind::Before,
                    timestamp: 123456,
                },
            );
        });

        assert_invariants(
            &env,
            &timelock_contract_id,
            &token_client,
            &input
        );
    }

    // Claim, then assert invariants.
    {
        let _ = fuzz_catch_panic(|| {
            timelock_client.claim(
                &claimant_address,
                &input.claim_amount,
            );
        });

        assert_invariants(
            &env,
            &timelock_contract_id,
            &token_client,
            &input
        );
    }
```

There are a number of potential strategies for writing fuzz tests. The strategy in this test is to make arbitrary, possibly weird and unrealistic, calls to the contract, disregarding whether those calls succeed or fail, and then to make assertions about the state of the contract.

Because there are many potential failure cases for any given contract call, we don't want to write a fuzz test by attempting to interpret the success or failure of any given call: that path leads to duplicating the contract's logic within the fuzz test. Instead we just want to ensure that, regardless of what happened during execution, the contract is never left in an invalid state.

Notice the use of the `fuzz_catch_panic` function to invoke the contract: This is a special function in the Soroban SDK for intercepting panics in a way that works with `cargo-fuzz`, and is needed to call contract functions that might fail. Without `fuzz_catch_panic` a panic from within a contract will immediately cause the fuzz test to fail, but in most cases a panic within a contract does not indicate a bug - it is simply how a Soroban contract cancels a transaction. `fuzz_catch_panic` returns a `Result`, but here we discard it.

Finally, the `assert_invariants` function is where we make any assertions we can about the state of the contract:

```
/// Directly inspect the contract state and make assertions about it.
fn assert_invariants(
    env: &Env,
    timelock_contract_id: &Address,
    token_client: &TokenClient,
    input: &Input,
) {
    // Configure the environment to access the timelock contract's storage.
    env.as_contract(timelock_contract_id, || {
```

Interpreting `cargo-fuzz` Output

If you run `cargo-fuzz` with `fuzz_target_1`, from inside the `soroban-examples/fuzzing` directory, you will see output similar to:

```
$ cargo +nightly fuzz run fuzz_target_1
   Compiling soroban-fuzzing-contract v0.0.0 (/home/azureuser/data/stellar/soroban-examples/fuzzing)
   Compiling soroban-fuzzing-contract-fuzzer v0.0.0 (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz)
    Finished release [optimized + debuginfo] target(s) in 25.18s
    Finished release [optimized + debuginfo] target(s) in 0.08s
     Running `fuzz/target/x86_64-unknown-linux-gnu/release/fuzz_target_1 -artifact_prefix=/home/azureuser/data/stellar/soroban-
examples/fuzzing/fuzz/artifacts/fuzz_target_1/ /home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/corpus/fuzz_target_1`
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1384064486
INFO: Loaded 1 modules   (1122058 inline 8-bit counters): 1122058 [0x561f6ecd4fc0, 0x561f6ede6eca),
INFO: Loaded 1 PC tables (1122058 PCs): 1122058 [0x561f6ede6ed0,0x561f6ff05f70),
INFO:      173 files found in /home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/corpus/fuzz_target_1
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: seed corpus: files: 173 min: 32b max: 61b total: 6039b rss: 83Mb
#4      pulse  cov: 4848 ft: 10214 corp: 1/32b exec/s: 2 rss: 313Mb
#8      pulse  cov: 8507 ft: 11743 corp: 4/128b exec/s: 4 rss: 315Mb
#16     pulse  cov: 8512 ft: 12393 corp: 10/320b exec/s: 8 rss: 319Mb
thread '<unnamed>' panicked at 'assertion failed: claimable_balance.amount > 0', fuzz_targets/fuzz_target_1.rs:130:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
==6102== ERROR: libFuzzer: deadly signal
    #0 0x561f6ae3a431  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x1c80431) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
    #1 0x561f6e3855b0  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x51cb5b0) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
    #2 0x561f6e35c08a  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x51a208a) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
    #3 0x7fce05f5e08f  (/lib/x86_64-linux-gnu/libc.so.6+0x4308f) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)
    #4 0x7fce05f5e00a  (/lib/x86_64-linux-gnu/libc.so.6+0x4300a) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)
    #5 0x7fce05f3d858  (/lib/x86_64-linux-gnu/libc.so.6+0x22858) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)
    ...
    #27 0x561f6e3847b9  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x51ca7b9) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
    #28 0x561f6ad98346  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x1bde346) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
    #29 0x7fce05f3f082  (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)
    #30 0x561f6ad9837d  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x1bde37d) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)

NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better crash reports.
SUMMARY: libFuzzer: deadly signal
MS: 0 ; base unit: 0000000000000000000000000000000000000000
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x5d,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0xff,0x5f,0x5f,0x52,0xff,
\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000]\000\000\000\000\000\000\000\000\377__R\377
artifact_prefix='/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/artifacts/fuzz_target_1/'; Test unit written to /home/
azureuser/data/stellar/soroban-examples/fuzzing/fuzz/artifacts/fuzz_target_1/crash-04704b1542f61a21a4649e39023ec57ff502f627
Base64: AAAAAAAAAAAAAAAAAAAAAAAAXQAAAAAAAAAA/19fUv8=

────────────────────────────────────────────────────────────────────────────────

Failing input:

        fuzz/artifacts/fuzz_target_1/crash-04704b1542f61a21a4649e39023ec57ff502f627

Output of `std::fmt::Debug`:

        Input {
            deposit_amount: 0,
            claim_amount: -901525218878596739118967460911579136,
        }

Reproduce with:

        cargo fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/crash-04704b1542f61a21a4649e39023ec57ff502f627

Minimize test case with:
```

This is a fuzzing failure, indicating a bug in either the fuzzer or the program. The details will be different.

Here is the same output, with less important lines trimmed:

```
thread '<unnamed>' panicked at 'assertion failed: claimable_balance.amount > 0', fuzz_targets/fuzz_target_1.rs:130:13
...
Failing input:

        fuzz/artifacts/fuzz_target_1/crash-04704b1542f61a21a4649e39023ec57ff502f627

Output of `std::fmt::Debug`:

        Input {
            deposit_amount: 0,
            claim_amount: -901525218878596739118967460911579136,
        }

Reproduce with:

        cargo fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/crash-04704b1542f61a21a4649e39023ec57ff502f627

Minimize test case with:

        cargo fuzz tmin fuzz_target_1 fuzz/artifacts/fuzz_target_1/crash-04704b1542f61a21a4649e39023ec57ff502f627
```

The first line here is printed by our Rust program, and indicates exactly where the fuzzer panicked. The later lines indicate how to reproduce this failing case.

The first thing to do when you get a fuzzing failure is copy the command to reproduce the failure, so that you can use it to debug:

```
cargo +nightly fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/crash-04704b1542f61a21a4649e39023ec57ff502f627
```

Notice though that we need to tell `cargo` to use the nightly toolchain with the `+nightly` flag, something that `cargo-fuzz` doesn't print in its version of the command.

Another thing to notice is that by default, `cargo-fuzz` / `libfuzzer` does not print names of functions in its output, as in the stack trace:

```
==6102== ERROR: libFuzzer: deadly signal
    #0 0x561f6ae3a431  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x1c80431) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
    ...
    #28 0x561f6ad98346  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x1bde346) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
    #29 0x7fce05f3f082  (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)
    #30 0x561f6ad9837d  (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/
fuzz_target_1+0x1bde37d) (BuildId: 6a95a932984a405ebab8171dddc9f812fdf16846)
```

Depending on how your system is set up, you may or may not have this problem. In order to print stack traces, `libfuzzer` needs the `llvm-symbolizer` program. On Ubuntu-based systems this can be installed with the `llvm-dev` package:

```
sudo apt install llvm-dev
```

After which `libfuzzer` will print demangled function names instead of addresses:

```
==6323== ERROR: libFuzzer: deadly signal
    #0 0x557c9da6a431 in __sanitizer_print_stack_trace /rustc/llvm/src/llvm-project/compiler-rt/lib/asan/asan_stack.cpp:87:3
    #1 0x557ca0fb55b0 in fuzzer::PrintStackTrace() /home/azureuser/.cargo/registry/src/index.crates.io-6f17d22bba15001f/libfuzzer-
sys-0.4.5/libfuzzer/FuzzerUtil.cpp:210:38
    #2 0x557ca0f8c08a in fuzzer::Fuzzer::CrashCallback() /home/azureuser/.cargo/registry/src/index.crates.io-6f17d22bba15001f/
libfuzzer-sys-0.4.5/libfuzzer/FuzzerLoop.cpp:233:18
    #3 0x557ca0f8c08a in fuzzer::Fuzzer::CrashCallback() /home/azureuser/.cargo/registry/src/index.crates.io-6f17d22bba15001f/
libfuzzer-sys-0.4.5/libfuzzer/FuzzerLoop.cpp:228:6
    #4 0x7ff19e84d08f  (/lib/x86_64-linux-gnu/libc.so.6+0x4308f) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)
```

To continue, our program has a bug that should be easy to fix by inspecting the error and making a slight modification to the source.

Once the bug is fixed, the fuzzer will run continuously, producing output that looks like

```
$ cargo +nightly fuzz run fuzz_target_1
   Compiling soroban-fuzzing-contract v0.0.0 (/home/azureuser/data/stellar/soroban-examples/fuzzing)
   Compiling soroban-fuzzing-contract-fuzzer v0.0.0 (/home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz)
    Finished release [optimized + debuginfo] target(s) in 24.91s
    Finished release [optimized + debuginfo] target(s) in 0.08s
     Running `fuzz/target/x86_64-unknown-linux-gnu/release/fuzz_target_1 -artifact_prefix=/home/azureuser/data/stellar/soroban-
examples/fuzzing/fuzz/artifacts/fuzz_target_1/ /home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/corpus/fuzz_target_1`
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1619748028
INFO: Loaded 1 modules   (1122061 inline 8-bit counters): 1122061 [0x5647a55b9080, 0x5647a56caf8d),
INFO: Loaded 1 PC tables (1122061 PCs): 1122061 [0x5647a56caf90,0x5647a67ea060),
INFO:      173 files found in /home/azureuser/data/stellar/soroban-examples/fuzzing/fuzz/corpus/fuzz_target_1
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: seed corpus: files: 173 min: 32b max: 61b total: 6039b rss: 85Mb
#2      pulse  ft: 8067 exec/s: 1 rss: 312Mb
#4      pulse  cov: 8068 ft: 10709 corp: 1/32b exec/s: 2 rss: 315Mb
#8      pulse  cov: 8476 ft: 11498 corp: 5/160b exec/s: 4 rss: 317Mb
#16     pulse  cov: 8512 ft: 12362 corp: 9/288b exec/s: 8 rss: 320Mb
#32     pulse  cov: 8516 ft: 13290 corp: 19/608b exec/s: 10 rss: 326Mb
#64     pulse  cov: 8516 ft: 13311 corp: 27/864b exec/s: 21 rss: 340Mb
#128    pulse  cov: 8540 ft: 13536 corp: 37/1196b exec/s: 25 rss: 365Mb
#175    INITED cov: 8540 ft: 13580 corp: 42/1387b exec/s: 29 rss: 382Mb
#177    NEW    cov: 8545 ft: 13821 corp: 43/1419b lim: 48 exec/s: 29 rss: 384Mb L: 32/48 MS: 1 ChangeASCIIInt-
#178    NEW    cov: 8545 ft: 13824 corp: 44/1451b lim: 48 exec/s: 29 rss: 384Mb L: 32/48 MS: 1 ChangeBinInt-
#229    NEW    cov: 8545 ft: 13826 corp: 45/1483b lim: 48 exec/s: 38 rss: 401Mb L: 32/48 MS: 1 ChangeByte-
#256    pulse  cov: 8545 ft: 13826 corp: 45/1483b lim: 48 exec/s: 36 rss: 410Mb
#361    NEW    cov: 8545 ft: 13830 corp: 46/1521b lim: 48 exec/s: 40 rss: 451Mb L: 38/48 MS: 5 ShuffleBytes-CMP-EraseBytes-CopyPart-
ChangeBinInt- DE: "\005\000\000\000"-
        NEW_FUNC[1/1]: 0x5647a2964640 in
rand::rngs::adapter::reseeding::ReseedingCore$LT$R$C$Rsdr$GT$::reseed_and_generate::ha760ded93293681c /home/azureuser/.cargo/registry/
src/index.crates.io-6f17d22bba15001f/rand-0.7.3/src/rngs/adapter/reseeding.rs:235
#368    NEW    cov: 8557 ft: 13842 corp: 47/1566b lim: 48 exec/s: 40 rss: 454Mb L: 45/48 MS: 2 CrossOver-InsertRepeatedBytes-
#512    pulse  cov: 8557 ft: 13842 corp: 47/1566b lim: 48 exec/s: 46 rss: 502Mb
#850    NEW    cov: 8557 ft: 13843 corp: 48/1610b lim: 48 exec/s: 53 rss: 591Mb L: 44/48 MS: 2 CopyPart-ChangeBit-
#1024   pulse  cov: 8557 ft: 13843 corp: 48/1610b lim: 48 exec/s: 56 rss: 645Mb
#1796   NEW    cov: 8557 ft: 13863 corp: 49/1642b lim: 53 exec/s: 71 rss: 669Mb L: 32/48 MS: 1 ChangeBinInt-
#1913   NEW    cov: 8557 ft: 13864 corp: 50/1675b lim: 53 exec/s: 73 rss: 669Mb L: 33/48 MS: 2 ShuffleBytes-InsertByte-
#3749   REDUCE cov: 8557 ft: 13864 corp: 50/1670b lim: 68 exec/s: 98 rss: 669Mb L: 39/48 MS: 1 EraseBytes-
...
```

And this output will continue until the fuzzer is killed with `Ctrl-C`.

Next, let's look at a single line of fuzzer output:

```
#177    NEW    cov: 8545 ft: 13821 corp: 43/1419b lim: 48 exec/s: 29 rss: 384Mb L: 32/48 MS: 1 ChangeASCIIInt-
```

The most important column here is `cov`. This is a cumulative measure of branches covered by the fuzzer. When this number stops increasing the fuzzer has probably explored as much of the program as it can. The other columns are described in the [`libfuzzer` documentation](#).

Finally, lets look at this warning:

```
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes.
```

By default, `libfuzzer` only generates input up to 4096 bytes. In a lot of cases, this is probably reasonable, but `cargo-fuzz` can increase the `max_len` by appending the argument after `--`:

```
cargo +nightly fuzz run fuzz_target_1 -- -max_len=20000
```

All the options to libfuzzer can be listed with

```
cargo +nightly fuzz run fuzz_target_1 -- -help=1
```

See the `libfuzzer` documentation for more.

Accepting Soroban Types as Input with the `SorobanArbitrary` Trait

Inputs to the `fuzz_target!` macro must implement the `Arbitrary` trait, which accepts bytes from the fuzzer driver and converts them to Rust values. Soroban types though are managed by the host environment, and so must be created from an `Env` value, which is not available to the fuzzer driver. The `SorobanArbitrary` trait, implemented for all Soroban contract types, exists to bridge this gap: it defines a *prototype* pattern whereby the `fuzz_target` macro creates prototype values that the fuzz program can convert to contract values with the standard soroban conversion traits, `FromVal` or `IntoVal`.

The types of prototypes are identified by the associated type, `SorobanArbitrary::Prototype`:

```
pub trait SorobanArbitrary:
    TryFromVal<Env, Self::Prototype> + IntoVal<Env, Val> + TryFromVal<Env, Val>
{
    type Prototype: for <'a> Arbitrary<'a>;
}
```

Types that implement `SorobanArbitrary` include:

- `i32`, `u32`, `i64`, `u64`, `i128`, `u128`, `I256`, `U256`, `()`, and `bool`,
- `Error`,
- `Bytes`, `BytesN`, `Vec`, `Map`,
- `Address`, `Symbol`,
- `Val`,

All user-defined contract types, those with the `contracttype` attribute, automatically derive `SorobanArbitrary`. Note that `SorobanArbitrary` is only derived when the "testutils" Cargo feature is active. This implies that, in general, to make a Soroban contract fuzzable, the contract crate must define a "testutils" Cargo feature, that feature should turn on the "soroban-sdk/testutils" feature, and the fuzz test, which is its own crate, must turn that feature on.

A More Complex Fuzz Test

The `fuzz_target_2.rs` example, demonstrates the use of `SorobanArbitrary`, the advancement of time, and more advanced fuzzing techniques.

This fuzz test takes a much more complex input, where some of the values are user-defined types exported from the contract under test. This test is structured as a simple interpreter, where the fuzzing harness provides arbitrarily-generated "steps", where each step is either a `deposit` command or a `claim` command. The test then treats each of these steps as a separate transaction: it maintains a snapshot of the blockchain state, and for each step creates a fresh environment in which to execute the contract call, simulating the advancement of time between each step. As in the previous example, assertions are made after each step.

The input to the fuzzer looks, in part, like:

```
#[derive(Arbitrary, Debug)]
struct Input {
    addresses: [<Address as SorobanArbitrary>::Prototype; NUM_ADDRESSES],
    #[arbitrary(with = |u: &mut Unstructured| u.int_in_range(0..=i128::MAX))]
    token_mint: i128,
    steps: RustVec<Step>,
}

#[derive(Arbitrary, Debug)]
```

This shows how to use the `SorobanArbitrary::Prototype` associated type to define inputs to the fuzzer. A Soroban `Address` can only be created with an `Env`, so cannot be generated directly by the `Arbitrary` trait. Instead we use the fully-qualified name of the `Address` prototype, `<Address as SorobanArbitrary>::Prototype`, to ask for `Address`'s prototype instead. Then when our fuzzer needs the `Address` we instantiate it with the `FromVal` trait:

```
let depositor_address = Address::from_val(&env, &input.addresses[cmd.depositor_index]);
```

The contract we are fuzzing is a *timelock* contract, where calculation of time is crucial for correctness. So our testing must account for the advancement of time.

The contract defines a `TimeBound` type and accepts it in the `deposit` method:

```
#[derive(Clone, Debug)]
#[contracttype]
pub struct TimeBound {
    pub kind: TimeBoundKind,
    pub timestamp: u64,
}

#[contractimpl]
impl ClaimableBalanceContract {
    pub fn deposit(
        env: Env,
        from: Address,
        token: Address,
        amount: i128,
        claimants: Vec<Address>,
        time_bound: TimeBound,
    ) {
        ...
    }
}
```

In our fuzzer, one of the possible commands issued each step is a `DepositCommand`:

```
#[derive(Arbitrary, Debug)]
struct DepositCommand {
    #[arbitrary(with = |u: &mut Unstructured| u.int_in_range(0..=NUM_ADDRESSES - 1))]
    depositor_index: usize,
    amount: i128,
    // This is an ugly way to get a vector of integers in range
    #[arbitrary(with = |u: &mut Unstructured| {
        u.arbitrary_len::<usize>().map(|len| {
            (0..len).map(|_| {
                u.int_in_range(0..=NUM_ADDRESSES - 1)
            }).collect::<Result<RustVec<usize>, _>>()
        }).and_then(|inner_result| inner_result)
    })]
    claimant_indexes: RustVec<usize>,
    time_bound: <TimeBound as SorobanArbitrary>::Prototype,
}
```

Notice that this command again uses the `SorobanArbitrary::Prototype` associated type to accept a `TimeBound` as input.

To advance time we maintain a `LedgerSnapshot`, defined in the `soroban-ledger-snapshot` crate. For each step we call `Env::from_snapshot` to create a fresh environment to execute the step, then `Env::to_snapshot` to create a new snapshot to use in the following step.

Here is a simplified outline of how this works. See the full source code for details.

```
let init_snapshot = {
    let init_ledger = LedgerInfo {
        timestamp: 12345,
        protocol_version: 1,
```

Converting a Fuzz Test to a Property Test

In addition to fuzz testing, Soroban supports property testing in the style of quickcheck, by using the `proptest` and `proptest-arbitrary-interop` crates in conjunction with the `SorobanArbitrary` trait.

Property tests are similar to fuzz tests in that they generate randomized input. Property tests though do not instrument their test cases or mutate their input based on feedback from previous tests. Thus they are a weaker form of test.

The great benefit of property tests though is that they can be included in standard Rust test suites and require no extra tooling to execute. One might take advantage of this by interactively fuzzing to discover deep bugs, then convert fuzz tests to property tests to help prevent regressions.

The `proptest.rs` file is a translation of `fuzz_target_1.rs` to a property test.

How-To Guides

The page lists all guides we have available for Soroban. Simply put, a "guide" is a short, bite-sized example that details how to accomplish a specific task. These guides are focused on a single topic, and are limited in scope.

State Archival

Restore a contract using the JavaScript SDK

Restore archived contract data using the JavaScript SDK

Test TTL extension logic in Smart Contracts

Soroban CLI

Deploy a Contract from Installed Wasm Bytecode

Deploy the Stellar Asset Contract for a Stellar Asset

Extend a deployed contract instance's TTL

Extend a deployed contract's storage entry TTL

Extend a deployed contract's Wasm code TTL

Install and Deploy a Smart Contract

Install Wasm Bytecode

Restore an archived contract using the Soroban CLI

Restore archived contract data using the Soroban CLI

Conventions

Organize contract errors with an error enum type

Upgrade the Wasm bytecode of a deployed contract

Write metadata for your contract.

Dapp Development

Use Docker to build and run dapps

Initialize a dapp using scripts

Create a frontend for your dapp using React

Events

Ingest events published from a contract

Publish events from a Rust contract

Freighter Wallet

Connect to the Testnet

Enable Soroban tokens

Integrate Freighter with a React dapp

As a dapp developer, prompt Freighter to sign transactions

Send Soroban token payments

Sign authorization entries

Sign Soroban XDRs

RPC

Generate ledger key parameters with a symbol key using the Python SDK

[Retrieve a contract code ledger entry using the JavaScript SDK]

[Retrieve a contract code ledger entry using the Python SDK]

Storage

[Use instance storage in a contract]

[Use persistent storage in a contract]

[Use temporary storage in a contract]

Testing

[Implement basic tests for a contract]

[Test authorized contract invocations]

Transactions

[Invoke a contract function in a Stellar transaction using JavaScript]

[Submit a transaction to Soroban RPC using the JavaScript SDK]

State Archival

Soroban's novel strategy to combat state bloat can present a learning curve for developers. Here are some quick guides that will help you through the process.

Guides in this category:

Restore a contract using the JavaScript SDK

As you can imagine, if your deployed contract instance or the code that backs it is archived, it can't be loaded to execute your invocations. Remember, there's a distinct, one-to-many relationship on the chain between a contract's code and deployed instances of that contract:

flowchart LR A[my instance] & B[your instance]--> C[contract WASM]
We need **both** to be live for our contract calls to work.

Let's work through how these can be recovered. The recovery process is slightly different for a convenient reason: we don't need simulation to figure out the footprints. Instead, we can leverage `Contract.getFootprint()`, which prepares a footprint with the ledger keys used by a given contract instance (including its backing WASM code).

Unfortunately, we still need simulation to figure out the *fees* for our restoration. This, however, can be easily covered by the SDK's `Server.prepareTransaction()` helper, which will do simulation and assembly for us.

> INFO
>
> This guide makes use of the (aptly named) `yeetTx` function we created in another guide.

```
import {
  BASE_FEE,
  Contract,
  Keypair,
  Networks,
  TransactionBuilder,
  SorobanDataBuilder,
  Operation,
  SorobanRpc,
} from "@stellar/stellar-sdk";

async function restoreContract(
  signer: Keypair,
  c: Contract,
): Promise<SorobanRpc.Api.GetTransactionResponse> {
  const instance = c.getFootprint();

  const account = await server.getAccount(signer.publicKey());
  const wasmEntry = await server.getLedgerEntries(
    getWasmLedgerKey(instance)
  );

  const restoreTx = new TransactionBuilder(account, { fee: BASE_FEE })
    .setNetworkPassphrase(Networks.TESTNET)
    .setSorobanData(
      // Set the restoration footprint (remember, it should be in the
      // read-write part!)
      new SorobanDataBuilder().setReadWrite([
        instance,
        wasmEntry
      ]).build(),
    )
    .addOperation(Operation.restoreFootprint({}))
    .build();

  const preppedTx = await server.prepareTransaction(restoreTx);
  preppedTx.sign(signer);
  return yeetTx(preppedTx);
}

function getWasmLedgerKey(entry: xdr.ContractDataEntry):  {
  return xdr.LedgerKey.contractCode(
    new xdr.LedgerKeyContractCode({
      hash: entry.val().instance().wasmHash()
    })
  );
}
```

Guides in this category:

Restore a contract using the JavaScript SDK

As you can imagine, if your deployed contract instance or the code that backs it is archived, it can't be loaded to execute your invocations. Remember, there's a distinct...

Restore archived contract data using the JavaScript SDK

This is a pretty likely occurrence: my piece of persistent data is archived because I haven't interacted with my contract in a while. How do I make it accessible again?

Test TTL extension logic in Smart Contracts

In order to test contracts that extend the contract data TTL via extendttl storage operations, you can use the TTL getter operation (getttl) in combination with manipulati...

Restore archived contract data using the JavaScript SDK

This is a pretty likely occurrence: my piece of persistent data is archived because I haven't interacted with my contract in a while. How do I make it accessible again?

If you find that a piece of persistent data is archived, it can be restored using a Stellar transaction containing a `RestoreFootprintOp` operation. We'll make two assumptions for the sake of this guide:

- The contract instance itself is still live (i.e., others have been extending its TTL while you've been away).
- You don't know how your archived data is represented on the ledger.

The restoration process we'll use involves three discrete steps:

1. Simulate our transaction as we normally would.
2. If the simulation indicated it, we perform restoration with a `RestoreFootprintOp` operation using the hints we got from the simulation.
3. We retry running our initial transaction.

Here's a function called `submitOrRestoreAndRetry()` that will take care of all those steps for us:

> INFO
>
> This guide makes use of the (aptly named) `yeetTx` function we created in another guide.

```
import {
  BASE_FEE,
  Networks,
  Keypair,
  TransactionBuilder,
  SorobanDataBuilder,
  SorobanRpc,
  xdr,
} from "@stellar/stellar-sdk"; // add'l imports to yeetTx
const { Api, assembleTransaction } = SorobanRpc;

// assume that `server` is the Server() instance from the yeetTx

async function submitOrRestoreAndRetry(
  signer: Keypair,
  tx: Transaction,
): Promise<Api.GetTransactionResponse> {
  // We can't use `prepareTransaction` here because we want to do
  // restoration if necessary, basically assembling the simulation ourselves.
  const sim = await server.simulateTransaction(tx);

  // Other failures are out of scope of this tutorial.
  if (!Api.isSimulationSuccess(sim)) {
    throw sim;
  }

  // If simulation didn't fail, we don't need to restore anything! Just send it.
  if (!Api.isSimulationRestore(sim)) {
    const prepTx = assembleTransaction(tx, sim);
    prepTx.sign(signer);
    return yeetTx(prepTx);
  }

  // Build the restoration operation using the RPC server's hints.
  const account = await server.getAccount(signer.publicKey());
  let fee = parseInt(BASE_FEE);
  fee += parseInt(sim.restorePreamble.minResourceFee);

  const restoreTx = new TransactionBuilder(account, { fee: fee.toString() })
    .setNetworkPassphrase(Networks.TESTNET)
    .setSorobanData(sim.restorePreamble.transactionData.build())
    .addOperation(Operation.restoreFootprint({}))
    .build();
```

Guides in this category:

Restore a contract using the JavaScript SDK

As you can imagine, if your deployed contract instance or the code that backs it is archived, it can't be loaded to execute your invocations. Remember, there's a distinct...

Restore archived contract data using the JavaScript SDK

This is a pretty likely occurrence: my piece of persistent data is archived because I haven't interacted with my contract in a while. How do I make it accessible again?

Test TTL extension logic in Smart Contracts

In order to test contracts that extend the contract data TTL via extendttl storage operations, you can use the TTL getter operation (getttl) in combination with manipulati...

Test TTL extension logic in Smart Contracts

In order to test contracts that extend the contract data TTL via `extend_ttl` storage operations, you can use the TTL getter operation (`get_ttl`) in combination with manipulating the ledger sequence number. Note, that `get_ttl` function is only available for tests and only in Soroban SDK v21+.

Example

Follow along the example that tests TTL extensions. The example has extensive comments, this document just highlights the most important parts.

We use a very simple contract that only extends an entry for every Soroban storage type:

```
#[contractimpl]
impl TtlContract {
    /// Creates a contract entry in every kind of storage.
    pub fn setup(env: Env) {
        env.storage().persistent().set(&DataKey::MyKey, &0);
        env.storage().instance().set(&DataKey::MyKey, &1);
        env.storage().temporary().set(&DataKey::MyKey, &2);
    }

    /// Extend the persistent entry TTL to 5000 ledgers, when its
    /// TTL is smaller than 1000 ledgers.
    pub fn extend_persistent(env: Env) {
        env.storage()
            .persistent()
            .extend_ttl(&DataKey::MyKey, 1000, 5000);
    }

    /// Extend the instance entry TTL to become at least 10000 ledgers,
    /// when its TTL is smaller than 2000 ledgers.
    pub fn extend_instance(env: Env) {
        env.storage().instance().extend_ttl(2000, 10000);
    }

    /// Extend the temporary entry TTL to become at least 7000 ledgers,
    /// when its TTL is smaller than 3000 ledgers.
    pub fn extend_temporary(env: Env) {
        env.storage()
            .temporary()
            .extend_ttl(&DataKey::MyKey, 3000, 7000);
    }
}
```

The focus of the example is the tests, so the following code snippets come from `test.rs`.

It's a good idea to define the custom values of TTL-related network settings, since the defaults are defined by the SDK and aren't immediately obvious for the reader of the tests:

```
env.ledger().with_mut(|li| {
    // Current ledger sequence - the TTL is the number of
    // ledgers from the `sequence_number` (exclusive) until
    // the last ledger sequence where entry is still considered
    // alive.
    li.sequence_number = 100_000;
    // Minimum TTL for persistent entries - new persistent (and instance)
    // entries will have this TTL when created.
    li.min_persistent_entry_ttl = 500;
    // Minimum TTL for temporary entries - new temporary
    // entries will have this TTL when created.
    li.min_temp_entry_ttl = 100;
    // Maximum TTL of any entry. Note, that entries can have their TTL
    // extended indefinitely, but each extension can be at most
    // `max_entry_ttl` ledger from the current `sequence_number`.
    li.max_entry_ttl = 15000;
});
```

You could also use the current network settings when setting up the tests, but keep in mind that these are subject to change, and the contract should be able to work with any values of these settings.

Now we run a test scenario that verifies the TTL extension logic (see `test_extend_ttl_behavior` test for the full scenario). First, we setup the data and ensure that the initial TTL values correspond to the network settings we've defined above:

```
client.setup();
env.as_contract(&contract_id, || {
    // Note, that TTL doesn't include the current ledger, but when entry
    // is created the current ledger is counted towards the number of
    // ledgers specified by `min_persistent/temp_entry_ttl`, thus
    // the TTL is 1 ledger less than the respective setting.
    assert_eq!(env.storage().persistent().get_ttl(&DataKey::MyKey), 499);
    assert_eq!(env.storage().instance().get_ttl(), 499);
    assert_eq!(env.storage().temporary().get_ttl(&DataKey::MyKey), 99);
});
```

Notice, that we use `env.as_contract` in order to access the contract's storage.

Then we call the TTL extension operations and verify that they behave as expected, for example:

```
// Extend persistent entry TTL to 5000 ledgers - now it is 5000.
client.extend_persistent();
env.as_contract(&contract_id, || {
    assert_eq!(env.storage().persistent().get_ttl(&DataKey::MyKey), 5000);
});
```

In order to test the extension thresholds (i.e. maximum current TTL that requires extension), we need to increase the ledger sequence number:

```
// Now bump the ledger sequence by 5000 in order to sanity-check
// the threshold settings of `extend_ttl` operations.
env.ledger().with_mut(|li| {
    li.sequence_number = 100_000 + 5_000;
});
// Now the TTL of every entry has been reduced by 5000 ledgers.
env.as_contract(&contract_id, || {
    assert_eq!(env.storage().persistent().get_ttl(&DataKey::MyKey), 0);
    assert_eq!(env.storage().instance().get_ttl(), 5000);
    assert_eq!(env.storage().temporary().get_ttl(&DataKey::MyKey), 2000);
});
```

Then we can extend the entries again and ensure that only entries that are below threshold have been extended (specifically, persistent and temporary entries in this example):

```
client.extend_persistent();
client.extend_instance();
client.extend_temporary();
env.as_contract(&contract_id, || {
    assert_eq!(env.storage().persistent().get_ttl(&DataKey::MyKey), 5000);
    // Instance TTL hasn't been increased because the remaining TTL
    // (5000 ledgers) is larger than the threshold used by
    // `extend_instance` (2000 ledgers)
    assert_eq!(env.storage().instance().get_ttl(), 5000);
    assert_eq!(env.storage().temporary().get_ttl(&DataKey::MyKey), 7000);
});
```

Soroban SDK also emulates the behavior for the entries that have their TTL expired. Temporary entries behave 'as if' they were deleted (see `test_temp_entry_removal` test for the full scenario):

```
client.extend_temporary();
// Bump the ledger sequence by 7001 ledgers (one ledger past TTL).
env.ledger().with_mut(|li| {
    li.sequence_number = 100_000 + 7001;
});
// Now the entry is no longer present in the environment.
```

Persistent entries are more subtle: when a transaction that is executed on-chain contains a persistent entry that has been archived (i.e. it has it's TTL expired) in the footprint, then the Soroban environment will not even be instantiated. Since this behavior is not directly reproducible in test environment, instead an irrecoverable 'internal' error will be produced as soon as an archived entry is accessed, and the test will `panic`:

```
#[test]
#[should_panic(expected = "[testing-only] Accessed contract instance key that has been archived.")]
fn test_persistent_entry_archival() {
    let env = create_env();
    let contract_id = env.register_contract(None, TtlContract);
    let client = TtlContractClient::new(&env, &contract_id);
    client.setup();
    // Extend the instance TTL to 10000 ledgers.
    client.extend_instance();
    // Bump the ledger sequence by 10001 ledgers (one ledger past TTL).
    env.ledger().with_mut(|li| {
        li.sequence_number = 100_000 + 10_001;
    });
    // Now any call involving the expired contract (such as `extend_instance`
    // call here) will panic as soon as that contract is accessed.
    client.extend_instance();
}
```

Testing TTL extension for other contract instances

Sometimes a contract may want to extend TTL of another contracts and/or their Wasm entries (usually that would happen in factory contracts). This logic may be covered in a similar fashion to the example above using `env.deployer().get_contract_instance_ttl(&contract)` to get TTL of any contract's instance, and `env.deployer().get_contract_code_ttl(&contract)` to get TTL of any contract's Wasm entry. You can find an example of using these function in the SDK test suite.

Guides in this category:

Restore a contract using the JavaScript SDK

As you can imagine, if your deployed contract instance or the code that backs it is archived, it can't be loaded to execute your invocations. Remember, there's a distinct...

Restore archived contract data using the JavaScript SDK

This is a pretty likely occurrence: my piece of persistent data is archived because I haven't interacted with my contract in a while. How do I make it accessible again?

Test TTL extension logic in Smart Contracts

In order to test contracts that extend the contract data TTL via extendttl storage operations, you can use the TTL getter operation (getttl) in combination with manipulati...

Soroban CLI

The Soroban CLI is a crucial tool for developers to use while creating and interacting with Soroban smart contracts.

Guides in this category:

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Deploy a Contract from Installed Wasm Bytecode

To deploy an instance of a compiled smart contract that has already been isntalled onto the Stellar network, use the `soroban contract deploy` command:

```
soroban contract deploy \
    --source S... \
    --network testnet \
    --wasm-hash <hex-encoded-wasm-hash>
```

Guides in this category:

Deploy a Contract from Installed Wasm Bytecode

To deploy an instance of a compiled smart contract that has already been isntalled onto the Stellar network, use the soroban contract deploy command:

Deploy the Stellar Asset Contract for a Stellar Asset

The Soroban CLI can deploy a [Stellar Asset Contract] for a Stellar asset so that any Soroban contract can interact with the asset.

Extend a deployed contract instance's TTL

You can use the Soroban CLI to extend the TTL of a contract instance like so:

Extend a deployed contract's storage entry TTL

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple Symbol as its storage key, you can run a ...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Deploy the Stellar Asset Contract for a Stellar Asset

The Soroban CLI can deploy a Stellar Asset Contract for a Stellar asset so that any Soroban contract can interact with the asset.

Every Stellar asset has reserved a contract that anyone can deploy. Once deployed any contract can interact with that asset by holding a balance of the asset, receiving the asset, or sending the asset.

Deploying the Stellar Asset Contract for a Stellar asset enables that asset for use on Soroban.

The Stellar Asset Contract can be deployed for any possible Stellar asset, either assets already in use on Stellar or assets that have never seen any activity. This means that the issuer doesn't need to have been created, and no one needs to be yet holding the asset on Stellar.

To perform the deploy, use the following command:

```
soroban contract asset deploy \
    --source S... \
    --network testnet \
    --asset USDC:GCYEIQEWOCTTSA72VPZ6LYIZIK4W4KNGJR72UADIXUXG45VDFRVCQTYE
```

The `asset` argument corresponds to the symbol and it's issuer address, which is how assets are identified on Stellar.

The same can be done for the native Lumens asset:

```
soroban contract asset deploy \
    --source S... \
    --network testnet \
    --asset native
```

> NOTE
>
> Deploying the native asset will fail on testnet or mainnet as a Stellar Asset Contract already exists.

For any asset, the contract address can be fetched with:

```
soroban contract id asset \
    --source S... \
    --network testnet \
    --asset native
```

Guides in this category:

Deploy a Contract from Installed Wasm Bytecode

To deploy an instance of a compiled smart contract that has already been isntalled onto the Stellar network, use the soroban contract deploy command:

Deploy the Stellar Asset Contract for a Stellar Asset

The Soroban CLI can deploy a [Stellar Asset Contract] for a Stellar asset so that any Soroban contract can interact with the asset.

Extend a deployed contract instance's TTL

You can use the Soroban CLI to extend the TTL of a contract instance like so:

Extend a deployed contract's storage entry TTL

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple Symbol as its storage key, you can run a ...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Extend a deployed contract instance's TTL

You can use the Soroban CLI to extend the TTL of a contract instance like so:

```
soroban contract extend \
    --source S... \
    --network testnet \
    --id C... \
    --ledgers-to-extend 535679 \
    --durability persistent
```

This example uses 535,679 ledgers as the new archival TTL. This is the maximum allowable value for this argument on the CLI. This corresponds to roughly 30 days (averaging 5 second ledger close times).

When you extend a contract instance, this includes:

- the contract instance itself
- any `env.storage().instance()` entries in the contract
- the contract's Wasm code

Guides in this category:

Deploy a Contract from Installed Wasm Bytecode

To deploy an instance of a compiled smart contract that has already been isntalled onto the Stellar network, use the soroban contract deploy command:

Deploy the Stellar Asset Contract for a Stellar Asset

The Soroban CLI can deploy a [Stellar Asset Contract] for a Stellar asset so that any Soroban contract can interact with the asset.

Extend a deployed contract instance's TTL

You can use the Soroban CLI to extend the TTL of a contract instance like so:

Extend a deployed contract's storage entry TTL

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple Symbol as its storage key, you can run a ...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Extend a deployed contract's storage entry TTL

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple `Symbol` as its storage key, you can run a command like so:

```
soroban contract extend \
    --source S... \
    --network testnet \
    --id C... \
    --key COUNTER \
    --ledgers-to-extend 535679 \
    --durability persistent
```

This example uses 535,679 ledgers as the new archival TTL. This is the maximum allowable value for this argument on the CLI. This corresponds to roughly 30 days (averaging 5 second ledger close times).

If your storage entry uses a more advanced storage key, such as `Balance(Address)` in a token contract, you'll need to provide the key in a base64-encoded XDR form:

```
soroban contract extend \
    --source S... \
    --network testnet \
    --id C... \
    --key-xdr AAAABgAAAAHXkotywnA8z+r365/0701QSlWouXn8m0UOoshCtNHOYQAAAA4AAAAHQmFsYW5jZQAAAAAB \
    --ledgers-to-extend 535679 \
    --durability persistent
```

> INFO
>
> Be sure to check out our guide on creating XDR ledger keys for help generating them.

Guides in this category:

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple Symbol as its storage key, you can run a ...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms: if you do or do not have the compiled contract locally. If you do have the compiled binary on your local machine:

```
soroban contract extend \
    --source S... \
    --network testnet \
    --wasm ../relative/path/to/soroban_contract.wasm \
    --ledgers-to-extend 535679 \
    --durability persistent
```

This example uses 535,679 ledgers as the new archival TTL. This is the maximum allowable value for this argument on the CLI. This corresponds to roughly 30 days (averaging 5 second ledger close times).

If you do not have the compiled binary on your local machine, you can still use the CLI to extend the bytecode TTL. You'll need to know the Wasm hash of the installed contract code:

```
soroban contract extend \
    --source S... \
    --network testnet \
    --wasm-hash <hex-encoded-wasm-hash> \
    --ledgers-to-extend 535679 \
    --durability persistent
```

> INFO
>
> You can learn more about finding the correct Wasm hash for a contract instance here (JavaScript) and here (Python).

Guides in this category:

Deploy a Contract from Installed Wasm Bytecode

To deploy an instance of a compiled smart contract that has already been isntalled onto the Stellar network, use the soroban contract deploy command:

Deploy the Stellar Asset Contract for a Stellar Asset

The Soroban CLI can deploy a [Stellar Asset Contract] for a Stellar asset so that any Soroban contract can interact with the asset.

Extend a deployed contract instance's TTL

You can use the Soroban CLI to extend the TTL of a contract instance like so:

Extend a deployed contract's storage entry TTL

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple Symbol as its storage key, you can run a ...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Install and Deploy a Smart Contract

You can combine the `install` and `deploy` commands of the Soroban CLI to accomplish both tasks:

```
soroban contract deploy \
    --source S... \
    --network testnet \
    --wasm ../relative/path/to/soroban_contract.wasm
```

Guides in this category:

Deploy a Contract from Installed Wasm Bytecode

To deploy an instance of a compiled smart contract that has already been isntalled onto the Stellar network, use the soroban contract deploy command:

Deploy the Stellar Asset Contract for a Stellar Asset

The Soroban CLI can deploy a [Stellar Asset Contract] for a Stellar asset so that any Soroban contract can interact with the asset.

Extend a deployed contract instance's TTL

You can use the Soroban CLI to extend the TTL of a contract instance like so:

Extend a deployed contract's storage entry TTL

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple Symbol as its storage key, you can run a ...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the `soroban contract install` command:

```
soroban contract install \
    --source S... \
    --network testnet \
    --wasm ../relative/path/to/soroban_contract.wasm
```

NOTE

Note this command will return the hash ID of the Wasm bytecode, rather than an address for a contract instance.

Guides in this category:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

```
soroban contract restore \
    --source S... \
    --network testnet \
    --id C... \
    --durability persistent
```

Guides in this category:

Deploy a Contract from Installed Wasm Bytecode

To deploy an instance of a compiled smart contract that has already been isntalled onto the Stellar network, use the soroban contract deploy command:

Deploy the Stellar Asset Contract for a Stellar Asset

The Soroban CLI can deploy a [Stellar Asset Contract] for a Stellar asset so that any Soroban contract can interact with the asset.

Extend a deployed contract instance's TTL

You can use the Soroban CLI to extend the TTL of a contract instance like so:

Extend a deployed contract's storage entry TTL

You can use the Soroban CLI to extend the TTL of a contract's persistent storage entry. For a storage entry that uses a simple Symbol as its storage key, you can run a ...

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple `Symbol` as its storage key, you can run a command like so:

```
soroban contract restore \
    --source S... \
    --network testnet \
    --id C... \
    --key COUNTER \
    --durability persistent
```

If your storage entry uses a more advanced storage key, such as `Balance(Address)` in a token contract, you'll need to provide the key in a base64-encoded XDR form:

```
soroban contract restore \
    --source S... \
    --network testnet \
    --id C... \
    --key-xdr AAAABgAAAAHXkotywnA8z+r365/0701QSlWouXn8m0UOoshCtNHOYQAAAA4AAAAHQmFsYW5jZQAAAAAB \
    --durability persistent
```

> INFO
>
> Be sure to check out our guide on creating XDR ledger keys for help generating them.

Guides in this category:

Extend a deployed contract's Wasm code TTL

You can use the Soroban CLI to extend the TTL of a contract's Wasm bytecode. This can be done in two forms

Install and Deploy a Smart Contract

You can combine the install and deploy commands of the Soroban CLI to accomplish both tasks:

Install Wasm Bytecode

To use the Soroban CLI to install a compiled smart contract on the ledger, use the soroban contract install command:

Restore an archived contract using the Soroban CLI

If your contract instance has been archived, it can easily be restored using the Soroban CLI.

Restore archived contract data using the Soroban CLI

If a contract's persistent storage entry has been archived, you can restore it using the Soroban CLI. For a storage entry that uses a simple Symbol as its storage key, yo...

Conventions

These guides describe the "typical" way something might be accomplished in a Rust contract. These guides aren't meant to be quite as *prescriptive* as some others; instead, they serve to highlight some of the norms we've seen when crop up in contract development.

Guides in this category:

Organize contract errors with an error enum type

A convenient way to manage and meaningfully communicate contract errors is to collect them into an enum struct. These errors are a special type of enum integer type...

Upgrade the Wasm bytecode of a deployed contract

Upgrade the Wasm bytecode of a deployed contract

Write metadata for your contract.

Write structured metadata.

Organize contract errors with an error enum type

A convenient way to manage and meaningfully communicate contract errors is to collect them into an `enum` struct. These errors are a special type of enum integer type that are stored on ledger as Status values containing a `u32` code. First, create the `Error` struct in your smart contract.

```
#[contracterror]
#[derive(Copy, Clone, Debug, Eq, PartialEq, PartialOrd, Ord)]
#[repr(u32)]
pub enum Error {
    FirstError = 1,
    AnotherError = 2,
    YetAnotherError = 3,
    GenericError = 4
}
```

Then, panic with an error when the conditions are met. This example will panic with the specified error.

```
#[contractimpl]
impl Contract {
    pub fn causeerror(env: Env, error_code: u32) -> Result<(), Error> {
        match error_code {
            1 => Err(Error::FirstError),
            2 => Err(Error::AnotherError),
            3 => Err(Error::YetAnotherError),
            _ => Err(Error::GenericError),
        }
    }
}
```

When converted to XDR, the value becomes an `ScVal`, containing a `ScStatus`, containing the integer value of the error as contract error.

```
{ "status": { "contractError": 1 } }
```

Guides in this category:

Upgrade the Wasm bytecode of a deployed contract

The upgradeable contract example demonstrates how to upgrade a Wasm contract.

Open in Gitpod

Code

The example contains both an "old" and "new" contract, where we upgrade from "old" to "new". The code below is for the "old" contract.

upgradeable_contract/old_contract/src/lib.rs

```
#![no_std]

use soroban_sdk::{contractimpl, contracttype, Address, BytesN, Env};

#[contracttype]
#[derive(Clone)]
enum DataKey {
    Admin,
}

#[contract]
pub struct UpgradeableContract;

#[contractimpl]
impl UpgradeableContract {
    pub fn init(e: Env, admin: Address) {
        e.storage().instance().set(&DataKey::Admin, &admin);
    }

    pub fn version() -> u32 {
        1
    }

    pub fn upgrade(e: Env, new_wasm_hash: BytesN<32>) {
        let admin: Address = e.storage().instance().get(&DataKey::Admin).unwrap();
        admin.require_auth();

        e.deployer().update_current_contract_wasm(new_wasm_hash);
    }
}
```

How it works

The upgrade is only possible because the contract calls `e.update_current_contract_wasm`, with the wasm hash of the new contract as a parameter. The contract ID does **not** change. Note that the contract required authorization from an admin before upgrading. This is to prevent anyone from upgrading the contract. You can read more about the `update_current_contract_wasm` function in the Soroban Rust SDK here.

```
pub fn upgrade(e: Env, new_wasm_hash: BytesN<32>) {
    let admin: Address = e.storage().instance().get(&DataKey::Admin).unwrap();
    admin.require_auth();

    e.deployer().update_current_contract_wasm(new_wasm_hash);
}
```

The `update_current_contract_wasm` host function will also emit a `SYSTEM` contract event that contains the old and new wasm reference, allowing downstream users to be notified when a contract they use is updated. The event structure will have `topics = ["executable_update",` `old_executable: ContractExecutable, old_executable: ContractExecutable]` and `data = []`.

Tests

Open the `upgradeable_contract/old_contract/src/test.rs` file to follow along.

upgradeable_contract/old_contract/srctest.rs

```rust
#![cfg(test)]

use soroban_sdk::{testutils::Address as _, Address, BytesN, Env};

mod old_contract {
    soroban_sdk::contractimport!(
        file =
            "target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_old_contract.wasm"
    );
}

mod new_contract {
    soroban_sdk::contractimport!(
        file = "../new_contract/target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_new_contract.wasm"
    );
}

fn install_new_wasm(e: &Env) -> BytesN<32> {
    e.install_contract_wasm(new_contract::Wasm)
}

#[test]
fn test() {
    let env = Env::default();
    env.mock_all_auths();

    // Note that we use register_contract_wasm instead of register_contract
    // because the old contracts Wasm is expected to exist in storage.
    let contract_id = env.register_contract_wasm(None, old_contract::Wasm);

    let client = old_contract::Client::new(&env, &contract_id);
    let admin = Address::random(&env);
    client.init(&admin);

    assert_eq!(1, client.version());

    let new_wasm_hash = install_new_wasm(&env);

    client.upgrade(&new_wasm_hash);
    assert_eq!(2, client.version());

    // new_v2_fn was added in the new contract, so the existing
    // client is out of date. Generate a new one.
    let client = new_contract::Client::new(&env, &contract_id);
    assert_eq!(1010101, client.new_v2_fn());
}
```

We first import wasm files for both contracts -

```rust
mod old_contract {
    soroban_sdk::contractimport!(
        file =
            "target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_old_contract.wasm"
    );
}

mod new_contract {
    soroban_sdk::contractimport!(
        file = "../new_contract/target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_new_contract.wasm"
    );
}
```

We register the old contract, intialize it with an admin, and verify the version it reutrns. The note in the code below is important-

```
// Note that we use register_contract_wasm instead of register_contract
// because the old contracts Wasm is expected to exist in storage.
let contract_id = env.register_contract_wasm(None, old_contract::Wasm);

let client = old_contract::Client::new(&env, &contract_id);
let admin = Address::random(&env);
client.init(&admin);

assert_eq!(1, client.version());
```

We install the new contract's Wasm

```
let new_wasm_hash = install_new_wasm(&env);
```

Then we run the upgrade, and verify that the upgrade worked.

```
client.upgrade(&new_wasm_hash);
assert_eq!(2, client.version());
```

Build the Contract

To build the contract `.wasm` files, run `soroban contract build` in both `upgradeable_contract/old_contract` and `upgradeable_contract/new_contract` in that order.

Both `.wasm` files should be found in both contract `target` directories after building both contracts:

```
target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_old_contract.wasm
```

```
target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_new_contract.wasm
```

Run the Contract

If you have `soroban-cli` installed, you can invoke contract functions. Deploy the old contract and install the wasm for the new contract.

```
soroban contract deploy \
    --wasm target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_old_contract.wasm \
    --id a
```

```
soroban contract install \
    --wasm target/wasm32-unknown-unknown/release/soroban_upgradeable_contract_new_contract.wasm
```

You should see this Wasm hash from the install command:

```
c30c71a382438ed7e56669ba172aa862cc813d093b8d2f45e85b47ba38a89ddc
```

You also need to call the `init` method so the `admin` is set. This requires us to setup som identities.

```
soroban keys generate acc1 && \
soroban keys address acc1
```

Example output:

```
GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B
```

Now call `init` with this key (make sure to substitute with the key you generated).

```
soroban contract invoke \
    --id a \
    -- \
    init \
    --admin GAJGHZ44IJXYFNOVRZGBCVKC2V62DB2KHZB7BEMYOWOLFQH4XP2TAM6B
```

Invoke the `version` function.

```
soroban contract invoke \
    --id a \
    -- \
    version
```

The following output should occur using the code above.

```
1
```

Now upgrade the contract. Notice the `--source` must be the identity name matching the address passed to the `init` function.

```
soroban contract invoke \
    --source acc1 \
    --id a \
    -- \
    upgrade \
    --new_wasm_hash c30c71a382438ed7e56669ba172aa862cc813d093b8d2f45e85b47ba38a89ddc
```

Invoke the `version` function again.

```
soroban contract invoke \
    --id a \
    -- \
    version
```

Now that the contract was upgraded, you'll see a new version.

```
2
```

Guides in this category:

Organize contract errors with an error enum type

A convenient way to manage and meaningfully communicate contract errors is to collect them into an enum struct. These errors are a special type of enum integer type...

Upgrade the Wasm bytecode of a deployed contract

Upgrade the Wasm bytecode of a deployed contract

Write metadata for your contract.

Write structured metadata.

Write metadata for your contract.

The `contractmeta!` macro provided in the Rust SDK allows users to write two strings - a `key` and a `val` - within a serialized `SCMetaEntry::SCMetaV0` XDR object to the custom section of Wasm contracts. The section name for this metadata is `contractmetav0`. Developers can utilize this macro to write metadata, and tools can then read and display this information to users.

The liquidity pool example provides a clear demonstration of how to use the `contractmeta!` macro:

```
// Metadata that is added on to the Wasm custom section
contractmeta!(
    key = "Description",
    val = "Constant product AMM with a .3% swap fee"
);

pub trait LiquidityPoolTrait {...
```

Guides in this category:

Type Conversions

A collection of guides for converting from one data type to another in a variety of SDK languages.

Guides in this category:

State Archival

Restore a contract using the JavaScript SDK

Restore archived contract data using the JavaScript SDK

Test TTL extension logic in Smart Contracts

Soroban CLI

Deploy a Contract from Installed Wasm Bytecode

Deploy the Stellar Asset Contract for a Stellar Asset

Extend a deployed contract instance's TTL

Extend a deployed contract's storage entry TTL

Extend a deployed contract's Wasm code TTL

Install and Deploy a Smart Contract

Install Wasm Bytecode

Restore an archived contract using the Soroban CLI

Restore archived contract data using the Soroban CLI

Conventions

Organize contract errors with an error enum type

Upgrade the Wasm bytecode of a deployed contract

Write metadata for your contract.

Dapp Development

Use Docker to build and run dapps

Initialize a dapp using scripts

Create a frontend for your dapp using React

Events

Ingest events published from a contract

Publish events from a Rust contract

Freighter Wallet

Connect to the Testnet

Enable Soroban tokens

Integrate Freighter with a React dapp

As a dapp developer, prompt Freighter to sign transactions

Send Soroban token payments

Sign authorization entries

Sign Soroban XDRs

RPC

Generate ledger key parameters with a symbol key using the Python SDK

Retrieve a contract code ledger entry using the JavaScript SDK

Retrieve a contract code ledger entry using the Python SDK

Storage

Use instance storage in a contract

Use persistent storage in a contract

Use temporary storage in a contract

Testing

Implement basic tests for a contract

Test authorized contract invocations

Transactions

Invoke a contract function in a Stellar transaction using JavaScript

Submit a transaction to Soroban RPC using the JavaScript SDK

Dapp Development

We've written some helpful guides on some of the most useful tools available to you, the dapp developer.

Guides in this category:

Use Docker to build and run dapps

What is Docker?

Initialize a dapp using scripts

When setting up an example Soroban Dapp, correct initialization is crucial. This process entails several steps, including deploying Docker, cloning and deploying smart ...

Create a frontend for your dapp using React

This section elaborates on how the frontends from your dapp can interact with the example contracts and access chain data, and connect to a freighter wallet. This will ...

Use Docker to build and run dapps

What is Docker?

Welcome to the world of Docker, an essential tool for software development. Docker packages software into units known as containers, ensuring consistency, isolation, portability, and scalability.

Docker is particularly useful in dapp development. It helps manage microservices, maintain consistent environments throughout development stages, and simulate a decentralized network during testing.

Understanding Docker begins with understanding Docker images and containers. A Docker image, created from a Dockerfile, is a package that contains everything needed to run the software. A Docker container is a running instance of this image.

Building and Running a Docker Image

You can create a Docker image using the docker build command with a Dockerfile. Once the image is created, you can run a Docker container using the docker run command.

In the context of the example Soroban dapps, understanding how to build Docker images is crucial. The Docker images serve as the basis for our container, which provides the environment for our dapp to run.

Here's an example from our example

To illustrate the process, let's take an example from our example crowdfund dapp. In order to build the Docker image, you utilize a command that is encapsulated within our Makefile:

```
make build-docker
```

This command simplifies the Docker build process and ensures it's consistently executed each time. When you run `make build-docker`, Docker executes the following instructions:

```
docker build . \
 --tag soroban-preview:11 \
 --force-rm \
 --rm
```

Makefile Overview

```
docker build .
```

Instructs Docker to build an image using the Dockerfile in the current directory (denoted by the ".").

```
--tag soroban-preview:11
```

Gives a name and tag to our image, in this case, soroban-preview with the tag 9.

```
--force-rm
```

Ensures Docker removes any intermediate containers after the build process completes. This keeps our environment clean.

```
--rm
```

Guarantees the removal of the intermediate container, even if the build fails. By using `make build-docker`, you're harnessing the power of Docker to create a consistent, reliable environment for our dapp.

Container Deployment

You can streamline the deployment process by using a script to run the Docker container. The following script is a wrapper for the `stellar/quickstart` Docker image, which provides a quick way to run a Stellar network. You can find an example of the `quickstart.sh` script located in the root directory of the example crowdfund dapp.

quickstart.sh

```bash
#!/bin/bash

set -e

case "$1" in
standalone)
    echo "Using standalone network"
    ARGS="--standalone"
    ;;
futurenet)
    echo "Using Futurenet network"
    ARGS="--futurenet"
    ;;
*)
    echo "Usage: $0 standalone|futurenet"
    exit 1
    ;;
esac

shift

# Run the soroban-preview container
# Remember to do:
# make build-docker

echo "Creating docker soroban network"
(docker network inspect soroban-network -f '{{.Id}}' 2>/dev/null) \
  || docker network create soroban-network

echo "Searching for a previous soroban-preview docker container"
containerID=$(docker ps --filter="name=soroban-preview" --all --quiet)
if [[ ${containerID} ]]; then
    echo "Start removing soroban-preview container."
    docker rm --force soroban-preview
    echo "Finished removing soroban-preview container."
else
    echo "No previous soroban-preview container was found"
fi

currentDir=$(pwd)
docker run -dti \
  --volume ${currentDir}:/workspace \
  --name soroban-preview \
  -p 8001:8000 \
  --ipc=host \
  --network soroban-network \
  soroban-preview:11

# Run the stellar quickstart image

docker run --rm -ti \
  --name stellar \
  --network soroban-network \
  -p 8000:8000 \
  stellar/quickstart:testing \
  $ARGS \
  --enable-soroban-rpc \
  "$@" # Pass through args from the CLI
```

The `quickstart.sh` script sets up the Docker environment for running the dapp. It allows you to choose between a standalone network or the

Futurenet network. The script performs the following steps:

- Determines the network based on the provided argument (`standalone` or `futurenet`).
- Creates the Docker network named `soroban-network` if it doesn't exist.
- Removes any existing `soroban-preview` Docker container.
- Runs the `soroban-preview` container, which provides the Soroban Preview environment for development.
- Runs the `stellar/quickstart` Docker image, which sets up the Stellar network using the chosen network type and enables Soroban RPC.

Guides in this category:

Use Docker to build and run dapps

What is Docker?

Initialize a dapp using scripts

When setting up an example Soroban Dapp, correct initialization is crucial. This process entails several steps, including deploying Docker, cloning and deploying smart ...

Create a frontend for your dapp using React

This section elaborates on how the frontends from your dapp can interact with the example contracts and access chain data, and connect to a freighter wallet. This will ...

Initialize a dapp using scripts

When setting up an example Soroban Dapp, correct initialization is crucial. This process entails several steps, including deploying Docker, cloning and deploying smart contracts, and invoking functions to configure them. In this comprehensive guide, you will walk you through the necessary steps to successfully build and deploy these smart contracts, ensuring a seamless setup for your Soroban Dapp.

Building and Deploying the Soroban Token Smart Contract

In dapps like the Example Payment Dapp, the Soroban Token smart contracts are used to represent the tokenized asset that users can send and receive. Here is an example of how to build and deploy the Soroban Token smart contracts:

Start by cloning the Soroban examples repository:

```
git clone https://github.com/stellar/soroban-examples.git
```

Then, navigate to the `token` directory:

```
cd soroban-examples/token
```

At this point you can build the smart contract:

```
make
```

This action will compile the smart contracts and place them in the `token/target/wasm32-unknown-unknown/release` directory.

After building, you're ready to deploy the smart contracts to Futurenet. To do this, open a terminal in the `soroban-examples/token` directory and execute the following:

```
soroban contract deploy \
 --wasm target/wasm32-unknown-unknown/release/soroban_token_contract.wasm \
 --source <ADMIN_ACCOUNT_SECRET_KEY> \
 --rpc-url https://rpc-futurenet.stellar.org:443 \
 --network-passphrase 'Test SDF Future Network ; October 2022'
```

This command deploys the smart contracts to Futurenet using the `soroban contract deploy` function.

Initializing a Token Contract

With the contracts deployed, it's time to initialize the token contract:

```
soroban contract invoke \
 --id <TOKEN_CONTRACT_ID> \
 --source-account <ADMIN_ACCOUNT_SECRET_KEY> \
 --rpc-url https://rpc-futurenet.stellar.org:443 \
 --network-passphrase 'Test SDF Future Network ; October 2022' \
 -- initialize \
 --admin <ADMIN_PUBLIC_KEY> \
 --decimal 7 \
 --name '44656d6f20546f6b656e' \
 --symbol '"4454"'
```

This command requires certain inputs:

- Administrator Account: This is the public key of the administrator account. The administrator has control and authority over the token contract, enabling management of various contract functionalities. Learn more about the administrator's role from the Soroban Token

Interface.

- Decimal Precision: The decimal precision value of 7 specifies that the token can support transactions up to 7 decimal places. This precision level enables flexibility when transferring token amounts.
- Token Name: The token's name, represented as a hex-encoded string. In this case, '44656d6f20546f6b656e' corresponds to "Demo Token".
- Token Symbol: This is the token's symbol, also represented as a hex string. '4454' translates to the symbol "DT".

Minting Tokens

Lastly, you need to mint some tokens to the sender's account:

```
soroban contract invoke \
 --id <TOKEN_CONTRACT_ID> \
 --source-account <ADMIN_ACCOUNT_SECRET_KEY> \
 --rpc-url https://rpc-futurenet.stellar.org:443 \
 --network-passphrase 'Test SDF Future Network ; October 2022' \
 -- mint \
 --to <USER_PUBLIC_KEY> \
 --amount 1000000000
```

This command will mint 100 tokens to the designated user's account.

By following these steps, you ensure that the Soroban token smart contracts are correctly deployed and initialized, setting the stage for the Dapp to effectively interact with the token.

For a deeper dive into Soroban CLI commands, check out the Soroban CLI repo.

Automating Initialization with Scripts

To streamline the initialization process, you can use a script. This script should automate various tasks such as setting up the network, wrapping Stellar assets, generating token-admin identities, funding the token-admin account, building and deploying the contracts, and initializing them with necessary parameters.

Here's an example initializer script:

initialize.sh

```
#!/bin/bash

set -e

NETWORK="$1"

# If soroban-cli is called inside the soroban-preview docker container,
# it can call the stellar standalone container just using its name "stellar"
if [[ "$IS_USING_DOCKER" == "true" ]]; then
  SOROBAN_RPC_HOST="http://stellar:8000"
else
  SOROBAN_RPC_HOST="http://localhost:8000"
fi

case "$1" in
standalone)
  echo "Using standalone network"
  SOROBAN_NETWORK_PASSPHRASE="Standalone Network ; February 2017"
  FRIENDBOT_URL="$SOROBAN_RPC_HOST/friendbot"
  SOROBAN_RPC_URL="$SOROBAN_RPC_HOST/soroban/rpc"
  ;;
futurenet)
  echo "Using Futurenet network"
```

Here's a summary of what the `initialize.sh` script does:

- Identifies the network (standalone or futurenet) based on user input
- Determines the Soroban RPC host URL depending on its execution environment (either inside the soroban-preview Docker container or locally)
- Sets the Soroban RPC URL based on the previously determined host URL
- Sets the Soroban network passphrase and Friendbot URL depending on the chosen network
- Adds the network configuration to Soroban using `soroban network add`
- Generates a token-admin identity using `soroban keys generate`
- Fetches the TOKEN_ADMIN_SECRET and TOKEN_ADMIN_ADDRESS from the newly generated identity
- Saves the TOKEN_ADMIN_SECRET and TOKEN_ADMIN_ADDRESS in the .soroban directory
- Funds the token-admin account using Friendbot
- Deploy the Stellar asset contract with `soroban contract asset deploy` and stores the resulting TOKEN_ID
- Builds the crowdfund contract with `make build` and deploys it using `soroban contract deploy`, storing the returned CROWDFUND_ID
- Initializes the crowdfund contract by invoking the initialize function with necessary parameters
- Prints "Done" to signify the end of the initialization process

By leveraging automated initialization, you can streamline the setup process for your Soroban Dapp, ensuring it is correctly deployed and initialized.

Guides in this category:

Use Docker to build and run dapps

What is Docker?

Initialize a dapp using scripts

When setting up an example Soroban Dapp, correct initialization is crucial. This process entails several steps, including deploying Docker, cloning and deploying smart ...

Create a frontend for your dapp using React

This section elaborates on how the frontends from your dapp can interact with the example contracts and access chain data, and connect to a freighter wallet. This will ...

Create a frontend for your dapp using React

This section elaborates on how the frontends from your dapp can interact with the example contracts and access chain data, and connect to a freighter wallet. This will be illustrated by utilizing libraries provided by `@soroban-react`, a simple, powerful framework for building modern Dapps using React. `@soroban-react` was created and is maintained by an amazing member of the community!

> INFO
>
> This guide will demonstrate how an example crowdfund dapp frontend was developed with React. While much of the code is specific to this project, the principles demonstrated should be educational enough to get you started.

Below is a list of the libraries used throughout the frontend code and their respective imports:

```
import { SorobanReactProvider } from "@soroban-react/core";
import { testnet, sandbox, standalone } from "@soroban-react/chains";
import { freighter } from "@soroban-react/freighter";
import { ChainMetadata, Connector } from "@soroban-react/types";
import type {
  WalletChain,
  ChainMetadata,
  ChainName,
} from "@soroban-react/types";
import { useSorobanReact } from "@soroban-react/core";
```

These imports include `SorobanReactProvider` from `@soroban-react/core`, which is a context provider used to pass the SorobanReact instance to other components. You also import several types such as `WalletChain`, `ChainMetadata`, and `ChainName`, which help to maintain type safety within our application.

React Components and Prop Passing

React thrives on its component-based architecture. Components are reusable pieces of code that return a React element to be rendered on the page. A typical React application consists of multiple components working harmoniously to create a dynamic user interface.

Let's look at a component from the the example crowdfund dapp, the `MintButton` component:

```
function MintButton({
  account,
  decimals,
  symbol,
}: {
  account: string;
  decimals: number;
  symbol: string;
}) {
  const [isSubmitting, setSubmitting] = useState(false);
  const { activeChain, server } = useNetwork();
  const networkPassphrase = activeChain?.networkPassphrase ?? "";
  const { sendTransaction } = useSendTransaction();
  const amount = BigNumber(100);

  return <Button props omitted here />;
}
```

This functional component takes three properties as arguments: `account`, `decimals`, and `symbol`. It demonstrates the concept of prop passing, a way to pass data from parent to child components in React. The `onComplete` prop even allows you to pass functions to your copmonents as props. We also see React's `useState` hook for local state management, a method to preserve values between function calls.

State Management and Hooks

State management is another core concept of React, allowing components to create and manage their own data. The `useState` hook is a feature introduced in React 16.8 that allows functional components to have their own state.

In the `MintButton` component, the `useState` hook is used to manage the `isSubmitting` state:

```
const [isSubmitting, setSubmitting] = useState(false);
```

The `useState` hook returns a pair of values: the current state and a function that updates it. In this case, the `isSubmitting` state is initialized to `false` and the `setSubmitting` function is used to update it. React also allows for the creation of custom hooks, like `useNetwork` and `useSendTransaction`, for encapsulating and reusing stateful logic across multiple components.

Custom Hooks

React hooks are functions that let you “hook into” React state and lifecycle features from functional components. Custom hooks allow you to encapsulate complex logic and make it reusable across components. Let's take a look at `useNetwork` and `useSendTransaction`, two custom hooks used in the example crowdfund dapp.

The `useNetwork` hook is utilized to interact with the blockchain network, and the `useSendTransaction` hook is used to dispatch transactions. These hooks abstract away complex logic, making it easier to read and understand the main component code.

Here's how you use these hooks in the `MintButton` component:

```
const { activeChain, server } = useNetwork();
const networkPassphrase = activeChain?.networkPassphrase ?? "";
const { sendTransaction } = useSendTransaction();
```

`useNetwork` provides the active chain and the server, and `useSendTransaction` gives us the `sendTransaction` method, which you'll later use to mint tokens. This way, you can keep the component focused on rendering and event handling logic, making it easier to test and maintain.

Asynchronous Processing and Robust Error Handling

When dealing with operations that might take an unpredictable amount of time, like network requests or, in our case, minting tokens on the blockchain, React's support for asynchronous operations is crucial. This allows the execution of the rest of the code without being blocked by these operations.

Let's dive into the code snippet that handles the asynchronous minting process:

```
try {
  console.log("Minting the token...");
  const paymentResult = await sendTransaction(
    new SorobanClient.TransactionBuilder(adminSource, {
      networkPassphrase,
      fee: "1000",
    })
      .setTimeout(10)
      .addOperation(
        SorobanClient.Operation.payment({
          destination: walletSource.accountId(),
          asset: new SorobanClient.Asset(symbol, Constants.TokenAdmin),
          amount: amount.toString(),
        }),
      )
      .build(),
    {
      timeout: 10 * 1000,
      skipAddingFootprint: true,
      secretKey: Constants.TokenAdminSecretKey,
      sorobanContext,
    },
  );
  console.debug(paymentResult);
  sorobanContext.connect();
```

This block is where the actual token minting occurs. It's wrapped in a `try-catch` block, ensuring that any errors during the minting process are caught and handled appropriately, preventing the application from crashing and giving you a chance to provide feedback to the user.

The `await` keyword pauses the execution of the function until the promise returned by `sendTransaction` resolves. `sendTransaction` is a function obtained from our `useSendTransaction` hook, and it builds and sends a payment operation to the Stellar network.

The `sendTransaction` method accepts two arguments: a `TransactionBuilder` instance and an options object. The `TransactionBuilder` sets up the details of the transaction, such as the source account, network passphrase, transaction fee, and operations to be performed—in this case, a payment operation.

If the transaction is successful, `paymentResult` contains the result, which you log for debugging purposes. If an error occurs during the transaction, the function throws an error, which you catch and log.

Conclusion

React offers a host of high-level concepts that can drastically improve your web development process. By understanding and utilizing these concepts—such as components, prop passing, state management, asynchronous operations, and error handling—you can create scalable, maintainable, and efficient applications.

Remember, the key to mastering React is practice. So, keep building and experimenting!

Guides in this category:

📄 Use Docker to build and run dapps

What is Docker?

📄 Initialize a dapp using scripts

When setting up an example Soroban Dapp, correct initialization is crucial. This process entails several steps, including deploying Docker, cloning and deploying smart ...

📄 Create a frontend for your dapp using React

This section elaborates on how the frontends from your dapp can interact with the example contracts and access chain data, and connect to a freighter wallet. This will ...

Events

Learn how to emit, ingest, and use events published from a Soroban smart contract.

Guides in this category:

Ingest events published from a contract

Soroban RPC provides a getEvents method which allows you to query events from a smart contract. However, the data retention window for these events is roughly 24 ...

Publish events from a Rust contract

An event can contain up to 4 topics, alongside the data it is publishing. The data can be any value or type you want. However, the topics must not contain:

Ingest events published from a contract

Soroban RPC provides a `getEvents` method which allows you to query events from a smart contract. However, the data retention window for these events is roughly 24 hours. If you need access to a longer-lived record of these events you'll want to "ingest" the events as they are published, maintaining your own record or database as events are ingested.

There are many strategies you can use to ingest and keep the events published by a smart contract. Among the simplest might be using a community-developed tool such as Mercury which will take all the infrastructure work off your plate for a low subscription fee.

Another approach we'll explore here is using a cron job to query Soroban RPC periodically and store the relevant events in a locally stored SQLite database, using Prisma as a database abstraction layer. By using Prisma here, it should be relatively trivial to scale this approach up to any other database software of your choosing.

Setup the Database Client

The finer details of choosing a Prisma configuration are beyond the scope of this document. You can get a lot more information in the Prisma quickstart. Here is our Prisma schema's model:

```
model SorobanEvent {
  id          String @id
  type        String
  ledger      Int
  contract_id String
  topic_1     String?
  topic_2     String?
  topic_3     String?
  topic_4     String?
  value       String
}
```

We'll use this model to create and query for the events stored in our database.

Query Events from Soroban RPC

First, we'll need to query the events from Soroban RPC. This simple JavaScript example will use the `@stellar/stellar-sdk` library to make an RPC request using the `getEvents` method, filtering for all `transfer` events that are emitted by the native XLM contract.

> NOTE
>
> We are making some assumptions here. We'll assume that your contract sees enough activity, and that you are querying for events frequently enough that you aren't in danger of needing to figure out the oldest ledger Soroban RPC is aware of. The approach we're taking is to find the largest (most recent) ledger sequence number in the database and query for events starting there. Your use-case may require some logic to determine what the latest ledger is, and what the oldest ledger available is, etc.

```
import { SorobanRpc } from "@stellar/stellar-sdk";
import { PrismaClient } from "@prisma/client";

const server = new SorobanRpc.Server("https://soroban-testnet.stellar.org");
const prisma = new PrismaClient();

let latestEventIngested = await prisma.sorobanEvent.findFirst({
  orderBy: [
    {
      ledger: "desc",
    },
  ],
});

let events = await server.getEvents({
  startLedger: latestEventIngested.ledger,
```

Store Events in the Database

Now, we'll check if the `events` object contains any new events we should store, and we do exactly that. We're storing the event's topics and values as base64-encoded strings here, but you could decode the necessary topics and values into the appropriate data types for your use-case.

```
if (events.events?.length) {
  events.events.forEach(async (event) => {
    await prisma.sorobanEvent.create({
      data: {
        id: event.id,
        type: event.type,
        ledger: event.ledger,
        contract_id: event.contractId.toString(),
        topic_1: event.topic[0].toXDR("base64") || null,
        topic_2: event.topic[1].toXDR("base64") || null,
        topic_3: event.topic[2].toXDR("base64") || null,
        topic_4: event.topic[3].toXDR("base64") || null,
        value: event.value.toXDR("base64"),
      },
    });
  });
}
```

Run the Script with Cron

A cron entry is an excellent way to automate this script to gather and ingest events every so often. You could configure this script to run as (in)frequently as you want or need. This example would run the script every 24 hours at 1:14 pm:

```
14 13 * * * node /absolute/path/to/script.js
```

Here's another example that will run the script every 30 minutes:

```
30 * * * * node /absolute/path/to/script.js
```

Guides in this category:

Ingest events published from a contract

Soroban RPC provides a getEvents method which allows you to query events from a smart contract. However, the data retention window for these events is roughly 24 ...

Publish events from a Rust contract

An event can contain up to 4 topics, alongside the data it is publishing. The data can be any value or type you want. However, the topics must not contain:

Publish events from a Rust contract

An event can contain up to 4 topics, alongside the data it is publishing. The `data` can be any value or type you want. However, the topics must not contain:

- `Vec`
- `Map`
- `Bytes` or `BytesN` longer than 32 bytes
- `contracttype`

```rust
// This function does nothing beside publishing events. Topics we are using are
// some `u32` integers for the sake of simplicity here.
pub fn events_function(env: Env) {
    // A symbol will be our `data` we want published
    my_data = Symbol::new(&env, "data_to_publish");

    // an event with 0 topics
    env.events().publish((), my_data.clone());

    // an event with 1 topic (Notice the extra comma after the topic in the
    // tuple? That comma is required in Rust to make a one-element tuple)
    env.events().publish((1u32,), my_data.clone());

    // an event with 2 topics
    env.events().publish((1u32, 2u32), my_data.clone());

    // an event with 3 topics
    env.events().publish((1u32, 2u32, 3u32), my_data.clone());

    // an event with 4 topics
    env.events().publish((1u32, 2u32, 3u32, 4u32), my_data.clone());
}
```

A more realistic example can be found in the way the token interface works. For example, the interface requires an event to be published every time the `transfer` function is invoked, with the following information:

```rust
pub fn transfer(env: Env, from: Address, to: Address, amount: i128) {
    // transfer logic omitted here
    env.events().publish(
        (symbol_short!("transfer"), from, to),
        amount
    );
}
```

Guides in this category:

Ingest events published from a contract

Soroban RPC provides a getEvents method which allows you to query events from a smart contract. However, the data retention window for these events is roughly 24 ...

Publish events from a Rust contract

An event can contain up to 4 topics, alongside the data it is publishing. The data can be any value or type you want. However, the topics must not contain:

Fees & Metering

Fees and metering in Soroban smart contracts work differently than the fees for "regular" Stellar transactions. The Stellar network still provides cheap, accessible transaction and that now includes smart contract metering!

Guides in this category:

State Archival

Soroban CLI

Conventions

Organize contract errors with an error enum type

Upgrade the Wasm bytecode of a deployed contract

Write metadata for your contract.

Dapp Development

Use Docker to build and run dapps

Initialize a dapp using scripts

Create a frontend for your dapp using React

Events

Ingest events published from a contract

Publish events from a Rust contract

Freighter Wallet

Connect to the Testnet

Enable Soroban tokens

Integrate Freighter with a React dapp

As a dapp developer, prompt Freighter to sign transactions

Send Soroban token payments

Sign authorization entries

Sign Soroban XDRs

RPC

Generate ledger key parameters with a symbol key using the Python SDK

Retrieve a contract code ledger entry using the JavaScript SDK

Retrieve a contract code ledger entry using the Python SDK

Storage

Use instance storage in a contract

Use persistent storage in a contract

Use temporary storage in a contract

Testing

Implement basic tests for a contract

Test authorized contract invocations

Transactions

Invoke a contract function in a Stellar transaction using JavaScript

Submit a transaction to Soroban RPC using the JavaScript SDK

Freighter Wallet

Freighter is a browser extension wallet provided by the Stellar Development Foundation. It provides users a way to interact with Soroban tokens directly from the web browser.

Guides in this category:

Connect to the Testnet

1. Install the Freighter browser extension.

Enable Soroban tokens

With a funded Stellar account, you can now add Soroban tokens to your Freighter wallet.

Integrate Freighter with a React dapp

Wallets are an essential part of any dapp. They allow users to interact with the blockchain and sign transactions. In this section, you'll learn how to integrate the Freight...

As a dapp developer, prompt Freighter to sign transactions

If you're building a JS dapp, easily sign Soroban transactions using the Freighter browser extension and its corresponding client library @stellar/freighter-api:

Send Soroban token payments

Once you have added a Soroban token to your Freighter wallet, you can now send a payment of that token directly from Freighter.

Sign authorization entries

In order to take advantage of contract authorization, you can use Freighter's API to sign an authorization entry. A good example of how signing an authorization entry w...

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

Connect to the Testnet

1. Install the Freighter browser extension.
2. Create a keypair or import an existing account using a mnemonic phrase to complete setup.
3. Next, switch to Testnet. Testnet is available from the network dropdown.
4. If your account does not exist on the selected network, Freighter will prompt you to fund it the account using Friendbot. Alternatively, you can do so in the Stellar Laboratory.

Guides in this category:

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

Enable Soroban tokens

With a funded Stellar account, you can now add Soroban tokens to your Freighter wallet.

1. On the Freighter account screen, click this `Manage Assets` button at the bottom of the screen.
2. You will now see a button to `Add Soroban token` at the bottom of the screen. Click this `Add Soroban token` button.
3. On the next screen, enter the Token ID of the token you want to add to Freighter and click `Add New Token`.
4. You will now see your token's balance on Freighter's account page. Clicking on the balance will show a history of payments sent using this token.

Guides in this category:

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

Integrate Freighter with a React dapp

Wallets are an essential part of any dapp. They allow users to interact with the blockchain and sign transactions. In this section, you'll learn how to integrate the Freighter wallet into your React dapps.

WalletData Component

In the example crowdfund dapp, the `WalletData` component plays a key role in wallet integration. Let's break down the code and understand its functionality:

/components/moleculres/wallet-data/index.tsx

```
import React from "react";
import { useAccount, useIsMounted } from "../../../hooks";
import { ConnectButton } from "../../atoms";
import styles from "./style.module.css";

export function WalletData() {
  const mounted = useIsMounted();
  const account = useAccount();

  return (
    <>
      {mounted && account ? (
        <div className={styles.displayData}>
          <div className={styles.card}>{account.displayName}</div>
        </div>
      ) : (
        <ConnectButton label="Connect Wallet" />
      )}
    </>
  );
}
```

Here's a breakdown of the code:

- The `mounted` variable is obtained using the `useIsMounted` hook, indicating whether the component is currently mounted or not.
- The `useAccount` hook is used to fetch the user's account data, and the `data` property is destructured from the result.
- Conditional rendering is used to display different content based on the component's mount status and the availability of account data.
- If the component is mounted and the account data is available, the user's wallet data is displayed. This includes the account's display name.
- If the component is not mounted or the account data is not available, a `ConnectButton` component is rendered, allowing the user to connect with Freighter.

Guides in this category:

📄 Connect to the Testnet

1. Install the Freighter browser extension.

📄 Enable Soroban tokens

With a funded Stellar account, you can now add Soroban tokens to your Freighter wallet.

Integrate Freighter with a React dapp

Wallets are an essential part of any dapp. They allow users to interact with the blockchain and sign transactions. In this section, you'll learn how to integrate the Freight...

As a dapp developer, prompt Freighter to sign transactions

If you're building a JS dapp, easily sign Soroban transactions using the Freighter browser extension and its corresponding client library @stellar/freighter-api:

Send Soroban token payments

Once you have added a Soroban token to your Freighter wallet, you can now send a payment of that token directly from Freighter.

Sign authorization entries

In order to take advantage of contract authorization, you can use Freighter's API to sign an authorization entry. A good example of how signing an authorization entry w...

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

As a dapp developer, prompt Freighter to sign transactions

If you're building a JS dapp, easily sign Soroban transactions using the Freighter browser extension and its corresponding client library @stellar/freighter-api:

1. Follow the setup instructions to connect to the Testnet, if required during development.
2. Now, you can use the `signTransaction` method from `@stellar/freighter-api` in your dapp to sign Soroban XDRs using the account in Freighter.
3. Upon calling `signTransaction`, Freighter will open and prompt the user to sign the transaction. Approving the transaction will return the signed XDR to the requesting dapp.

Guides in this category:

Connect to the Testnet

1. Install the Freighter browser extension.

Enable Soroban tokens

With a funded Stellar account, you can now add Soroban tokens to your Freighter wallet.

Integrate Freighter with a React dapp

Wallets are an essential part of any dapp. They allow users to interact with the blockchain and sign transactions. In this section, you'll learn how to integrate the Freight...

As a dapp developer, prompt Freighter to sign transactions

If you're building a JS dapp, easily sign Soroban transactions using the Freighter browser extension and its corresponding client library @stellar/freighter-api:

Send Soroban token payments

Once you have added a Soroban token to your Freighter wallet, you can now send a payment of that token directly from Freighter.

Sign authorization entries

In order to take advantage of contract authorization, you can use Freighter's API to sign an authorization entry. A good example of how signing an authorization entry w...

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

Send Soroban token payments

Once you have added a Soroban token to your Freighter wallet, you can now send a payment of that token directly from Freighter.

1. On the Freighter account screen, click the `Send Payment` icon in the upper right of the screen.
2. Enter a recipient public key. Click `Continue`.
3. Select your token from the asset dropdown at the bottom of the screen and enter a token amount. Click `Continue`.
4. Enter a memo (optional). Click `Review Send`.
5. Review the details of your payment. Click `Send`.

Guides in this category:

Connect to the Testnet

1. Install the Freighter browser extension.

Enable Soroban tokens

With a funded Stellar account, you can now add Soroban tokens to your Freighter wallet.

Integrate Freighter with a React dapp

Wallets are an essential part of any dapp. They allow users to interact with the blockchain and sign transactions. In this section, you'll learn how to integrate the Freight...

As a dapp developer, prompt Freighter to sign transactions

If you're building a JS dapp, easily sign Soroban transactions using the Freighter browser extension and its corresponding client library @stellar/freighter-api:

Send Soroban token payments

Once you have added a Soroban token to your Freighter wallet, you can now send a payment of that token directly from Freighter.

Sign authorization entries

In order to take advantage of contract authorization, you can use Freighter's API to sign an authorization entry. A good example of how signing an authorization entry w...

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

Sign authorization entries

In order to take advantage of contract authorization, you can use Freighter's API to sign an authorization entry. A good example of how signing an authorization entry works can be found in the `authorizeEntry` helper of `stellar-sdk`.

Like in the helper, you can construct a `HashIdPreimageSorobanAuthorization` and use the xdr representation of that structure to call `await freighterApi.signAuthEntry(preimageXdr)`. This call will return a `Buffer` of the signed hash of the `HashIdPreimageSorobanAuthorization` passed in, which can then be used to submit to the network during a contract authorization workflow.

For a full example of how to use contract authorization, the scaffold-soroban demo for an atomic swap makes use of both contract auth and Freighter's `signAuthEntry` API.

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

1. On the Lab's transaction signer, enter a Soroban XDR into the form field.
2. Click `Sign with Freighter`.
3. Freighter will open with the details of the XDR. Click `Approve` to sign or `Reject` to dismiss without a signature.
4. If approved, Freighter will transmit a signed XDR back to the Lab.

Guides in this category:

Connect to the Testnet

1. Install the Freighter browser extension.

Enable Soroban tokens

With a funded Stellar account, you can now add Soroban tokens to your Freighter wallet.

Integrate Freighter with a React dapp

Wallets are an essential part of any dapp. They allow users to interact with the blockchain and sign transactions. In this section, you'll learn how to integrate the Freight...

As a dapp developer, prompt Freighter to sign transactions

If you're building a JS dapp, easily sign Soroban transactions using the Freighter browser extension and its corresponding client library @stellar/freighter-api:

Send Soroban token payments

Once you have added a Soroban token to your Freighter wallet, you can now send a payment of that token directly from Freighter.

Sign authorization entries

In order to take advantage of contract authorization, you can use Freighter's API to sign an authorization entry. A good example of how signing an authorization entry w...

Sign Soroban XDRs

With a funded Testnet account, you can now sign Soroban XDRs using dApps that are integrated with Freighter. An example of an integrated dApp is Stellar's Laboratory.

RPC

Using and interacting with Soroban RPC is an important part of the smart contract development lifecycle.

Guides in this category:

Generate ledger key parameters with a symbol key using the Python SDK

In the [increment example contract] stores an integer value in a ledger entry that is identified by a key with the symbol COUNTER. The value of this ledger key can be d...

Retrieve a contract code ledger entry using the JavaScript SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a LedgerEntry containing the Wasm byte-code, which is u...

Retrieve a contract code ledger entry using the Python SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a LedgerEntry containing the Wasm byte-code, which is u...

Generate ledger key parameters with a symbol key using the Python SDK

In the `increment` example contract stores an integer value in a ledger entry that is identified by a key with the symbol `COUNTER`. The value of this ledger key can be derived using the following code snippets.

```python
from stellar_sdk import xdr, scval, Address

def get_ledger_key_symbol(contract_id: str, symbol_text: str) -> str:
    ledger_key = xdr.LedgerKey(
        type=xdr.LedgerEntryType.CONTRACT_DATA,
        contract_data=xdr.LedgerKeyContractData(
            contract=Address(contract_id).to_xdr_sc_address(),
            key=scval.to_symbol(symbol_text),
            durability=xdr.ContractDataDurability.PERSISTENT
        ),
    )
    return ledger_key.to_xdr()

print(
    get_ledger_key_symbol(
        "CCPYZFKEAXHHS5VVW5J45TOU7S2EODJ7TZNJIA5LKDVL3PESCES6FNCI",
        "COUNTER"
    )
)
```

Guides in this category:

Retrieve a contract code ledger entry using the JavaScript SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a `LedgerEntry` containing the Wasm byte-code, which is uniquely identified by its hash (that is, the hash of the uploaded code itself). Then, when the contract is "deployed," we create a `LedgerEntry` with a reference to that code's hash. So fetching the contract code is a two-step process:

1. First, we look up the contract itself, to see which code hash it is referencing.
2. Then, we can look up the raw Wasm byte-code using that hash.

```
import { Contract } from "@stellar/stellar-sdk";

function getLedgerKeyContractCode(contractId) {
  const instance = new Contract(contractId).getFootprint();
  return instance.toXDR("base64");
}

console.log(
  getLedgerKeyContractCode(
    "CCPYZFKEAXHHS5VVW5J45TOU7S2EODJ7TZNJIA5LKDVL3PESCES6FNCI",
  ),
);
// OUTPUT: AAAABgAAAAGfjJVEBc55drW3U87N1Py0Rw0/nlqUA6tQ6r28khEl4gAAABQAAAAB
```

We then take our output from this function, and use it as the element in the `keys` array parameter in our call to the `getLedgerEntries` method.

```
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "method": "getLedgerEntries",
  "params": {
    "keys": ["AAAABgAAAAGfjJVEBc55drW3U87N1Py0Rw0/nlqUA6tQ6r28khEl4gAAABQAAAAB"]
  }
}
```

And the response we get contains the `LedgerEntryData` that can be used to find the `hash` we must use to request the Wasm byte-code. This hash is the `LedgerKey` that's been associated with the deployed contract code.

```
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "result": {
    "entries": [
      {
        "key": "AAAABgAAAAGfjJVEBc55drW3U87N1Py0Rw0/nlqUA6tQ6r28khEl4gAAABQAAAAB",
        "xdr": "AAAABgAAAAAAAABn4yVRAX0eXa1t1POzdT8tEcNP55alAOrUOq9vJIRJeIAAAAUAAAAAQAAABMAAAAA5DNtbckOGVRsNVb8L7X/
lIhAOy2o5G6GkLKXvc7W8foAAAAA",
        "lastModifiedLedgerSeq": 261603
      }
    ],
    "latestLedger": 262322
  }
}
```

Now take the `xdr` field from the previous response's `result` object, and create a `LedgerKey` from the hash contained inside.

```
import { xdr } from "@stellar/stellar-sdk";

function getLedgerKeyWasmId(contractCodeLedgerEntryData) {
  const entry = xdr.LedgerEntryData.fromXDR(
    contractCodeLedgerEntryData,
    "base64",
  );
```

Now, finally we have a `LedgerKey` that correspond to the Wasm byte-code that has been deployed under the `ContractId` we started out with so very long ago. This `LedgerKey` can be used in a final request to the Soroban-RPC endpoint.

```json
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "method": "getLedgerEntries",
  "params": {
    "keys": ["AAAAB+QzbW3JDhlUbDVW/C+1/5SIQDstqORuhpCyl73O1vH6"]
  }
}
```

And the response we get contains (even more) `LedgerEntryData` that we can decode and parse to get the actual, deployed, real-life contract byte-code. We'll leave that exercise up to you. You can check out what is contained using the "View XDR" page of the Stellar Laboratory.

```json
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "result": {
    "entries": [
      {
        "key": "AAAAB+QzbW3JDhlUbDVW/C+1/5SIQDstqORuhpCyl73O1vH6",
        "xdr": "AAAABwAAAADkM21tyQ4ZVGw1Vvwvtf+UiEA7LajkboaQspe9ztbx+gAAAkgAYXNtAQAAAAEVBGACfn4BfmADfn5+AX5gAAF+YAAAAhkEAWwBMAAAAWwBMQAAAWwBXwABAWwBOAAAAwUEAgM
AYNCBFINASABQiCIpyEACyAAQQFqIgBFDQFCjrrQr4bUOSAArUIghkIEhCIBQgEQgoCAgAAaQoSAgICgBkKEgICAwAwQg4CAgAAaIAEPCwAACxCFgICAAAALCQAQhoCAgAAACwQAAAALAgALAHMOY29udHJhY3R
        "lastModifiedLedgerSeq": 368441,
        "liveUntilLedgerSeq": 2442040
      }
    ],
    "latestLedger": 370940
  }
}
```

Guides in this category:

Generate ledger key parameters with a symbol key using the Python SDK

In the [increment example contract] stores an integer value in a ledger entry that is identified by a key with the symbol COUNTER. The value of this ledger key can be d...

Retrieve a contract code ledger entry using the JavaScript SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a LedgerEntry containing the Wasm byte-code, which is u...

Retrieve a contract code ledger entry using the Python SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a LedgerEntry containing the Wasm byte-code, which is u...

Retrieve a contract code ledger entry using the Python SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a `LedgerEntry` containing the Wasm byte-code, which is uniquely identified by its hash (that is, the hash of the uploaded code itself). Then, when the contract is "deployed," we create a `LedgerEntry` with a reference to that code's hash. So fetching the contract code is a two-step process:

1. First, we look up the contract itself, to see which code hash it is referencing.
2. Then, we can look up the raw Wasm byte-code using that hash.

```python
from stellar_sdk import xdr, Address

def get_ledger_key_contract_code(contract_id: str) -> str:
    ledger_key = xdr.LedgerKey(
        type=xdr.LedgerEntryType.CONTRACT_DATA,
        contract_data=xdr.LedgerKeyContractData(
            contract=Address(contract_id).to_xdr_sc_address(),
            key=xdr.SCVal(xdr.SCValType.SCV_LEDGER_KEY_CONTRACT_INSTANCE),
            durability=xdr.ContractDataDurability.PERSISTENT
        )
    )
    return ledger_key.to_xdr()

print(
    get_ledger_key_contract_code(
        "CCPYZFKEAXHHS5VVW5J45TOU7S2EODJ7TZNJIA5LKDVL3PESCES6FNCI"
    )
)
# OUTPUT: AAAABgAAAAGfjJVEBc55drW3U87N1Py0Rw0/nlqUA6tQ6r28khEl4gAAABQAAAAB
```

We then take our output from this function, and use it as the element in the `keys` array parameter in our call to the `getLedgerEntries` method.


```json
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "method": "getLedgerEntries",
  "params": {
    "keys": ["AAAABgAAAAGfjJVEBc55drW3U87N1Py0Rw0/nlqUA6tQ6r28khEl4gAAABQAAAAB"]
  }
}
```


And the response we get contains the `LedgerEntryData` that can be used to find the `hash` we must use to request the Wasm byte-code. This hash is the `LedgerKey` that's been associated with the deployed contract code.


```json
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "result": {
    "entries": [
      {
        "key": "AAAABgAAAAGfjJVEBc55drW3U87N1Py0Rw0/nlqUA6tQ6r28khEl4gAAABQAAAAB",
        "xdr": "AAAABgAAAAAAAABn4yVRAXOeXa1t1POzdT8tEcNP55alAOrUOq9vJIRJeIAAAAUAAAAAQAAABMAAAAA5DNtbckOGVRsNVb8L7X/
lIhAOy2o5G6GkLKXvc7W8foAAAAA",
        "lastModifiedLedgerSeq": 261603
      }
    ],
    "latestLedger": 262322
  }
}
```


Now take the `xdr` field from the previous response's `result` object, and create a `LedgerKey` from the hash contained inside.

```python
from stellar_sdk import xdr
```

Now, finally we have a `LedgerKey` that correspond to the Wasm byte-code that has been deployed under the `ContractId` we started out with so very long ago. This `LedgerKey` can be used in a final request to the Soroban-RPC endpoint.

```
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "method": "getLedgerEntries",
  "params": {
    "keys": ["AAAAB+QzbW3JDhlUbDVW/C+1/5SIQDstqORuhpCyl73O1vH6"]
  }
}
```

And the response we get contains (even more) `LedgerEntryData` that we can decode and parse to get the actual, deployed, real-life contract byte-code. We'll leave that exercise up to you. You can check out what is contained using the "View XDR" page of the Stellar Laboratory.

```
{
  "jsonrpc": "2.0",
  "id": 8675309,
  "result": {
    "entries": [
      {
        "key": "AAAAB+QzbW3JDhlUbDVW/C+1/5SIQDstqORuhpCyl73O1vH6",
        "xdr": "AAAABwAAAADkM21tyQ4ZVGw1Vvwvtf+UiEA7LajkboaQspe9ztbx+gAAAkgAYXNtAQAAAAEVBGACfn4BfmADfn5+AX5gAAF+YAAAAhkEAWwBMAAAAWwBMQAAAWwBXwABAWwBOAAAAwUEAgM
AYNCBFINASABQiCIpyEACyAAQQFqIgBFDQFCjrrQr4bUOSAArUIghkIEhCIBQgEQgoCAgAAaQoSAgICgBkKEgICAwAwQg4CAgAAaIAEPCwAACxCFgICAAAALCQAQhoCAgAAACwQAAAALAgALAHMOY29udHJhY3R
        "lastModifiedLedgerSeq": 368441,
        "liveUntilLedgerSeq": 2442040
      }
    ],
    "latestLedger": 370940
  }
}
```

Guides in this category:

Generate ledger key parameters with a symbol key using the Python SDK

In the [increment example contract] stores an integer value in a ledger entry that is identified by a key with the symbol COUNTER. The value of this ledger key can be d...

Retrieve a contract code ledger entry using the JavaScript SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a LedgerEntry containing the Wasm byte-code, which is u...

Retrieve a contract code ledger entry using the Python SDK

When you deploy a contract, first the code is "installed" (i.e., it is uploaded onto the blockchain). This creates a LedgerEntry containing the Wasm byte-code, which is u...

Storage

Soroban storage is available to affordably accommodate a wide range of uses.

Guides in this category:

📄 Use instance storage in a contract

Under the hood, instance storage is exactly like persistent storage. The only difference is that anything stored in instance storage has an archival TTL that is tied to the ...

📄 Use persistent storage in a contract

Persistent storage can be very useful for ledger entrys that are not common across every user of the contract instance, but that are not suitable to be temporary (user b...

📄 Use temporary storage in a contract

Temporary storage is useful for a contract to store data that can quickly become irrelevant or out-dated. For example, here's how a contract might be used to store a re...

Use instance storage in a contract

Under the hood, instance storage is exactly like persistent storage. The only difference is that anything stored in instance storage has an archival TTL that is tied to the contract instance itself. So, if a contract is live and available, the instance storage is guaranteed to be so, too.

Instance storage is really useful for global contract data that is shared among all users of the contract (token administrator, for example). From the token example contract, the helper functions to set and retrieve the admininistrator address are basically just wrappers surrounding the one Admin ledger entry.

> **CAUTION**
>
> It should be noted that *every* piece of data stored in `instance()` storage is retrieved from the ledger *every* time the contract is invoked. Even if the invoked function does not interact with any ledger data at all. This can lead to more expensive (computationally and financially) function invocations if the stored data grows over time. Choose judiciously which bits of data actually belong in the instance storage, and which should be kept in persistent storage.

```rust
pub fn has_administrator(e: &Env) -> bool {
    let key = DataKey::Admin;
    e.storage().instance().has(&key)
}

pub fn read_administrator(e: &Env) -> Address {
    let key = DataKey::Admin;
    e.storage().instance().get(&key).unwrap()
}

pub fn write_administrator(e: &Env, id: &Address) {
    let key = DataKey::Admin;
    e.storage().instance().set(&key, id);
}
```

Guides in this category:

Use instance storage in a contract

Under the hood, instance storage is exactly like persistent storage. The only difference is that anything stored in instance storage has an archival TTL that is tied to the ...

Use persistent storage in a contract

Persistent storage can be very useful for ledger entrys that are not common across every user of the contract instance, but that are not suitable to be temporary (user b...

Use temporary storage in a contract

Temporary storage is useful for a contract to store data that can quickly become irrelevant or out-dated. For example, here's how a contract might be used to store a re...

Use persistent storage in a contract

Persistent storage can be very useful for ledger entrys that are not common across every user of the contract instance, but that are not suitable to be temporary (user balances, for example). In this guide, we'll assume we want to store a random number for a user, and store it in the contract's persistent storage as though it were their favorite number.

```rust
#[contracttype]
pub enum DataKey {
    Favorite(Address),
}

#[contract]
pub struct FavoriteContract;

#[contractimpl]
impl FavoriteContract {
    // This function generates, stores, and returns a random number for the user
    pub fn generate_fave(env: Env, user: Address) -> u64 {
        let key = DataKey::Favorite(user);
        let fave: u64 = env.prng().gen();
        env.storage().persistent().set(&key, &fave);
        fave
    }

    // This function retrieves and returns the random number for the user
    pub fn get_fave(env: Env, user: Address) -> u64 {
        let key = DataKey::Favorite(user);
        if let Some(fave) = env.storage().persistent().get(&key) {
            fave
        } else {
            0
        }
    }
}
```

Guides in this category:

Use temporary storage in a contract

Temporary storage is useful for a contract to store data that can quickly become irrelevant or out-dated. For example, here's how a contract might be used to store a recent price of BTC against the US Dollar.

```
// This function updates the BTC price
pub fn update_btc_price(env: Env, price: i128) {
    env.storage().temporary().set(&!symbol_short("BTC"), &price);
}

// This function reads and returns the current BTC price (zero if the storage
// entry is archived)
pub fn get_btc_price(env: Env) -> i128 {
    if let Some(price) = env.storage().temporary().get(&!symbol_short("BTC")) {
        price
    } else {
        0
    }
}
```

Guides in this category:

Use instance storage in a contract

Under the hood, instance storage is exactly like persistent storage. The only difference is that anything stored in instance storage has an archival TTL that is tied to the ...

Use persistent storage in a contract

Persistent storage can be very useful for ledger entrys that are not common across every user of the contract instance, but that are not suitable to be temporary (user b...

Use temporary storage in a contract

Temporary storage is useful for a contract to store data that can quickly become irrelevant or out-dated. For example, here's how a contract might be used to store a re...

Testing

Smart contract testing is vital to ensure safe, resilient, and accurate computation.

Guides in this category:

Implement basic tests for a contract

A contract's test functions can be used as a simple way to ensure a contract's functions behave as expected. The increment example contract has a function that incre...

Test authorized contract invocations

A contract's test functions can be used as a way to ensure the authorization is indeed carried out the way a developer intends. A simple example can be found in the au...

Implement basic tests for a contract

A contract's test functions can be used as a simple way to ensure a contract's functions behave as expected. The increment example contract has a function that increments a counter by one on every invocation. The corresponding test invokes that function several time, ensuring with `assert_eq!(...)` the count increases as expected.

```rust
#![cfg(test)]

use super::{IncrementContract, IncrementContractClient};
use soroban_sdk::{Env};

#[test]
fn test() {
    // Almost every test will begin this same way. A default Soroban environment
    // is created and the contract (along with its client) is registered in it.
    let env = Env::default();
    let contract_id = env.register_contract(None, IncrementContract);
    let client = IncrementContractClient::new(&env, &contract_id);

    assert_eq!(client.increment(), 1);
    assert_eq!(client.increment(), 2);
    assert_eq!(client.increment(), 3);
}
```

Guides in this category:

Test authorized contract invocations

A contract's test functions can be used as a way to ensure the authorization is indeed carried out the way a developer intends. A simple example can be found in the auth example contract. (In the following code block, some code has been omitted for brevity.)

```
#[test]
fn test() {
    let env = Env::default();
    env.mock_all_auths();

    assert_eq!(client.increment(&user_1, &5), 5);
    // Verify that the user indeed had to authorize a call of `increment` with
    // the expected arguments:
    assert_eq!(
        env.auths(),
        std::vec![(
            // Address for which authorization check is performed
            user_1.clone(),
            // Invocation tree that needs to be authorized
            AuthorizedInvocation {
                // Function that is authorized. Can be a contract function or
                // a host function that requires authorization.
                function: AuthorizedFunction::Contract((
                    // Address of the called contract
                    contract_id.clone(),
                    // Name of the called function
                    symbol_short!("increment"),
                    // Arguments used to call `increment` (converted to the
                    // env-managed vector via `into_val`)
                    (user_1.clone(), 5_u32).into_val(&env),
                )),
                // The contract doesn't call any other contracts that require
                // authorization,
                sub_invocations: std::vec![]
            }
        )]
    )
}
```

Guides in this category:

Implement basic tests for a contract

A contract's test functions can be used as a simple way to ensure a contract's functions behave as expected. The increment example contract has a function that incre...

Test authorized contract invocations

A contract's test functions can be used as a way to ensure the authorization is indeed carried out the way a developer intends. A simple example can be found in the au...

Transactions

The entry point for every smart contract interaction is ultimately a transaction on the Stellar network.

Guides in this category:

📄 Invoke a contract function in a Stellar transaction using JavaScript

This is a simple example using the @stellar/stellar-sdk JavaScript library to create, simulate, and then assemble a Stellar transaction which invokes an increment functio...

📄 Submit a transaction to Soroban RPC using the JavaScript SDK

Here is a simple, rudimentary looping mechanism to submit a transaction to Soroban RPC and wait for a result.

Invoke a contract function in a Stellar transaction using JavaScript

This is a simple example using the `@stellar/stellar-sdk` JavaScript library to create, simulate, and then assemble a Stellar transaction which invokes an `increment` function of the auth example contract.

```
(async () => {
  const {
    Keypair,
    Contract,
    SorobanRpc,
    TransactionBuilder,
    Networks,
    BASE_FEE,
  } = require("@stellar/stellar-sdk");

  // The source account will be used to sign and send the transaction.
  // GCWY3M4VRW4NXJRI7IVAU3CC7XOPN6PRBG6I5M7TAOQNKZXLT3KAH362
  const sourceKeypair = Keypair.fromSecret(
    "SCQN3XGRO65BHNSWLSHYIR4B65AHLDUQ7YLHGIWQ4677AZFRS77TCZRB",
  );

  // Configure SorobanClient to use the `soroban-rpc` instance of your
  // choosing.
  const server = new SorobanRpc.Server(
    "https://soroban-testnet.stellar.org:443",
  );

  // Here we will use a deployed instance of the `increment` example contract.
  const contractAddress =
    "CCTAMZGXBVCQJJCX64EVYTM6BKW5BXDI5PRCXTAYT6DVEDXKGS347HWU";
  const contract = new Contract(contractAddress);

  // Transactions require a valid sequence number (which varies from one
  // account to another). We fetch this sequence number from the RPC server.
  const sourceAccount = await server.getAccount(sourceKeypair.publicKey());

  // The transaction begins as pretty standard. The source account, minimum
  // fee, and network passphrase are provided.
  let builtTransaction = new TransactionBuilder(sourceAccount, {
    fee: BASE_FEE,
    networkPassphrase: Networks.FUTURENET,
  })
    // The invocation of the `increment` function of our contract is added
    // to the transaction. Note: `increment` doesn't require any parameters,
    // but many contract functions do. You would need to provide those here.
    .addOperation(contract.call("increment"))
    // This transaction will be valid for the next 30 seconds
    .setTimeout(30)
    .build();

  console.log(`builtTransaction=${builtTransaction.toXDR()}`);

  // We use the RPC server to "prepare" the transaction. This simulating the
  // transaction, discovering the storage footprint, and updating the
  // transaction to include that footprint. If you know the footprint ahead of
  // time, you could manually use `addFootprint` and skip this step.
  let preparedTransaction = await server.prepareTransaction(builtTransaction);

  // Sign the transaction with the source account's keypair.
  preparedTransaction.sign(sourceKeypair);

  // Let's see the base64-encoded XDR of the transaction we just built.
  console.log(
    `Signed prepared transaction XDR: ${preparedTransaction
      .toEnvelope()
      .toXDR("base64")}`,
  );

  // Submit the transaction to the Soroban-RPC server. The RPC server will
  // then submit the transaction into the network for us. Then we will have to
  // wait, polling `getTransaction` until the transaction completes.
  try {
```

Guides in this category:

Invoke a contract function in a Stellar transaction using JavaScript

This is a simple example using the @stellar/stellar-sdk JavaScript library to create, simulate, and then assemble a Stellar transaction which invokes an increment functio...

Submit a transaction to Soroban RPC using the JavaScript SDK

Here is a simple, rudimentary looping mechanism to submit a transaction to Soroban RPC and wait for a result.

Submit a transaction to Soroban RPC using the JavaScript SDK

Here is a simple, rudimentary looping mechanism to submit a transaction to Soroban RPC and wait for a result.

```
import {
  Transaction,
  FeeBumpTransaction,
  SorobanRpc,
} from "@stellar/stellar-sdk";

const RPC_SERVER = "https://soroban-testnet.stellar.org/";
const server = new SorobanRpc.Server(RPC_SERVER);

// Submits a tx and then polls for its status until a timeout is reached.
async function yeetTx(
  tx: Transaction | FeeBumpTransaction,
): Promise<SorobanRpc.Api.GetTransactionResponse> {
  return server.sendTransaction(tx).then(async (reply) => {
    if (reply.status !== "PENDING") {
      throw reply;
    }

    let status;
    let attempts = 0;
    while (attempts++ < 5) {
      const tmpStatus = await server.getTransaction(reply.hash);
      switch (tmpStatus.status) {
        case "FAILED":
          throw tmpStatus;
        case "NOT_FOUND":
          await sleep(500);
          continue;
        case "SUCCESS":
          status = tmpStatus;
          break;
      }
    }

    if (attempts >= 5 || !status) {
      throw new Error(`Failed to find transaction ${reply.hash} in time.`);
    }

    return status;
  });
}

function sleep(ms: number) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}
```

CAUTION

Remember: You should always handle errors gracefully! This is a fail-hard and fail-fast approach that should only be used in these examples.

Guides in this category:

Invoke a contract function in a Stellar transaction using JavaScript

This is a simple example using the @stellar/stellar-sdk JavaScript library to create, simulate, and then assemble a Stellar transaction which invokes an increment functio...

Submit a transaction to Soroban RPC using the JavaScript SDK

Here is a simple, rudimentary looping mechanism to submit a transaction to Soroban RPC and wait for a result.

Tokens

Tokens are directly at the intersection of Stellar and Soroban. Learn more about how to use and interact with them here.

📄 Token Interface

The common interface implemented by tokens that are compatible with Soroban's built-in tokens.

📄 Stellar Asset Contract

Use Stellar assets on Soroban.

Token Interface

Token contracts, including the Stellar Asset Contract and example token implementations expose the following common interface.

Tokens deployed on Soroban can implement any interface they choose, however, they should satisfy the following interface to be interoperable with contracts built to support Soroban's built-in tokens.

Note, that in the specific cases the interface doesn't have to be fully implemented. For example, the custom token may not implement the administrative interface compatible with the Stellar Asset Contract - it won't stop it from being usable in the contracts that only perform the regular user operations (transfers, allowances, balances etc.).

Compatibility Requirements

For any given contract function, there are 3 requirements that should be consistent with the interface described here:

- Function interface (name and arguments) - if not consistent, then the users simply won't be able to use the function at all. This is the hard requirement.
- Authorization - the users have to authorize the token function calls with all the arguments of the invocation (see the interface comments). If this is inconsistent, then the custom token may have issues with getting the correct signatures from the users and may also confuse the wallet software.
- Events - the token has to emit the events in the specified format. If inconsistent, then the token may not be handled correctly by the downstream systems such as block explorers.

Code

The interface below uses the Rust soroban-sdk to declare a trait that complies with the SEP-41 token interface.

```rust
pub trait TokenInterface {
    /// Returns the allowance for `spender` to transfer from `from`.
    ///
    /// # Arguments
    ///
    /// - `from` - The address holding the balance of tokens to be drawn
from.
    /// - `spender` - The address spending the tokens held by `from`.
    fn allowance(env: Env, from: Address, spender: Address) -> i128;

    /// Set the allowance by `amount` for `spender` to transfer/burn from
    /// `from`.
    ///
    /// # Arguments
    ///
    /// - `from` - The address holding the balance of tokens to be drawn
from.
    /// - `spender` - The address being authorized to spend the tokens held
by
    /// `from`.
    /// - `amount` - The tokens to be made available to `spender`.
    /// - `live_until_ledger` - The minimum ledger number that this allowance
will live until.
    /// Cannot be less than the current ledger number unless the amount is
being
    /// set to 0.  An non-live entry (where live_until_ledger < the current
    /// ledger number) should be treated as a 0 amount allowance.
    ///
    /// # Events
    ///
    /// Emits an event with topics `["approve", from: Address,
    /// spender: Address], data = [amount: i128, live_until_ledger: u32]`
    ///
    /// Emits an event with:
    /// - topics - `["approve", from: Address, spender: Address]`
```

> CAUTION WHEN MODIFYING ALLOWANCES
>
> The `approve` function overwrites the previous value with `amount`, so it is possible for the previous allowance to be spent in an earlier transaction before `amount` is written in a later transaction. The result of this is that `spender` can spend more than intended. This issue can be avoided by first setting the allowance to 0, verifying that the spender didn't spend any portion of the previous allowance, and then setting the allowance to the new desired amount. You can read more about this issue here - https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729.

Metadata

Another requirement for complying with the token interface is to write the standard metadata (`decimal`, `name`, and `symbol`) for the token in a specific format. This format allows users to directly read constant data from the ledger instead of invoking a Wasm function. The token example demonstrates how to use the Rust soroban-token-sdk to write the metadata, and we strongly encourage token implementations to follow this approach.

Handling Failure Conditions

In the token interface, there are several instances where function calls can fail due to various reasons such as lack of proper authorization, insufficient allowance or balance, etc. To handle these failure conditions, it is important to specify the expected behavior when such situations arise.

Its important to note the that the token interface not only incorporates the authorization concept for matching asset authorization in Stellar Classic, but it also utilizes the Soroban authorization mechanism. So, if you try to make a token call and it fails, it could be because of either token authorization processes.

To provide more context, when you use the token interface, there is a function called `authorized` that returns "true" if an address has token authorization.

More details on Authorization can be found here.

For the functions in the token interface, trapping should be used as the standard way to handle failure conditions since the interface is not designed to return error codes. This means that when a function encounters an error, it will halt execution and revert any state changes that occurred during the function call.

Failure Conditions

Here is a list of basic failure conditions and their expected behavior for functions in the token interface:

Admin functions:

- If the admin did not authorize the call, the function should trap.
- If the admin attempts to perform an invalid action (e.g., minting a negative amount), the function should trap.

Token functions:

- If the caller is not authorized to perform the action (e.g., transferring tokens without proper authorization), the function should trap.
- If the action would result in an invalid state (e.g., transferring more tokens than available in the balance or allowance), the function should trap.

Example: Handling Insufficient Allowance in `burn_from` function

In the `burn_from` function, the token contract should check whether the spender has enough allowance to burn the specified amount of tokens from the `from` address. If the allowance is insufficient, the function should trap, halting execution and reverting any state changes.

Here's an example of how the `burn_from` function can be modified to handle this failure condition:

```
fn burn_from(
    env: soroban_sdk::Env,
    spender: Address,
    from: Address,
    amount: i128,
) {
    // Check if the spender has enough allowance
    let current_allowance = allowance(env, from, spender);
    if current_allowance < amount {
        // Trap if the allowance is insufficient
        panic!("Insufficient allowance");
    }

    // Proceed with burning tokens
    // ...
}
```

By clearly outlining how to handle failures and incorporating the right error management techniques in the token interface, we can make token contracts stronger and safer.

Stellar Asset Contract

The Stellar Asset Contract (SAC) is an implementation of CAP-46-6 Smart Contract Standardized Asset and SEP-41 Token Interface for Stellar assets.

Overview

> NOTE
>
> Stellar assets are issued by Stellar accounts. Issue an asset on Stellar by following the Issue an Asset Tutorial.

The Stellar Asset Contract allows users and contracts to make payments with, and interact with, assets. The SAC can interact with assets held by Stellar accounts or contracts.

The SAC is a special built-in contract that has access to functionality of the Stellar network that allows it to use Stellar assets directly.

Each Stellar asset has an instance of the SAC reserved on the network. To use the SAC reserved for an asset, the instance just needs to be deployed.

When the SAC transfers assets between accounts, the same debit and credits occur as they do when a Stellar payment operation is used, because the SAC interacts directly with Stellar account trust lines. When the SAC transfers assets between contracts, it uses Contract Data ledger entries to store the balances for contracts.

Stellar account balances for the native asset are always stored on the account, and Stellar contract balances for the native asset are always stored in a contract data entry.

Stellar account balances for issued assets are always stored in trust lines, and Stellar contract balances for issued assets are always stored in a contract data entry.

For example, when transferring from a Stellar account to a Stellar contract, the Stellar account's trust line entry is debited, and a contract data entry is credited.

And for example, when transferring from a Stellar contract to a Stellar account, a contract data entry is debited, and the account's trust line entry is credited.

In both those examples it is a single asset that is transferring from the account to the contract and back again. No bridging is required and no intermediary tokens are needed. An asset on Stellar and it's Stellar Asset Contract represent the same asset. The SAC for an asset is simply an API for interacting with the asset.

The SAC implements the SEP-41 Token Interface, which is similar to the widely used ERC-20 token standard. Contracts that depend on only the SEP-41 portion of the SAC's interface, are also compatible with any custom token that implements SEP-41.

Some functionality available on the Stellar network in transaction operations, such as the order book, do not have any functions exposed on the Stellar Asset Contract in the current protocol.

Deployment

Every Stellar asset on Stellar has reserved a contract address that the Stellar Asset Contract can be deployed to. Anyone can initiate the deploy and the Stellar asset issuer does not need to be involved.

It can be deployed using the Soroban-CLI as shown here.

Or the [Stellar SDK] can be used as shown here by calling `InvokeHostFunctionOp` with `HOST_FUNCTION_TYPE_CREATE_CONTRACT` and `CONTRACT_ID_FROM_ASSET`. The resulting token will have a deterministic identifier, which will be the sha256 hash of `HashIDPreimage::ENVELOPE_TYPE_CONTRACT_ID_FROM_ASSET` xdr specified here.

Anyone can deploy the instances of Stellar Asset Contract. Note, that the initialization of the Stellar Asset Contracts happens automatically during the deployment. Asset Issuer will have the administrative permissions after the contract has been deployed.

Interacting with classic Stellar assets

The Stellar Asset Contract is the only way for contracts to interact with Stellar assets, either the native XLM asset, or those issued by Stellar accounts.

The issuer of the asset will be the administrator of the deployed contract. Because the Native Stellar token doesn't have an issuer, it will not have an administrator either. It also cannot be burned.

After the contract has been deployed, users can use their classic account (for lumens) or trustline (for other assets) balance. There are some differences depending on if you are using a classic account `Address` vs a contract `Address` (corresponding either to a regular contract or to a custom account contract). The following section references some issuer and trustline flags from Stellar classic, which you can learn more about here.

- Using `Address::Account`
    - The balance must exist in a trustline (or an account for the native balance). This means the contract will not store the balance in ContractData. If the trustline or account is missing, any function that tries

to interact with that balance will fail.

- Classic trustline semantics will be followed.
    - Transfers will only succeed if the corresponding trustline(s) have the `AUTHORIZED_FLAG` set.
    - A trustline balance can only be clawed back using the `clawback` contract function if the trustline has `TRUSTLINE_CLAWBACK_ENABLED_FLAG` set.
    - Transfers to the issuer account will burn the token, while transfers from the issuer account will mint.
    - Trustline balances are stored in a 64-bit signed integer even though the interface accepts 128-bit signed integers. Any operation that attempts to send or receive an amount more than the maximum amount that can be represented by a 64-bit signed integer will fail.

- Using `Address::Contract`
    - The balance and authorization state will be stored in contract storage, as opposed to a trustline.
    - Balances are stored in a 128-bit signed integer.
    - A balance can only be clawed back if the issuer account had the `AUTH_CLAWBACK_ENABLED_FLAG` set when the balance was created. A balance is created when either an `Address::Contract` is on the receiving end of a successful transfer, or if the admin sets the authorization state. Read more about `AUTH_CLAWBACK_ENABLED_FLAG` here.

Balance Authorization Required

In the `Address::Contract` case, if the issuer has `AUTH_REQUIRED_FLAG` set, then the specified `Address::Contract` will need to be explicitly authorized with `set_auth` before it can receive a balance. This logic lines up with how trustlines interact with the `AUTH_REQUIRED_FLAG` issuer flag, allowing asset issuers to have the same control in Soroban as they do in Stellar classic. Read more about `AUTH_REQUIRED_FLAG` here.

Revoking Authorization

The admin can only revoke authorization from an `Address`, if the issuer of the asset has `AUTH_REVOCABLE_FLAG` set. The deauthorization will fail if the issuer is missing. This requirement is true for both the trustline balances of `Address::Account` and contract balances of `Address:Contract`. Note that when a trustline is deauthorized from Soroban, `AUTHORIZED_FLAG` is cleared and `AUTHORIZED_TO_MAINTAIN_LIABILITIES_FLAG` is set to avoid having to pull offers and redeeming pool shares.

Authorization semantics

See the authorization overview and auth example for general information about authorization in Soroban.

The token contract contains three kinds of operations that follow the token interface:

- getters, such as `balance`, which do not change the state of the contract
- unprivileged mutators, such as `incr_allow` and `xfer`, which change the state of the contract but do not require special privileges
- privileged mutators, such as `clawback` and `set_admin`, which change the state of the contract but require special privileges

Getters require no authorization because they do not change the state of the contract and all contract data is public. For example, `balance` simply returns the balance of the specified `Address` without changing it.

Unprivileged mutators require authorization from the `Address` that spends or allows spending their balance. The exceptions are `xfer_from` and `burn_from` operations where the `Address` that require authorization from the 'spender' entity that has got an allowance from another `Address` beforehand.

Priviliged mutators require authorization from a specific privileged identity, known as the "administrator". For example, only the administrator can `mint` more of the token. Similarly, only the administrator can appoint a new administrator.

Using Stellar Asset Contract with other contracts

From the contract perspective Stellar Asset Contract is not different from any other token that implements the Soroban token interface. The Rust SDK contains a pregenerated client for any contract that implements the token interface:

```
use soroban_sdk::token;

struct MyContract;

#[contractimpl]
impl MyContract {
  fn token_fn(e: Env, id: Address) {
    // Create a client instance for the provided token identifier. If the id
    // value corresponds to an SAC contract, then SAC implementation is used.
    let client = token::Client::new(&env, &id);
    // Call token operations part of the SEP-41 token interface
    client.transfer(...);
  }
}
```

CLIENTS

A client created by [`token::Client`] implements the functions defined by any contract that implements the SEP-41 Token Interface. But the Stellar

Asset Contract exposes additional functions such as `mint`. To access the additional functions, another client needs to be used: `token::StellarAssetClient`. This client only implements the functions which are not part of the SEP-41.

```
let client = token::StellarAssetClient::new(&env, &id);
// Call token operations which are not part of the SEP-41 token
interface
// but part of the CAP-46-6 Smart Contract Standardized Asset
client.mint(...);
```

Examples

See the full examples that utilize the token contract in various ways for more details:

- Timelock and single offer move token via `xfer` to and from the contract
- Atomic swap uses `incr_allow` to transfer token on behalf of the user

Notice, that these examples don't do anything to support SAC specifically.

Testing

Soroban Rust SDK provides an easy way to instantiate a Stellar Asset Contract tokens using `register_stellar_asset_contract`. For example:

```
let admin = Address::random();
let user = Address::random();
let token = StellarAssetClient::new(e,
&e.register_stellar_asset_contract(admin.clone()));
token.mint(&admin, &user, &1000);
```

See the tests in the examples above for the full test implementation.

Contract Interface

This interface can be found in the SDK. It extends the common SEP-41 Token Interface.