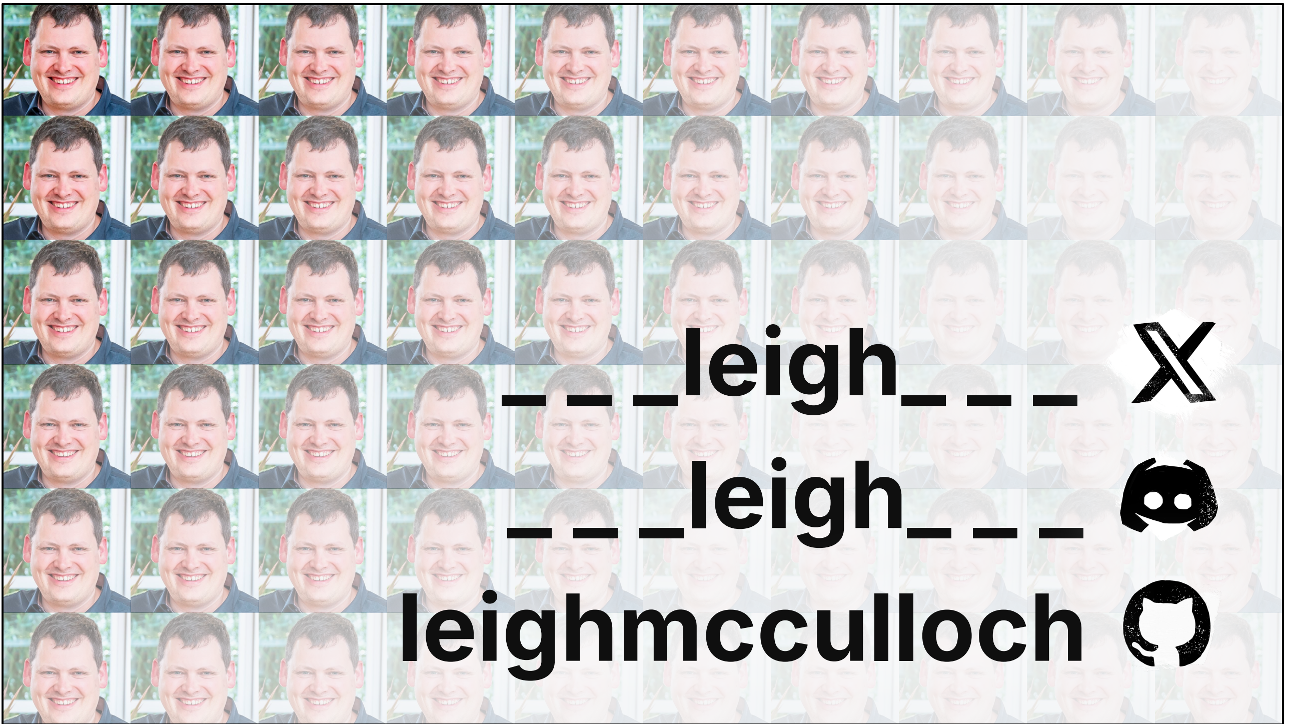
LEIGH MCCULLOCH

Principal Software Engineer,
Stellar Development Foundation



• Hi everyone, I'm Leigh. That's of the ways you can interact with me.
• I'm a software engineer at the Stellar Development Foundation. In the Stellar ecosystem you'll see me contributing to the Stellar CLI, and the Rust Soroban SDK, among other things.

DOES IT WORK?

- How do we know that something works?
- Scientists have for centuries used the “scientific method”.
- Wikipedia describes the scientific method as…

A METHOD FOR ACQUIRING KNOWLEDGE THAT…

—Wikipedia “scientific method” (CC BY-SA 4.0)

- “A method for acquiring knowledge that…

“

...INVOLVES CAREFUL OBSERVATION COUPLED WITH RIGOROUS SCEPTICISM

—Wikipedia “scientific method” (CC BY-SA 4.0)

- “... involves careful **observation** coupled with rigorous **scepticism**”
- Observation is maybe obvious, but
- rigorous scepticism, what’s that about.

“

…COGNITIVE ASSUMPTIONS CAN DISTORT THE INTERPRETATION OF OBSERVATION

—Wikipedia “scientific method” (CC BY-SA 4.0)

- Wikipedia goes on to say, “because cognitive assumptions can distort the interpretation of observation.”
- If we’re not being rigorously skeptical that something works, we’re making…assumptions.

ASSUMPTIONS

- What do we say about assumptions?
- Assumptions make a donkey out of u and me.
- Assumptions are the mother of all mistakes.
- We’ve all been there. Untested assumptions become bugs.
- Dijkstra said about bugs…

IF DEBUGGING IS THE PROCESS OF REMOVING SOFTWARE BUGS...

—Edsger W. Dijkstra

- Dijkstra said about bugs:
- “If debugging is the process of removing software bugs,

THEN PROGRAMMING MUST BE THE PROCESS OF PUTTING THEM IN.

—Edsger W. Dijkstra

- then programming must be the process of putting them in."
- Dijkstra highlights that bug creation is a natural part of development.
- Therefore for testing to be effective at identifying bugs early, it must be holistically integrated with development, just like bug creation.
- Not an afterthought. Not only done at the end.

THEN PROGRAMMING MUST BE THE PROCESS OF PUTTING THEM IN.

—Edsger W. Dijkstra

- We write the code, we feel confident that it'll do what we want, but until we run the code, until we take the position that it doesn't work and prove otherwise, we don't truly know, do we?
- And yes, the longer we do what we do, we get better at writing code that works. No errors, no warnings, but we're fallible, errors creep in.
- There's a limit to us humans and our ability to make things that work. That's why so much of life, the buildings we live in, the cars we drive, is observed, and tested.

THEN PROGRAMMING MUST BE THE PROCESS OF PUTTING THEM IN.

—Edsger W. Dijkstra

- So what do we do as software engineers, blockchain builders, contract developers…
- We use the scientific method, careful observation, rigorous scepticism.
- In other words, we test.

TEST

• We test.
• Automated testing is one of the most powerful tools we have for creating reliable, safe software.
• Good tests…
- give us confidence that the software does what we intend the first time we build it.
- give us confidence to refactor code.
- help us understand and prove the scope of a change to existing software.
- are a multiplier on our teams because they give others confidence to refactor and change code that they have no prior experience with.
• And good testing tools and strategies make our skepticism much more rigorous than we alone are capable.

TESTING STELLAR CONTRACTS

• Is testing different on Stellar vs other blockchains.
• So Stellar has been around for a while and is one of the OG blockchains.
• Before contracts, Stellar already had a great testing story: it has a very stable and accessible test network, super simple UIs for getting access to test lumens, and tools like the laboratory and the quickstart docker image enabled developers to explore, learn, and test.
• A consistent theme with adding contracts to Stellar was that the platform would bring along it's existing strengths, and learn from the successes and exploits of other blockchains that have gone before in the contract space:
• Surprise, a huge learning was that testing was critical to building safe contracts.
• So we set out to make testing contracts a first-class concern of the platform, and a great experience from day one.
• As apart of that effort we..

ONE LANGAGE

{RUST}

TESTING STELLAR CONTRACTS

- One language to build and test.
- A large reason for that was to…

ONE LANGAGE

{RUST}

REDUCE CONTEXT SWITCHING

TESTING STELLAR CONTRACTS

- Reducing context switching as much as possible
- What this really means is…
- Zero barriers to test. You learn how to write a contract, and then you already have the knowledge to test.

ONE LANGAGE

{RUST}

TESTING STELLAR CONTRACTS

REDUCE CONTEXT SWITCHING

CONSISTENT WITH MAINNET

- Making testing as close as possible to mainnet

TESTING STELLAR CONTRACTS

ONE LANGAGE

{RUST}

REDUCE CONTEXT SWITCHING

CONSISTENT WITH MAINNET

ADVANCED TESTING

- Support for advanced testing approaches like…

ONE LANGAGE

{RUST}

TESTING STELLAR CONTRACTS

REDUCE CONTEXT SWITCHING

PROPERTY TESTING

FUZZING

CONSISTENT WITH MAINNET

ADVANCED TESTING

- fuzzing and property testing

ONE LANGAGE

{RUST}

TESTING
STELLAR
CONTRACTS

BATTERIES INCLUDED

PROPERTY TESTING

FUZZING

REDUCE CONTEXT SWITCHING

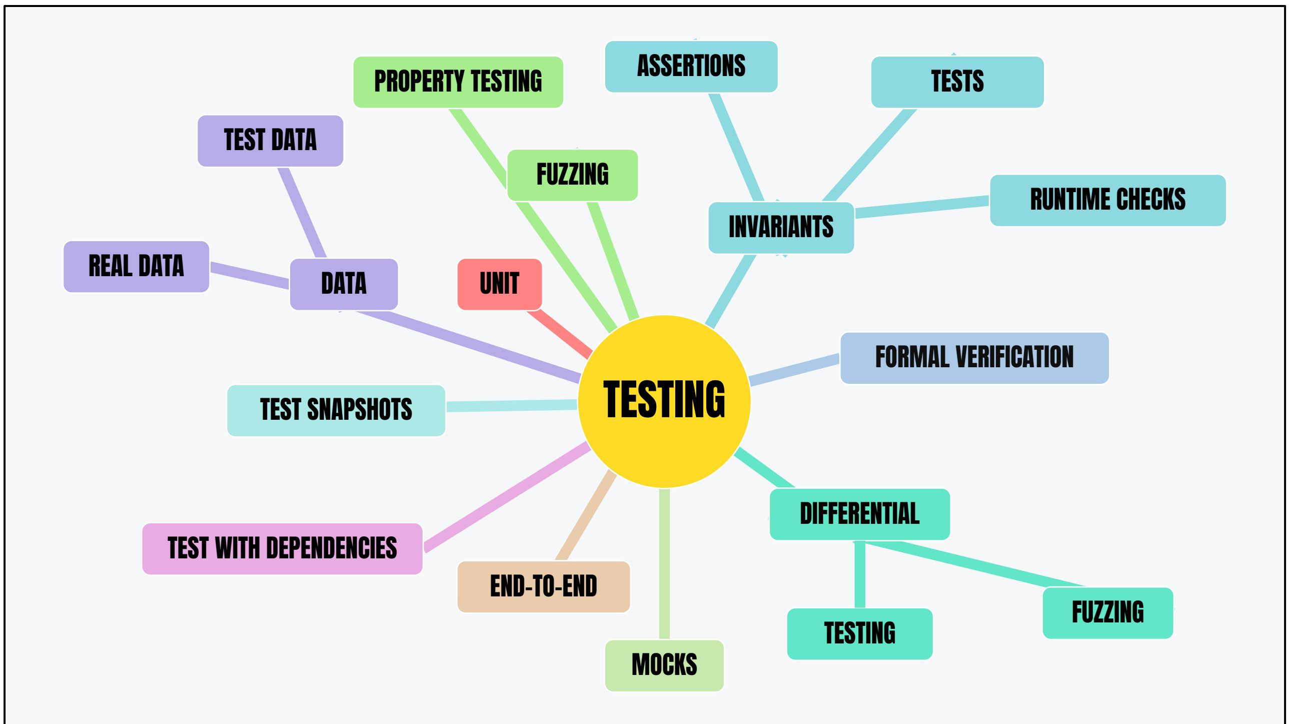
CONSISTENT WITH MAINNET

ADVANCED TESTING

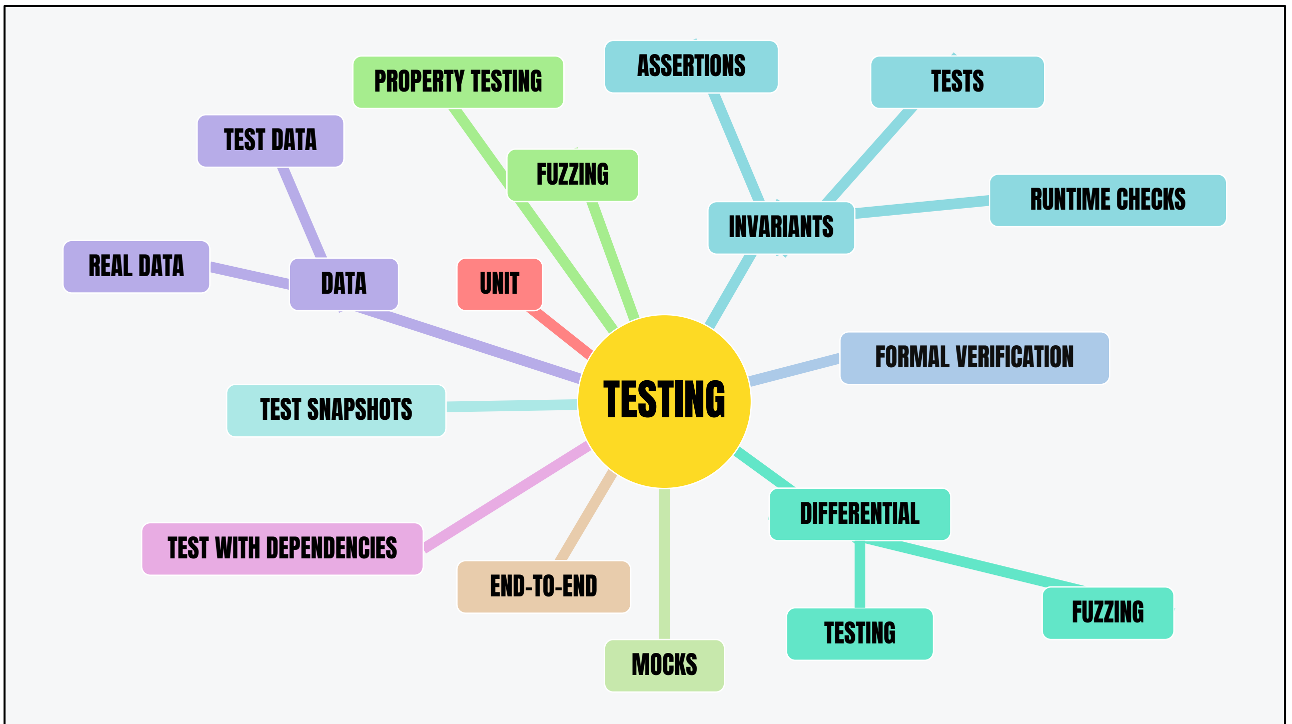
- The ability to leverage an existing test ecosystem
- A test and tooling ecosystem that Stellar can exist within

—

- What’s the scope of automated testing?



- When we say testing, we're talking about all the forms of automated, or "computer powered" testing.
- Unit testing.
- Mocks.
- Test snapshots, and be able to detect changes in it over time.
- Integration testing where we test against the third party contracts we depend.
- End-to-end tests.
- Fuzzing and property testing.
- Invariant testing and fuzzing.
- Differential testing and fuzzing.
- Formal verification.



- We’re going to talk about a lot of these strategies.
- To be as practical as possible, we’re going to look at code, and spend the rest of our time immersed in that code.
- My hope is that through looking at examples of real functioning tests, that we would see how easy it is to write tests that go further than a unit test, and that we have no reason not to adopt some or all of these testing strategies.

developers.stellar.org

- If you haven’t written a Stellar contract yet, the place to learn is developers.stellar.org.
- Give yourself the gift, spend 15 mins to get setup, and deploy a contract.

```rust
#[contractimpl]
impl Token {
    pub fn __constructor(admin: Address)

    pub fn mint(to: Address, amount: i128) -> Result<(), Error>

    pub fn balance(addr: Address) -> i128

    pub fn transfer(from: Address, to: Address, amount: i128)
        -> Result<(), Error>
}

#[contracterror]
pub enum Error {
    Overflow = 1,
    InsufficientBalance = 2,
    NegativeAmount = 3,
}
```

CONTRACT

• To be able show examples of tests, we need something to test.
• So the tests I'm going to show are testing a Token contract that has this small interface.
• It has:
- A constructor. Constructors are coming in protocol 22. They get called once on the creation of the contract instance, when it is deployed.
- A mint function that only the admin can authorize to create the token amount and give it to the address.
- A balance function for retrieving how much of a token an address holds.
- A transfer function that moves an amount from one address to another.
- An error defined so if the contract fails to execute we'll know why and what went wrong.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- This is a test. It's a very simple one.
- It expects that an address that doesn't hold the token will have a balance of zero.
- This is what most people would describe as a unit test. There's no dependencies being tested, a single function is under test.
- Stepping through how this test works…

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- It sets up the Soroban Environment, the Env value.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- It registers the Token contract with the environment, the admin is passed to the contract's constructor.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- The TokenClient is a generated type.
- Every contract has a client generated that you can use to call that contract's functions.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- The Address generate function generates a new unique address that the test will use for checking the balance.
- The token doesn't know about this address until the balance function is called, and so the address doesn't hold a balance of the token.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- Coming to the last line…
- We check that the balance function returns zero when given an address that doesn't hold the token.
- The assert equal function checks the two values are equal and outputs a meaningful message if they aren't.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

• Some things I want to call out about this test:
- It’s a simple test, but it is a complete test. There’s a full Soroban Environment setup, used, and torn down in this test, and that happens fast. If you write hundreds or thousands of tests, the Rust test harness will run them in parallel and each will have its own isolated Soroban Environment.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- The Env created at the beginning of the test is not a simulation of the Soroban Environment, it's the same Soroban Environment that mainnet uses to run actual contracts. There are some minor differences, test utilities are enabled, the storage backend is different, auth can be disabled, but you aren't running tests against a second implementation of the Environment. As a developer this gives me a lot of confidence about the tests I write, and the expectation that they behave like mainnet.

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- You will have noticed, contract tests are written in Rust, the same language the contract is written in.
- So that means no context switching from one toolchain to another to write your tests. The same tools, functions, APIs are available.
- Because it's written in Rust, you can use test utilities that exist in the Rust ecosystem, like mockall, or the predicates test library.
- Because it's written in Rust, you can use a step-through-debugger when running the test, and debug code in your IDE.
- And when I say your IDE, I'm not talking about a Stellar specific IDE, or a new IDE you need to learn, I truly mean your IDE…

YOU + YOUR IDE = GOOD TIMES

- …your IDE, any IDE that supports Rust, whether that be VSCode, IntelliJ's RustRover, Zed, Sublime, Vim. Anything that supports rust language server is going to help you write tests, and anything that supports lldb will help you step through debug your code.
- Getting back to the test…

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

    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    assert_eq!(token.balance(&a), 0);
}
```

TEST

- Getting back to the test, it doesn’t do very much, let’s look at another…

```rust
#[test]
fn test() {
    let env = Env::default();
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    let b = Address::generate(&env);
    token.mock_all_auths().mint(&a, &10);
    assert_eq!(token.balance(&a), 10);
    assert_eq!(token.balance(&b), 0);

    token.mock_all_auths().transfer(&a, &b, &2);
    // ... assert on auths
    assert_eq!(token.balance(&a), 8);
    assert_eq!(token.balance(&b), 2);
    // ... assert on events
```

TEST

- This test asserts that a transfer moves an amount from one address to another, from…

```rust
#[test]
fn test() {
    let env = Env::default();
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    let b = Address::generate(&env);
    token.mock_all_auths().mint(&a, &10);
    assert_eq!(token.balance(&a), 10);
    assert_eq!(token.balance(&b), 0);

    token.mock_all_auths().transfer(&a, &b, &2);
    // ... assert on auths
    assert_eq!(token.balance(&a), 8);
    assert_eq!(token.balance(&b), 2);
    // ... assert on events
```

TEST

- …a to b.
- The test sets up a and b with starting balances of 10 and 0.
- After the transfer occurs we assert on the changes in the balances.
- There's only so much code I can fit on a slide at once, so not pictured are:
- assertions on auths
- events
- but I want to show you them because it's really important to assert on these things.

```
token.mock_all_auths().transfer(&a, &b, &2);

assert_eq!(env.auths(), [
    (
        a.clone(),
        AuthorizedInvocation {
            function: AuthorizedFunction::Contract((
                token.address.clone(),
                symbol_short!("transfer"),
                (&a, &b, 2i128).into_val(&env),
            )),
            sub_invocations: [].into(),
        }
    ),
]);

assert_eq!(token.balance(&a), 8);
assert_eq!(token.balance(&b), 2);
```

TEST

- This is what asserting on the auths looks like.
- We get the set of auths from the environment of the last invocation.
- Then assert that they match the authorizations we expected to be required.
- In this case expecting that the a address was required to authorize the transfer call with the from, to, and amount parameters.

```
assert_eq!(token.balance(&a), 8);
assert_eq!(token.balance(&b), 2);

assert_eq!(
    env.events().all(),
    vec![
        &env,
        (
            token.address.clone(),
            (symbol_short!("transfer"), &a, &b).into_val(&env),
            2i128.into_val(&env),
        )
    ],
);
```

TEST

• And this is what asserting on the event looks like.

> Now none of the tests we've written have a need to mock other contracts, or setup other contracts, because it really is a very simple contract. But often there is some interaction or composability with other contracts. So let's make a change to the contract we're testing…

```rust
#[contractimpl]
impl Token {
    pub fn __constructor(admin: Address, pause: Address)

    pub fn mint(to: Address, amount: i128) -> Result<(), Error>

    pub fn balance(addr: Address) -> i128

    pub fn transfer(from: Address, to: Address, amount: i128)
        -> Result<(), Error>
}

#[contracterror]
pub enum Error {
    Overflow = 1,
    InsufficientBalance = 2,
    NegativeAmount = 3,
    Paused = 4,
}
```

CONTRACT

> …to introduce a dependency.
• We'll add a pause contract, and say that the token can only be transferred when the pause contract isn't paused.
• When the token is created it'll be given the address of the pause contract, and when transfer is called if the contract is paused the transfer will fail with the Paused error.
• When we go to write a test for this contract, we now have this dependency on another contract to figure out.
• If we're writing unit tests, it's pretty natural to reach for a mock, so let's mock the contract.
> This is what a test that mocks the pause contract looks like…

TEST WITH MOCK

```rust
#[contract]
struct Pause;
#[contractimpl]
impl Pause {
    pub fn paused() -> bool { false }
}

#[test]
fn test_not_paused() {
    let env = Env::default();
    let admin = Address::generate(&env);
    let pause = env.register(Pause, ());
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

- In this test we're running the scenario where the pause contract is not paused.
- Outside of the test setup a Pause contract is defined, it's a mock of the real thing, and will return false.
- In the test setup, that Pause contract gets registered just like the token contract does.
- And we pass the address of the Pause contract to the constructor of the Token contract.
- Our test is setup now and we can write our assertions.

TEST WITH MOCK

```rust
#[contract]
struct Pause;
#[contractimpl]
impl Pause {
    pub fn paused() -> bool { false }
}

#[test]
fn test_not_paused() {
    // test setup
    token.mock_all_auths().mint(&a, &10);
    token.mock_all_auths().transfer(&a, &b, &2);
    // assert auths
    assert_eq!(token.balance(&a), 8);
    assert_eq!(token.balance(&b), 2);
    // assert events
}
```

- And our assertion will check that when we call transfer function, it succeeds normally, auths are required, balances are updated, and events occur.

TEST WITH MOCK

```rust
#[contract]
struct Pause;
#[contractimpl]
impl Pause {
    pub fn paused() -> bool { true }
}

#[test]
fn test_paused() {
    // test setup
    token.mock_all_auths().mint(&a, &10);
    assert_eq!(
        token.mock_all_auths().try_transfer(&a, &b, &2),
        Err(Ok(Error::Paused))
    );
}
```

- In the second test we're running the scenario where the pause contract is paused.
- Our mock for this test returns true instead of false.
- And our assertion will check that when we call transfer, an error is returned and that error is the Paused error.

TEST WITH MOCK

```rust
#[contract]
struct Pause;
#[contractimpl]
impl Pause {
    pub fn paused() -> bool { true }
}

#[test]
fn test_paused() {
    // test setup
    token.mock_all_auths().mint(&a, &10);
    assert_eq!(
        token.mock_all_auths().try_transfer(&a, &b, &2),
        Err(Ok(Error::Paused))
    );
}
```

- So we've tested the obvious cases, our mock returns a bool, which can only be a true or false. So we've tested all the cases that our mock says can exist.
- What happens if the pause contract returns values that aren't…

TEST WITH MOCK

```
#[contract]
struct Pause;
#[contractimpl]
impl Pause {
    pub fn paused() → ? { ? }
}

#[test]
fn test_?() {
    // test setup
    token.mock_all_auths().mint(&a, &10);
    assert_eq!(
        token.mock_all_auths().try_transfer(&a, &b, &2),
        ?
    );
}
```

• true or false?
• How does our contract behave then, does it handle that situation in a way that's safe?
• What happens if we deploy this contract and one day find it's gone up in flames because a dependency returns values that we don't expect?

TEST WITH MOCK

```rust
#[contract]
struct Pause;
#[contractimpl]
impl Pause {
    pub fn paused() → ? { ? }
}

#[test]
fn test_?() {
    // test setup
    token.mock_all_auths().mint(&a, &10);
    assert_eq!(
        token.mock_all_auths().try_transfer(&a, &b, &2),
        ?
    );
}
```

- Mocks have their limitations. They make assumptions.
- These assumptions may come back to bite us if they don't align with reality.
- Even when those assumptions are reasonable, or even defendable.

> You may have heard of this story…

A SOFTWARE TESTER WALKS INTO A BAR

- A software tester walks into a bar

A SOFTWARE TESTER WALKS INTO A BAR

AND ORDERS

- And orders

A SOFTWARE TESTER WALKS INTO A BAR
AND ORDERS
1 DRINK

- 1 drink

A SOFTWARE TESTER WALKS INTO A BAR
AND ORDERS
1 DRINK
2 DRINKS

- 2 drinks

A SOFTWARE TESTER WALKS INTO A BAR
AND ORDERS
1 DRINK
2 DRINKS
0 DRINKS

- 0 drinks

A SOFTWARE TESTER WALKS INTO A BAR
AND ORDERS
1 BEER
2 BEERS
0 BEERS
99999999 DRINKS

- 99 million drinks

AND ORDERS
1 DRINK
2 DRINKS
0 DRINKS
99999999 DRINKS
A LIZARD IN A GLASS

- A lizard in a glass

1 DRINK
2 DRINKS
0 DRINKS
99999999 DRINKS
A LIZARD IN A GLASS
-1 DRINK

- -1 drink

2 DRINKS
0 DRINKS
99999999 DRINKS
A LIZARD IN A GLASS
-1 DRINK
“QWERTYUIOP” DRINKS

- Qwertyyuiop drinks

0 DRINKS
99999999 DRINKS
A LIZARD IN A GLASS
-1 DRINK
“QWERTYUIOP” DRINKS
TESTING COMPLETE

- The tester says that testing is complete

99999999 DRINKS
A LIZARD IN A GLASS
-1 DRINK
“QWERTYUIOP” DRINKS
TESTING COMPLETE
A REAL CUSTOMER WALKS INTO THE BAR AND ASKS

- A real customer walks into the bar and asks

A LIZARD IN A GLASS
-1 DRINK
"QWERTYUIOP" DRINKS
TESTING COMPLETE
A REAL CUSTOMER WALKS INTO THE BAR AND ASKS
WHERE IS THE BATHROOM?

- Where is the bathroom?

-1 DRINK
“QWERTYUIOP” DRINKS
TESTING COMPLETE
A REAL CUSTOMER WALKS INTO THE BAR AND ASKS
WHERE IS THE BATHROOM?
THE BAR GOES UP IN FLAMES

- The bar goes up in flames
- We can do a pretty good job of imagining all the variety of inputs our program might fail on.
- -1 Drink for example.
- This is actually a decent test set.
- But we need strategies for testing that are bigger than our imagination.

UNIT TESTING IS NOT ENOUGH

• Unit testing is not enough.
• Unit tests have limitations.
• Unit tests are limited to testing:
- the inputs that we can imagine
- the dependency behavior that we can imagine
- and the state that we can imagine
> To close the gap on the limitations of unit testing, …

1. **TESTING WITH REAL DEPENDENCIES**
2. **TESTING WITH REAL DATA**
3. **FUZZING AND PROPERTY TESTING**
4. **INVARIANTS**
5. **DIFFERENTIAL TESTING**

• we’re going to look at five testing strategies we can apply.
- Testing with mainnet dependencies.
- Testing with mainnet data.
- Fuzzing and property testing.
- Invariants.
- Differential Testing.

> The first strategy we’ll look at is…

TESTING WITH REAL DEPENDENCIES

TESTING WITH REAL DEPENDENCIES

- Testing with real dependencies.
- When we say real, we mean, the actual contracts that are deployed to mainnet.
- Instead of mocking like we did in the last test, we can…

TESTING WITH REAL DEPENDENCIES

```
$ stellar contract fetch --id C...  --out-file pause.wasm
```

- Instead of mocking like we did, we can…
- use the fetch command of the stellar-cli to download the contract from the network.
- The C… is the contract address of the pause contract.
- The contract will be written to the file, pause.wasm.

> Then the next thing we do is…import it into the test…

TESTING WITH REAL DEPENDENCIES

```
$ stellar contract fetch --id C...  --out-file pause.wasm
```

```rust
mod pause {
    soroban_sdk::contractimport!(file = "pause.wasm");
}

#[test]
fn test() {
    let env = Env::default();
    let pause = env.register(pause::WASM, ());

    // ...
}
```

- …import it into the test with the contract import call, giving it the name of the file.
- We register the contract as part of the test, where it says register pause wasm.
- This test will now run with the real dependency.

TESTING WITH REAL DEPENDENCIES

```
$ stellar contract fetch --id C... --out-file pause.wasm
```

```rust
mod pause {
    soroban_sdk::contractimport!(file = "pause.wasm");
}

#[test]
fn test() {
    let env = Env::default();
    let pause = env.register(pause::WASM, ());
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

- As we bring in the rest of the test setup.
- You'll notice that not much really changes compared to when we were mocking.
- Unless you need to do some additional test setup with the imported contract most of the test should be identical, the rest of the test setup is the same, the test assertions would be the same.
- The tests don't see or experience native or wasm contracts any differently.
- This means for many tests you can take a test using a mock and rerun the same test on a wasm contract,
- just by doing those three steps of:

```
$ stellar contract fetch --id C... --out-file pause.wasm
```

1

TESTING WITH REAL DEPENDENCIES

```rust
mod pause {
    soroban_sdk::contractimport!(file = "pause.wasm");
}

#[test]
fn test() {
    let env = Env::default();
    let pause = env.register(pause::WASM, ());
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

- Fetch the contract

```
$ stellar contract fetch --id C...  --out-file pause.wasm
```

1

TESTING WITH REAL DEPENDENCIES

```rust
mod pause {
    soroban_sdk::contractimport!(file = "pause.wasm");
}

#[test]
fn test() {
    let env = Env::default();
    let pause = env.register(pause::WASM, ());
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

2

- Import it

TESTING WITH REAL DEPENDENCIES

```
$ stellar contract fetch --id C... --out-file pause.wasm
```

1

```rust
mod pause {
    soroban_sdk::contractimport!(file = "pause.wasm");
}

#[test]
fn test() {
    let env = Env::default();
    let pause = env.register(pause::WASM, ());
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

2

3

- Register it

TESTING WITH REAL DEPENDENCIES

```
$ stellar contract fetch --id C... --out-file pause.wasm
```

```rust
mod pause {
    soroban_sdk::contractimport!(file = "pause.wasm");
}

#[test]
fn test() {
    let env = Env::default();
    let pause = env.register(pause::WASM, ());
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

• When do we write these types of tests? Whenever we're dependent on another contract.
• Even if you're building to an interface that'll be implemented by many contracts, we benefit from testing with at least some implementations. Doing so can identify assumptions, and areas the interface has left as undefined behavior.
• The bottom line is, don't make the first time your contract calls a third party the day it is deployed.
• You can test locally much faster than it takes to submit transactions on testnet or mainnet, and you get all the benefits of testing locally, like having a step-through-debugger.
>The next strategy is…

TESTING WITH REAL DATA

TESTING WITH REAL DATA

• Testing with real data.
• In the last example we had the real contract loaded, but not real state.
• Our ability to test the protocol we're building was limited to our own knowledge of this other contract and the different states it can get into.
• If the pause contract has relatively complex state behind it, and can return more values than just true or false. It's unlikely that we can exactly replicate the real contract behavior. We have a gap between our test and reality.
• What we can do to close that gap is write tests so that they are using not only mainnet dependencies, but mainnet data.

```
$ stellar snapshot create --address C...  --out snapshot.json
```

TESTING WITH REAL DATA

• The stellar-cli’s snapshot command can create a snapshot of ledger state from mainnet.
• This command will create a snapshot of the pause contract, and its contract state, and write it to a file, snapshot.json.

TESTING WITH REAL DATA

```
$ stellar snapshot create --address C... --out snapshot.json
```

```rust
#[test]
fn test() {
    let env = Env::from_ledger_snapshot("snapshot.json");

    // ...
}
```

• We can then create an environment in our test using that snapshot.
• There's no need to call register for the Pause contract, because when the environment loads that snapshot, it'll have the real pause contract already deployed.
• Just as before, the test is going to be run with the real dependency, but this time it'll also have real data.
• And just as before, nothing else in the test needs to change…

TESTING WITH REAL DATA

```
$ stellar snapshot create --address C... --out snapshot.json
```

```rust
#[test]
fn test() {
    let env = Env::from_ledger_snapshot("snapshot.json");
    let admin = Address::generate(&env);
    let pause = Address::from_str(&env, "C...");
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

- We're still writing our test the same way, with the same test setup, and same test assertions, the same tools, the same patterns.

TESTING WITH REAL DATA

```
$ stellar snapshot create --address C... --out snapshot.json
```

```rust
#[test]
fn test() {
    let env = Env::from_ledger_snapshot("snapshot.json");
    let admin = Address::generate(&env);
    let pause = Address::from_str(&env, "C...");
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

• This approach is helpful for making sure that when you're integration testing with your dependencies, that you're getting the best view possible into how they'll behave on mainnet.

TESTING WITH REAL DATA

```
$ stellar snapshot create --address C... --out snapshot.json
```

```rust
#[test]
fn test() {
    let env = Env::from_ledger_snapshot("snapshot.json");
    let admin = Address::generate(&env);
    let pause = Address::from_str(&env, "C...");
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

• This approach is also helpful once you have an established contract and you’re making changes or fixing a bug. Being able to test that new feature or bug fix against mainnet data is a bigger opportunity to see how it’ll behave in the real world.

TESTING WITH REAL DATA

```
$ stellar snapshot create --address C... --out snapshot.json
```

```
#[test]
fn test() {
    let env = Env::from_ledger_snapshot("snapshot.json");
    let admin = Address::generate(&env);
    let pause = Address::from_str(&env, "C...");
    let id = env.register(Token, (&admin, &pause));
    let token = TokenClient::new(&env, &id);

    // ...
}
```

• An area that's open to explore with testing with real data that I'd love to explore, is how to test with data from other chains. If you're porting a contract from the EVM to Soroban, you may want to consider how you can test your Soroban contract with mainnet data from other chains as a starting point.
> The next strategy is…

FUZZING

FUZZING AND PROPERTY TESTING

- Fuzzing and property testing.
- Fuzzing is the process of providing random data to programs to identify unexpected behavior.
- When using the term fuzzing people are usually talking about discovering ways to crash an application, and they’re not really concerned with testing the intended behavior of the program.

FUZZING

FUZZING AND PROPERTY TESTING

- Property testing is very similar.
- Property tests also accept random input, just like fuzzing, but you write assertions for some property of the program. So property testing is sort of the reverse, it cares a lot about the intended behavior of the program.
- Most property testing frameworks are lightweight in terms of functionality. They generate random inputs, test them, and then discard them.

FUZZING

FUZZING

AND PROPERTY TESTING

• Most fuzzers provide more functionality, and do what's called coverage-guided testing. That's where they generate random inputs, test them, and watch what code paths get executed by the test. The fuzzers goal is to increase test coverage, to find new inputs that execute new lines of code until all code paths are covered. Each time the fuzzer finds an input that executes new lines of code, it stores the input on disk in what's called a corpus, and then when you run the fuzzer again it works to expand that corpus to increase the code coverage.

FUZZING

FUZZING AND PROPERTY TESTING

• I'm only going to show how to use fuzzing tools, because we can do property testing with a fuzzer. Fuzz tools also have much more to offer with the added benefit of the coverage guidance, and the fact they build a corpus that you can continue to grow over time.
• If you really want to do property testing without a fuzzer then you could check out the proptest crate, it's fantastic at what it does and is very easy to use, but I find I often reach for a fuzzer over a property testing framework.

FUZZING

AND PROPERTY TESTING

FUZZING

- The reason to do fuzzing and property testing is that when we're writing tests by hand there's only so many inputs, and combinations of inputs, that we're going to test.
- Like in the bar story earlier, even if we do a pretty decent job of testing out-of-the-box inputs like -1 Drink, there will be inputs we don't think of.

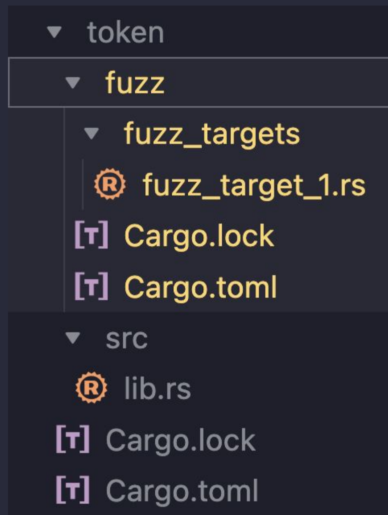
```
$ cargo install --locked cargo-fuzz
```

FUZZING AND PROPERTY TESTING

- In the Rust ecosystem a popular tool for fuzzing is cargo-fuzz, and you can be install it with cargo.
- To use the cargo-fuzz tool on a contract, the best starting point is to…

FUZZING AND PROPERTY TESTING

```
$ cargo fuzz init
```



- To use the cargo-fuzz tool on a contract, the best starting point is to…
- run the init command of cargo-fuzz in your project.
- It will write out the boilerplate directory structure for where your fuzz tests can live.

```
$ cargo fuzz init
```

FUZZING AND PROPERTY TESTING

```
fuzz_target!(|input: Input| {
    // ...
});
```

- Defining a fuzz test means writing one of these fuzz target blocks, that accepts a type that'll be generated by the fuzzer.
- In the example the fuzzer will call the fuzz target with a random Input value, when we run the…

FUZZING AND PROPERTY TESTING

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```
fuzz_target!(|input: Input| {
    // ...
});
```

- …cargo fuzz run command.
- The sanitizer thread option is a work around for some bugs in the default sanitizer.
- In this example we're going to write a fuzz test that sets up two accounts, and then transfers an amount between them.
- So we'll define the Input as…

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

FUZZING AND PROPERTY TESTING

```rust
fuzz_target!(|input: Input| {
    // ...
});
```

```rust
#[derive(Arbitrary)]
pub struct Input {
    pub a: i128,
    pub b: i128,
    pub amount: i128,
}
```

- …a struct with a starting balance amount for address a and address b
- and an amount to transfer between them.

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

FUZZING AND PROPERTY TESTING

```rust
fuzz_target!(|input: Input| {
    let env = Env::default();
    let admin = Address::generate(&env);
    let id = env.register(Token, (&admin,));
    let token = TokenClient::new(&env, &id);

    let a = Address::generate(&env);
    let b = Address::generate(&env);
    _ = token.mock_all_auths().try_mint(&a, &input.a);
    _ = token.mock_all_auths().try_mint(&b, &input.b);

    // ...
```

```rust
#[derive(Arbitrary)]
pub struct Input {
    pub a: i128,
    pub b: i128,
    pub amount: i128,
}
```

- The test setup is much the same as what we saw in the previous examples
- we're using the inputs from the input parameter, instead of hardcoded values.
- we're also ignoring any errors that arise from the mint functions because this fuzz test will be asserting only on the behavior of the transfer function
- The main difference in code is when we write the assertion…

FUZZING AND PROPERTY TESTING

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```
fuzz_target!(|input: Input| {

   // ...

   match token.mock_all_auths().try_transfer(&a, &b, &input.amount) {
       Ok(Ok(())) => {} // Expected success
       Err(Ok(_)) => {} // Expected error
       Ok(Err(_)) => panic!("success with wrong type returned")
       Err(Err(_)) => panic!("unrecognised error")
   }
});
```

- The main difference in the assertions…
- Instead of asserting on a specific outcome, we assert on a range of valid outcomes based on the variety of inputs that could be passed in.

FUZZING AND PROPERTY TESTING

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```
fuzz_target!(|input: Input| {

   // ...

   match token.mock_all_auths().try_transfer(&a, &b, &input.amount) {
       Ok(Ok(())) => {} // Expected success
       Err(Ok(_)) => {} // Expected error
       Ok(Err(_)) => panic!("success with wrong type returned")
       Err(Err(_)) => panic!("unrecognised error")
   }
});
```

- For the token contract we know that the transfer can succeed, or fail. If it fails we know there's that small set of errors in the error enum it should return. We don't expect it to fail with other errors or with a system error. So we can write our assertion to expect success with no value returned, or failure with one of the predefined errors.
- When we call the contract function we use the try_transfer function on the client instead of the transfer function. The same contract function gets called, the client just gives us an opportunity to handle errors.
- The try functions return a nested Result. Which I understand looks a bit odd.

FUZZING AND PROPERTY TESTING

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```rust
fuzz_target!(|input: Input| {

    // ...

    match token.mock_all_auths().try_transfer(&a, &b, &input.amount) {
        Ok(Ok(())) => {} // Expected success
        Err(Ok(_)) => {} // Expected error
        Ok(Err(_)) => panic!("success with wrong type returned")
        Err(Err(_)) => panic!("unrecognised error")
    }
});
```

- The first layer of the Result is Ok if the function call succeeded, and Err if it failed.

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

FUZZING AND PROPERTY TESTING

```
fuzz_target!(|input: Input| {

    // ...

    match token.mock_all_auths().try_transfer(&a, &b, &input.amount) {
        Ok(Ok(())) => {} // Expected success
        Err(Ok(_)) => {} // Expected error
        Ok(Err(_)) => panic!("success with wrong type returned")
        Err(Err(_)) => panic!("unrecognised error")
    }
});
```

- The second layer of the result is Ok when the type matches the type expected, or Err when the value returned is of some other type.
- For the err case, the second layer will be error, if the error returned is a system error, or an contract panic, or a contract error not included in the contract's error enum, so that's the case we're really interested in.
- When does this function in ways that we didn't anticipate it would fail.

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

FUZZING AND PROPERTY TESTING

```rust
fuzz_target!(|input: Input| {

    // ...

    match token.mock_all_auths().try_transfer(&a, &b, &input.amount) {
        Ok(Ok(())) => {} // Expected success
        Err(Ok(_)) => {} // Expected error
        Ok(Err(_)) => panic!("success with wrong type returned")
        Err(Err(_)) => panic!("unrecognised error")
    }
});
```

- When we run the fuzzer with the run command, it’s a different experience. This is because the fuzzer is generating random inputs and will run forever until you stop it.
- Typically it’s something you’ll run for some amount of time, or on a server somewhere and just keep running, and potentially check in and collect the corpus from time to time.
- Cargo fuzz also has the option when running to only run the corpus then exit. If you’re committing your corpus to git, it can be a good test to add to your CI as part of your regular testing.

FUZZING AND PROPERTY TESTING

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```
fuzz_target!(|input: Input| {

  // ...

  match token.mock_all_auths().try_transfer(&a, &b, &input.amount) {
      Ok(Ok(())) => {} // Expected success
      Err(Ok(_)) => {} // Expected error
      Ok(Err(_)) => panic!("success with wrong type returned")
      Err(Err(_)) => panic!("unrecognised error")
  }
});
```

- If you're thinking, what types of contracts or functions should be fuzzed? Pretty much all of them. All software has the potential to benefit from fuzzing, and when we get into the next strategy, we'll find another way we can use fuzzing too.

> The next strategy is…

INVARIANTS

INVARIANTS

- To leverage invariants.
- Invariants are something all software has, it’s not a type of test, more so, it’s an approach to defining expectations about software.
- An invariant is something that’s consistent about a program.
- We talked about property testing before, and it’s similar to that, except with invariants we’re normally talking about things that remain true about the state of the program, rather than properties of it’s behavior.
- For example, an invariant for a token contract might be that…all balances are greater or equal to zero.

```
assert!(all_balances ≥ 0);
```

INVARIANTS

• …all balances are greater or equal to zero.
• With our earlier tests we wrote a clear series of steps that arrived at an assertion.
• With fuzzing we wrote a clear series of steps with varying inputs that arrived at a set of valid outcome.
• With an invariant test we're setting out to prove that no matter what series of steps take place, the invariant remains true.
• There are a few ways to implement invariants in contracts. One is to check the invariant at runtime rather than in a test.
• For example, in our token contract an invariant assertion could be added at runtime to…

ASSERTING INVARIANTS

```rust
pub fn mint(env: &Env, to: Address, amount: i128) {
    // ...
    Self::assert_balances_gte_zero(env, &[to]);
}

pub fn transfer(env: &Env, from: Address, to: Address, amount: i128) {
    // ...
    Self::assert_balances_gte_zero(env, &[from, to]);
}

fn assert_balances_gte_zero(env: &Env, addrs: &[Address]) {
    for addr in addrs {
        assert!(Self::balance(env, addr.clone()) >= 0);
    }
}
```

- …check that any balance of any address that the current execution touches remains greater or equal to zero.
- We add into every place that updates contract balances that the balances must be greater than or equal to zero.
- With a check like this, even if we had a bug that created the invalid state, we'd prevent it, assuming we put this check in all the right places.
- We do have to be careful adding assertions at runtime, since you could inadvertently wedge your contract which depending on the state you're protecting may actually be worse than letting it get into an invalid state. We have to consider what actions are being blocked by an assertion. As the contract developer that's really in your hands to decide whether it's appropriate to use.
- If you only want these assertions to occur when the contract is being tested or debugged, then you can use…the debug_assert macro instead of the assert macro.

ASSERTING INVARIANTS

```
pub fn mint(env: &Env, to: Address, amount: i128) {
    // ...
    Self::assert_balances_gte_zero(env, &[to]);
}

pub fn transfer(env: &Env, from: Address, to: Address, amount: i128) {
    // ...
    Self::assert_balances_gte_zero(env, &[from, to]);
}

fn assert_balances_gte_zero(env: &Env, addrs: &[Address]) {
    for addr in addrs {
        debug_assert!(Self::balance(env, addr.clone()) >= 0);
    }
}
```

• the debug_assert macro instead of the assert macro.
• The debug_assert macro code is optimised away unless the debug_assertions build flag is set, and it's not set on release builds.
• Then your insertions only run in tests, or if you build your contract with debug assertions enabled.
• This is great this does level up our testing…
• But the goal with testing invariants is that we're setting out to prove that no matter what series of steps take place, the invariant remains true. By placing these invariants in the application we're going to be checking them for the large variety of ways that an application gets tested. But not all ways that an contract might get called. Because there's series of calls that might occur on a contract that we fail to imagine, and fail to include in our tests.
• So the next thing we can do with these invariants is to run fuzz tests that randomize not only the inputs, but also what functions get called, and the order they get called in…

FUZZING INVARIANTS

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```
#[derive(Arbitrary)]
pub enum Action {
   Mint(<Address as SorobanArbitrary>::Prototype, i128),
   Balance(<Address as SorobanArbitrary>::Prototype),
   Transfer(
       <Address as SorobanArbitrary>::Prototype,
       <Address as SorobanArbitrary>::Prototype,
       i128,
   ),
}

fuzz_target!(|actions: std::vec::Vec<Action>| {
```

- Now writing a test that does this type of invariant test requires a bit of boilerplate, and i've got to spread it over a couple slides.
- We define an enum, here it is called Action, that contains the functions we want the fuzzer to call and their inputs.
- We make the test accept as input a random list of the actions.
- And this is the test…

FUZZING INVARIANTS

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```
fuzz_target!(|actions: std::vec::Vec<Action>| {
    let env = Env::default();
    env.mock_all_auths();
    let id = env.register(Token, (&Address::generate(&env),));
    let token = TokenClient::new(&env, &id);
    for a in actions {
        match a {
            Action::Mint(addr, amount) => _ = token.try_mint(&addr.into_
            Action::Balance(addr) => _ = token.try_balance(&addr.into_va
            Action::Transfer(from, to, amount) => _ = token.try_transfer
        }
    }
    assert_balances_gte_zero(env);
```

- And this is the test…
- After the typical test setup, it iterates over the list of actions, doing each action, ignoring whether calls succeed or fail.
- Then it runs its invariant checks.

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

FUZZING INVARIANTS

```
fuzz_target!(|actions: std::vec::Vec<Action>| {
    let env = Env::default();
    env.mock_all_auths();
    let id = env.register(Token, (&Address::generate(&env),));
    let token = TokenClient::new(&env, &id);
    for a in actions {
        match a {
            Action::Mint(addr, amount) => _ = token.try_mint(&addr.into_
            Action::Balance(addr) => _ = token.try_balance(&addr.into_va
            Action::Transfer(from, to, amount) => _ = token.try_transfer
        }
    }
    assert_balances_gte_zero(env);
```

• Now it's easy to write a test like this, that won't actually test that many useful scenarios.
• If every balance call returns 0 because it's always using a random generated address that doesn't match an address that has been minted.
• Or if every transfer call fails because the from address doesn't have a balance.
• Then we're not actually going to be testing that many code paths.
• So we do have to put some effort in when writing these types of tests to make the actions meaningful, without constraining the steps that can take place.
• One way to do that is to pass in more than a list of actions, pass in a set of addresses, and a list of actions that will be performed with that smaller set of addresses.
• I haven't shown that here because I needed to limit the amount of code on the slide, but please try that out.

FUZZING INVARIANTS

```
$ cargo fuzz init
$ cargo +nightly fuzz run --sanitizer=thread fuzz_target_1
```

```rust
fuzz_target!(|actions: std::vec::Vec<Action>| {
    let env = Env::default();
    env.mock_all_auths();
    let id = env.register(Token, (&Address::generate(&env),));
    let token = TokenClient::new(&env, &id);
    for a in actions {
        match a {
            Action::Mint(addr, amount) => _ = token.try_mint(&addr.into_
            Action::Balance(addr) => _ = token.try_balance(&addr.into_va
            Action::Transfer(from, to, amount) => _ = token.try_transfer
        }
    }
    assert_balances_gte_zero(env);
```

- But, this is a test type that with an appropriate amount of effort can really push the boundary of testing beyond what our imagination is able to consider.
- The final strategy is…

DIFFERENTIAL TESTING

DIFFERENTIAL TESTING

- Differential testing.
- Differential testing is testing two things, that we expect to behave the same, and asserting that they do in fact behave the same.

DIFFERENTIAL TESTING

DIFFERENTIAL TESTING

• This strategy can be used in the context of regular tests, like unit tests, or can be used in the context of fuzzing as well.
• Effective when you're building something now that you want to make sure behaves like something that already exists.
• So a great example of that is:
- when you release a new version of a contract that adds some new behavior, and some of the contract has been refactored.
• You want to make sure that all your existing contract behavior acts exactly the same as it used to.
• So this really involves running two contracts with the same function calls, then comparing things you want to be the same.
> All contracts built with the Rust Soroban SDK actually have a form of differential testing built-in and enabled by default. If you've noticed when you build a contract…test snapshot files get saved to disk.

DIFFERENTIAL TESTING

```
▸ src
▾ test_snapshots
  ▸ bytes
  ▸ map
  ▾ tests
    ▾ auth
      ▾ auth_10_one
        test.1.json
      ▾ auth_15_one_repeat
        test.1.json
      ▾ auth_17_no_consume_requirement
        test.1.json
      ▾ auth_20_deep_one_address
        test_auth_tree.1.json
        test.1.json
```

- If you've noticed when you build a contract…
- test snapshot files get saved to disk. You'll see something like this in your project directory.
- If we refactor an area of our contract, we should be able to expect that not only did our tests pass, but the final state of the ledger was identical, and we can tell that's the case if the test snapshot file doesn't change.
- We can use these test snapshots to detect unintentional changes when refactoring, upgrading dependencies, upgrading the SDK, or preparing for an upcoming Stellar protocol upgrade.
- These test snapshots capture all the auths that occurred, the events that were published, and the final ledger state.

DIFFERENTIAL TESTING

```
▸ src
▾ test_snapshots
  ▸ bytes
  ▸ map
  ▾ tests
    ▾ auth
      ▾ auth_10_one
          test.1.json
      ▾ auth_15_one_repeat
          test.1.json
      ▾ auth_17_no_consume_requirement
          test.1.json
      ▾ auth_20_deep_one_address
          test_auth_tree.1.json
          test.1.json
```

- If we want to test two different contracts in the same test, we can do that too.
- So imagine we've release version 1 of our token, and are working on version 1.1 and want to test our token contract against v1 before we upgrade.
- We can…download the currently deployed contract with the stellar-cli by using the fetch command.

```
$ stellar contract fetch --id C...  --out-file token.wasm
```

DIFFERENTIAL TESTING

• We can…download the currently deployed contract with the stellar-cli by using the fetch command.
• And then…import that contract into our test file to use in the test environment.

```
$ stellar contract fetch --id C...  --out-file pause.wasm
```

```rust
mod token {
    soroban_sdk::contractimport!(file = "token.wasm");
}
```

DIFFERENTIAL TESTING

- And then…import it into our test file to use in the test environment.
- This is exactly the same thing we did earlier when importing a dependency to test with.
- Once imported, we can write a test…

DIFFERENTIAL TESTING

```
#[test]
fn test() {
    assert_eq!({
        let env = Env::default();
        let id = env.register(Token, (&Address::generate(&env),));
        let token = TokenClient::new(&env, &id);
        let a = Address::generate(&env);
        let b = Address::generate(&env);
        token.mock_all_auths().mint(&a, &10);
        token.mock_all_auths().transfer(&a, &b, &2);
        (token.balance(&a), token.balance(&b))
    }, {
        let env = Env::default();
        let id = env.register(pause::WASM, (&Address::generate(&env),));
        let token = TokenClient::new(&env, &id);
        let a = Address::generate(&env);
        let b = Address::generate(&env);
        token.mock_all_auths().mint(&a, &10);
        token.mock_all_auths().transfer(&a, &b, &2);
        (token.balance(&a), token.balance(&b))
    });
}
```

- Once imported, we can write a test…
- That registers the native token, and the wasm token, runs a series of identical steps, and then asserts that the balances after the transfers matches.
- The Soroban SDK's test framework tries really hard to not generate random data, and produce predictable values, like new addresses, which makes it easier to write tests like this and be able to compare the outcomes.
- Differential tests work really well inside a fuzzer for testing that two contracts behave the same across a wide variety of inputs.

> Something you'll notice with all the testing strategies…we talked about

1. TESTING WITH REAL DEPENDENCIES
2. TESTING WITH REAL DATA
3. FUZZING AND PROPERTY TESTING
4. INVARIANTS
5. DIFFERENTIAL TESTING

- Something you'll notice with all the testing strategies…we talked about
- Is that these are not mutually exclusive.
- For example, you can test real dependencies, and real data, invariants, and differentials, inside fuzz tests. You can use real contracts and data when running invariant or differential tests.

1. TESTING WITH REAL DEPENDENCIES
2. TESTING WITH REAL DATA
3. FUZZING AND PROPERTY TESTING
4. INVARIANTS
5. DIFFERENTIAL TESTING

• Also, this isn't an exhaustive list of strategies. They're a few solid strategies for rigorously testing, and applying rigorous skepticism to contract development.
• There are some other important topics to do with testing on Soroban, such as how to test state archival, and so I recommend visiting the developers.stellar.org website to learn more.
• Tomorrow there's a talk and workshop by Certora who will be demonstrating another testing strategy…formal verification.

1. TESTING WITH REAL DEPENDENCIES
2. TESTING WITH REAL DATA
3. FUZZING AND PROPERTY TESTING
4. INVARIANTS
5. DIFFERENTIAL TESTING
6. FORMAL VERIFICATION

- …formal verification.
- I believe they're going to take the token contract implementation that's behind this talk, and demonstrate how to formally verify it. So highly recommend joining that talk to collect more tools to test your contracts.

1. TESTING WITH REAL DEPENDENCIES
2. TESTING WITH REAL DATA
3. FUZZING AND PROPERTY TESTING
4. INVARIANTS
5. DIFFERENTIAL TESTING
6. FORMAL VERIFICATION

- All this testing requires a lot of effort. It has a cost on time. But testing is a critical part of making reliable software, and reliable contracts.

Karen N Johnson, who is someone who has had a lot to say about software testing, said…

THE EARLIER A BUG IS FOUND, THE CHEAPER IT IS TO FIX.

—Karen N. Johnson

- The earlier a bug is found,
- The cheaper it is to fix.
- The cost of a bug can measured in a lot of different ways.

- The cost in time to fix.
- The cost to users if the fix is actually a breaking change.
- The cost to the design, if the bug was the result of a design flaw.

THE EARLIER A BUG IS FOUND, THE CHEAPER IT IS TO FIX.

—Karen N. Johnson

- For contracts though, cost can be a lot less ambiguous, when bugs make it to mainnet everything is at stake.
- So this is our call to test. And not just test, but to employ rigorous skepticism.

Stellar

TEST!

___leigh___

___leigh___

leighmcculloch

© 2024 Stellar Development Foundation