Crowdfunding Dapp with Reach

Finished Dapp repo: https://github.com/JossDuff/reach-crowdfund/
If this tutorial is helpful at all please leave a star :)

Disclaimer: I'm not affiliated with Reach and this project is not audited and should not be used in a production environment.

Joss Duff

October 24, 2022

Reach

JavaScript

Requirements

1. An IDE to code. Visual Studio Code is recommended because of its Reach extension that supports Reach syntax highlighting, keyword hover documentation, real time diagnostic highlighting for Reach compilation errors, etc.
2. Basic understanding of how to run Reach apps. Reach download instructions
3. Docker - used in running Reach apps. get Docker

Background

CROWDFUNDING

> "The practice of funding a project or venture by raising money from a large number of people, in modern times typically via the Internet." -Wikipedia

Most people have contributed to a crowdfunding campaign on a popular website like Kickstarter, GoFundMe, Indiegogo, or Patreon. Crowdfunding is a great example of an application that can be improved upon in Web3. Currently, crowdfunding companies act as a trusted intermediary and control the collection and distribution of funds as well as the taking of fees to turn a profit. This can all be replaced with smart contracts, removing the need for a trusted intermediary and any fees needed to sustain a whole company without any loss of functionality.

project (the "receiver"). They specify a goal amount of money and a deadline to donate funds.
2. Anybody can donate money to the fund until it reaches its deadline (the "funders").
3. The deadline is reached and the fund does not accept any more donations.
4. Two possibilities:
   - The total amount of money donated is greater than or equal to the goal amount specified by the receiver. In this case, all the money donated is given to the receiver.
   - The total amount of money donated is less than the goal amount specified by the receiver. In this case, each funder receives back their donation.

REACH

> "Reach is a domain-specific language for building decentralized applications (DApps)." -Reach docs

This tutorial covers the building of a crowdfunding decentralized application using Reach, a blockchain development platform for writing smart contracts and decentralized applications. Reach aims to remove the possibility to introduce common smart contract bugs in your program by abstracting away the raw smart contract coding so the programmer can focus on the business logic of their application. No prior smart contract coding is necessary to build a full decentralized application using Reach.

A reach program consists of two files. A frontend `index.js` and a backend `index.rsh`. You'll recognize that the frontend is a common JavaScript file, but you probably aren't familiar with the backend file extension. ".rsh" is Reach's own language for specifying actions of participants in a DApp. Reach programs compile down into a smart contract.

Steps

12. Run the tests

13. Done!

14. Improve your skills and test your knowledge

1. Setup

In a terminal, navigate to the reach folder that you created when installing reach.
Create a folder inside the reach folder for this project:
`$ mkdir reach-crowdfund`
`$ cd reach-crowdfund`

Now, create a template reach program:
`$ ../reach init`

This creates the index.rsh and index.mjs template files required for a basic Reach DApp. It allows you to open the files and start writing code. The index.rsh file is the DApp and is written in Reach, and the index.mjs file is the frontend of the DApp and is written in JavaScript.

Go ahead and run this template program:
`$ ../reach run`

Your output should look like:

2. Start the backend index.rsh

Replace the contents of your index.rsh file with the following:

```
'reach 0.1';

export const main = Reach.App(() => {

  // Receiver is the address creating the fund to receive currency
  // It is the only participant
  const Receiver = Participant('Receiver', {

    // Gets the parameters of a fund
    // By default, these values are local and private
    receiverAddr: Address,
    deadline: UInt,
    goal: UInt,

    // For indicating to the frontend that the contract is deployed
    // and the fund is ready to receive payments.
    ready: Fun([], Null),
  });
```

```
  back (if fund doesn't reach
    // its goal) from a fund
    const Funder = API ('Funder', {
      // payFund function takes the amount
  that the funder wants
      // to donate to the fund as a UInt.
      donateToFund: Fun([UInt], Bool),
      // Pays the funder back if the fund
  didn't reach the goal.
      // Returns the amount the funder
  previously donated.
      payMeBack: Fun([], Bool),
    });

    // Bystander API for any address to use.
    const Bystander = API ('Bystander', {
      // Indicates that the fund has reached
  its deadline.
      timesUp: Fun([], Bool),
      // For displaying whether or not fund
  met its goal.
      getOutcome: Fun([], Bool),
    });

    init();
  });
```

- `const Receiver = Participant('Receiver', {...});` is the address creating the fund to receive currency. This uses Reach's `participant` and is the only `participant` in this tutorial. A participant is a logical actor which takes part in a DApp. It is associated with an account on the consensus network. A participant has persistently stored values, called its local state. It has a frontend which it interacts with. A frontend is an abstract actor which supports

- `receiverAddr: Address, deadline: UInt, goal: UInt,` are the parameters of the fund that the receiver will be creating. This values are specified in the frontend. Since these are defined within a participant class, they start out local and private variables known only to the receiver. In the next step we will `declassify` these variables to make them public, and then `publish` them to put them on the blockchain.

- `ready: Fun([], Null),` is the function signature for a function that is defined in the front end. Argument types are specified in [], and the return type follows. You can see this function takes no arguments and has no return type. We will call this later in this file simply to signal to the frontend that the fund is ready to receive donations.

- `const Funder = API ('Funder', {...});` is any address that donates to the fund. Instead of participant, this is a Reach API. APIs are functions that can be called by other contracts, as well as off-chain. Functions inside of an API definition (ex: donateToFund) are called API member functions and must be called exactly once during the execution of a Reach program. Read more about APIs.

- `donateToFund: Fun([UInt], Bool)` is for handling donations to the fund. Any address can call this function with a UInt (amount to donate). Returns a boolean.

- `payMeBack: Fun([], Bool)` is a function for paying back funders in the event the fund does not meet its goal. It takes no arguments because we are paying back funders only their initial donation, which is known to the dapp.

- `const Bystander = API ('Bystander', {...});` is a bystander API. Bystander functions are callable by anyone to progress the dapp.

- `timesUp: Fun([], Bool),` for indicating to the frontend that the funding period has reached its deadline and should

frontend whether or not the fund has met its goal.

- `init();` finalizes all of the available participants, views, and compilation options. Below this we will write the logic of our crowdfunding dapp.

---

3. Setting up the fund

Add the following to index.rsh after init(); but before the closing `});`:

```
  // Receiver declassifies details of the
fund.
  Receiver.only(() => {
    const receiverAddr =
declassify(interact.receiverAddr);
    const deadline =
declassify(interact.deadline);
    const goal =
declassify(interact.goal);
  });

  // The funder publishes the parameters
of the fund to the
  // blockchain.
  // Publish initiates a consensus step
and makes the values
  // known to all participants.
  Receiver.publish(receiverAddr, deadline,
goal);
```

```
  // Indicate to the frontend that the
fund is ready.
  Receiver.interact.ready();

  // Mapping to keep track of amount that
each address donates
  // to the fund.
  const funders = new Map(Address, UInt);
  // Set for tracking which addresses have
donated.
  // Used verifying an address has donated
in the payMeBack
  // function.
  const fundersSet = new Set();
```

- `Receiver.only(()=>{...})` is an action that only the receiver participant makes. For our program we are having the receiver declassify the parameters of the fund. Declassifying turns a local private variable (default initial state) into a local public variable that we can use in our backend. However, we want anyone to be able to view the parameters of the fund so we then `Receiver.publish(...)` the declassified variables. Publish is a type of consensus transfer where the variables given to it are made public on the blockchain for all participants to see.

- `commit();` moves the dapp from local state to consensus state. The computations between “commit();”s are in local state, meaning they are being computed for the individual participant. Those computations are bundled together and published to the blockchain when the dapp moves to consensus state. States (“steps”) are a fundamental concept of the Reach model. I found this diagram to be extremely helpful when developing in Reach. Article with diagram.

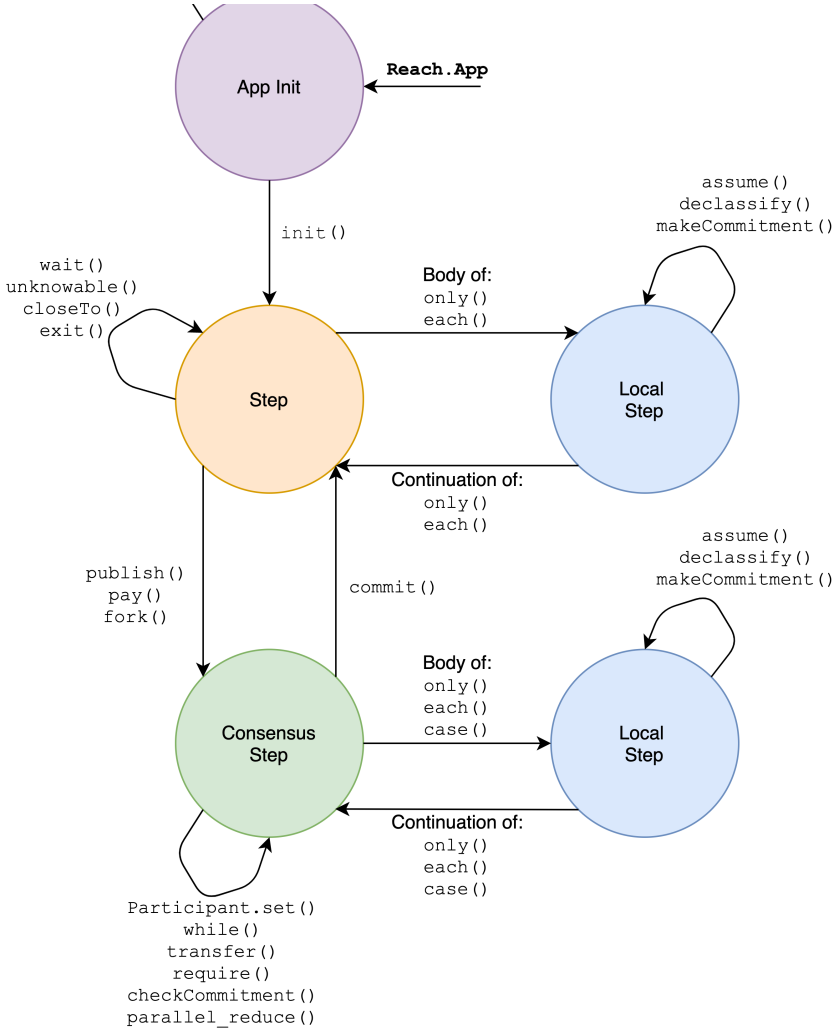

Figure 3-1 States (steps) are a fundamental concept.

- `Receiver.interact.ready();` Calls the 'ready()' function (we wrote the signature for it in step 1) to indicate to the frontend that the fund parameters are published and the fund is ready to receive donations.
- `const funders = new Map(Address, UInt);` is a mapping from addresses to a UInt to keep track of how much each address has donated to the fund.
- `const fundersSet = new Set();` is a set of addresses for an additional verification of which address has donated to the fund.

4. Funding period

This is the logic for accepting and tracking donations for a set amount of time.
Add the following to index.rsh after `const fundersSet = new Set();` but before the closing `});`:

```
  // ParallelReduce to allow for any
address to donate to a fund.
  // Address can donate until fund
deadline is reached.
  const [ keepGoing, fundBal, numFunders ]
=
  // fundBal and numFunders starts at 0
and keepGoing starts as true.
  parallelReduce([ true, 0, 0 ])
    // Define block allows you to define
variables/functions that
    // are used in the different cases.
    .define(()=>{
      const checkDonateToFund = (who,
donation) => {
        // Checks that the funder hasn't
donated yet
        check( !fundersSet.member(who),
"Not yet in set");
        // Doesn't allow donations of 0
        check( donation != 0, "Donation
equals 0");
        return () => {
          // Adds the funder to the
mapping with their donation.
          funders[who] = donation;
          // Adds the funder to the set of
```

```
with the new fund balance
          // and number of funders.
          return [ keepGoing, fundBal +
donation, numFunders + 1 ];
        };
      };
    })
    // Loop invariant helps us know things
that are true every time
    // we enter and leave loop.
    .invariant(
      // Balance in the contract is at
least as much as the total
      // amount in the fund
      balance() >= fundBal
      // The number of funders in the map
is the same as the
      // number of funders tracked by the
parallel Reduce.
      && fundersSet.Map.size() ==
numFunders
    )
    .while( keepGoing )
    // API function for any address to
call as a funder to
    // donate to the fund.  Is called with
'payment' UInt
    // indicating the amount of network
currency they want
    // to donate.
    .api(Funder.donateToFund,
      // Runs the checks in
checkDonateToFund function define in
      // .define() block of this parallel
Reduce.
      (payment) => { const _ =
```

```
      // and deposits into the contract.
      (payment) => payment,
      (payment, k) => {
        // Returns true for the API call,
indicating it was successful.
        k(true);
        // Calls the function within
checkDonateToFund to update the
        // funder mapping, funder set, and
parallel Reduce.
        return checkDonateToFund(this,
payment)();
      }
    )
    // Things in this block only happen
after the deadline.
    .timeout( relativeTime(deadline), ()
=> {
      // Any bystander calls the timesUp
function, indicating the
      // parallel reduce has finised and
the fund reached its deadline.
      const [ [], k ] =
call(Bystander.timesUp);
      // Returns true for the timesUp API
call, indicating it was successful.
      k(true);
      // Returns false for keepGoing to
stop the parallelReduce.
      return [ false, fundBal, numFunders
];
    });

  // Check to ensure that the balance in
the contract is always
  // greater than or equal to the
```

```
  // Outcome is true if fund met or
exceeded its goal
  // Outcome is false if the fund did not
meet its goal
  const outcome = fundBal >= goal;

  commit();
```

- Here we use a parallelReduce statement so we can loop until a condition is met (fund deadline). You can think of a parallelReduce like a while loop. Like publish, parallelReduce is a type of consensus transfer. Our parallel reduce will allow many funders to donate to the fund at the same time.
- `const [ keepGoing, fundBal, numFunders ]` defines the variables that we will need to update and use in each loop of the parallelReduce and `parallelReduce([ true, 0, 0 ])` starts the parallel reduce and gives our defined variables their initial values (keepgoing=true, fundBal=0, numFunders=0).
- `.define(()=>{...}` Define block allows you to define variables/functions that are used in the different cases of the paralellReduce. In our define block we define `const checkDonateToFund = (who, donation) => {...}` that takes an address and a UInt amount and checks to make sure that that address hasn't donated to the fund yet and returns a function to add the address to the donation mapping and set of funders. For some practice, try to implement functionality for funders to donate multiple times.
- `invariant()` are components of a while loop that we know will always be true before and after each iteration of the loop. In our invariant we assert that the balance in the contract is at least as much as the total amount in the fund

- `.api(Funder.donateToFund,...)` is an api function that is callable throughout the loop. This api function allows anyone to donate an amount to the fund. We wrote the signature to this function back in step 2 (Setting up index.rsh). `Funder.donateToFund` takes an integer and the callers address.

- `(payment) => { const _ = checkDonateToFund(this, payment); }` calls the first part of the function we defined in the 'define' block. This takes the address of the caller and their desired donation (payment int) and checks to make sure they haven't already donated.

- `(payment) => payment,` is called a pay expression. This takes an argument and sends a transaction to the calling address requesting those funds. So here, we are requesting the caller of `Funder.donateToFund` to send the indicated `payment` amount to the contract address. If the caller accepts the transaction then the function continues.

- `(payment, k) => {...}` Passes in arguments for the amount the caller just paid and k which we will use to return the API call.

- `k(true);` returns true for the `Funder.donateToFund` API call. Remember in step 2 (Setting up index.rsh) we indicated that this function returns a boolean.

- `return checkDonateToFund(this, payment)();` Calls the function returned from checkDonateToFund to update the funder mapping, funder set, and parallel Reduce to reflect the new amount of funders and the total amount of currency donated to the contract.

- `.timeout( relativeTime(deadline), () => {...}` This block executes when `relativeTime(deadline)` is reached. Deadline passed in from the front end during the initial fund creation. relativeTime counts until the given amount of network time has passed.

reduce has finished and the fund reached its deadline.

- `return [ false, fundBal, numFunders ];` Returns false for keepGoing to stop the parallelReduce.
- `assert( fundBal <= balance() );` Check to ensure that the balance in the contract is greater than or equal to the calculated balance of the fund after the parallel reduce exits. assert Always evaluates to true and adds this assumption to the tests ran during compilation of a Reach program. It is checking '<=' and not just '==' because at any point anyone can send funds to a smart contract address without using the functions within the smart contract.
- `const outcome = fundBal >= goal;` Outcome is true if fund met or exceeded its goal outcome is false if the fund did not meet its goal.
- `commit();` moves the dapp from local state to consensus state. The computations between "commit();"s are in local state, meaning they are being computed for the individual participant. Those computations are bundled together and published to the blockchain when the dapp moves to consensus state. I found this diagram to be extremely helpful when developing in Reach, as states ("steps") are a fundamental concept of the Reach model.

> ### 5. Sending back donations if the fund met its goal

This handles the case in which the fund met its goal and all the donations are sent to the receiver.

```
  // Bystander views the outcome of the
fund.
  const [ [], u ] =
call(Bystander.getOutcome);
  // Returns outcome.
  u(outcome);

  // If the fund met or exceeded its goal,
pay all the money
  // in the contract to the Receiver.
  if(outcome) {
    // Pays the receiver.
    transfer(balance()).to(Receiver);
    commit();
    exit();
  }

  // If the contract is at this point that
must mean the fund
  // did not meet its goal and the funders
must receive their
  // currency back.
  assert(outcome == false);
```

- `const [ [], u ] = call(Bystander.getOutcome);` Allows any bystander to call the 'getOutcome' function to get the status of the fund. This function takes no arguments (notice the empty argument array `[]`) and it returns a single boolean `u`. The outcome boolean is returned on the following line: `u(outcome);`.

- The `if(outcome)` block handles the case in which the fund was successful and met its goal. Handling this case is easy. First, we transfer out the balance of the contract to the receiver `transfer(balance()).to(Receiver);`. Then

- If `outcome` is false (fund did not meet its goal), then the `if(outcome)` block is not run and we need to handle sending back all the donations to the individual funders. `assert(outcome == false);` Is a check to ensure that if the contract gets to this point, that the fund was not successful.

> ### 6. Sending back donations if the fund DID NOT meet its goal

Sending back funds to each address that donated is a bit trickier. We can't simply loop over the mapping of funders to donations and return each donation back because each transaction has a fee. Theoretically, we could have a single function to pay back all funders, but who would pay the fees to execute all these transactions? If I was a funder, I wouldn't want to have to pay for other funders to receive their donations back. We couldn't simply have the contract pay for the transactions because all the funds in the contract "belong" to their original funder so if we used the contract balance to pay for transactions, not all funders would receive their money back. And asking a receiver to deposit money when they want to create a fund to cover paying back funders would dis-incentivize people from using our crowdfunding dapp. So the best option is to have each funder call a function and pay for the transaction to receive their original donation.

Insert the following after `assert(outcome == false);` but before the closing `});`:

```
  // to call a function to receive their
funds back.
  const [ fundsRemaining,
numFundersRemaining ] =
    parallelReduce([ fundBal, numFunders
])
    .define(()=> {
      const checkPayMeBack = (who) => {
        // Check that the address
previously donated and is in the
        // mapping and set.
        check( !isNone(funders[who]),
"Funder exists in mapping");
        check( fundersSet.member(who),
"Funder exists in set");
        // Unwraps the UInt (amount
doated) of the address in
        // the mapping.
        const amount =
fromSome(funders[who], 0);
        check( amount != 0, "Amount
doesn't equal 0");
        // Checks there is enough currency
in the contract to
        // pay back funder.
        check(balance() >= amount);
        return () => {
          // Transfers back the amount the
funder previously
          // donated
          transfer(amount).to(who);
          // Removes the funder from the
set and sets their
          // mapping to 0.
          funders[who] = 0;
          fundersSet.remove(who);
```

```
        }
      }
    })
    .invariant(
      // Ensures the balance of the
contract is equal
      // to or greater than the amount of
funds remaining
      // in the contract.
      balance() >= fundsRemaining
    )
    // Loop continues until either there
are no funds remaining
    // or all funders reclaimed their
funds.
    .while( fundsRemaining > 0 &&
numFundersRemaining > 0)
    // API for any address to call to
attempt to re-claim
    // their previous donation.
    .api(Funder.payMeBack,
      // Runs the checks in the
checkPayMeBack function
      // from the .define() block.
      () => {const _ =
checkPayMeBack(this); },
      // Pay expression is 0 because we
don't want
      // function callers to pay anything
for this.
      () => 0,
      (k) => {
        // Returns true for the payMeBack
API call,
        // indicating it was successful.
        k(true);
```

```
    the receiver.
        return checkPayMeBack(this)();
      }
    )
```

- Again we will use a parallelReduce statement so we can loop until a condition is met (there are no funds remaining or all funders received their donation back). Similar to step 4 Funding period.
- `const [ fundsRemaining, numFundersRemaining ] = parallelReduce([ fundBal, numFunders ])` starts off the parallel reduce by using `fundBal` and `numFunders` (returned from our parallel reduce in step 4 Funding period) as the starting values for `fundsRemaining` and `numFundersRemaining` respectively. In our parallel reduce in step 4 we incremented these values each time someone made an API call to donate to the fund. In this parallel reduce we will instead decrement these values each time someone makes an API call to receive back their previous donation to the fund.
- In our define block we create a function `const checkPayMeBack = (who) => {...}` that performs some checks to verify that an address has indeed donated an amount to the fund and returns a function that transfers the donation associated with the calling address.
- `const amount = fromSome(funders[who], 0);` Unwraps the UInt (amount dongasated) of the address in the mapping. Entries in mappings are stored as options (called `maybe` in Reach).
- `check(balance() >= amount);` makes sure there is enough money in the contract to pay back the funder. Always evaluates to true. This indicates to the Reach compiler to check for cases when this could be false. The Reach compiler checks all combinations of all possible inputs to

program is behaving as we intend it to. It's good practice to scatter assert, check, and require statements throughout your Reach program to have the compiler check edge cases and ensure your program is working as intended.

- The `checkPayMeBack` function returns a function that transfers out a funders previous donation `transfer(amount).to(who);`, removes the funder from the set of funders and sets their amount donated mapping to 0 `funders[who] = 0; fundersSet.remove(who);`, and finally continues the parallel reduce `return [ fundsRemaining-amount, numFunders-1];` with updated values of `fundsRemaining` and `numFundersRemaining`.
- Our loop invariant `balance() >= fundsRemaining` ensures that there are enough funds in the contract to pay back all of the `fundsRemaining`, which is the amount of funds left to be claimed.
- `.while( fundsRemaining > 0 && numFundersRemaining > 0)` Loop continues until either there are no funds remaining or all funders have reclaimed their funds.
- `.api(Funder.payMeBack, ...}` API for any address to call to attempt to re-claim their previous donation (if they previously donated).
- `() => {const _ = checkPayMeBack(this); },` Runs the checks in the `checkPayMeBack` function from the .define() block. `this` is the address of the function caller.
- `() => 0,` the pay expression is 0 because we don't want function callers to pay any amount to the contract for this.
- `(k) => {...}` passes in `k` to be used as the API return value.

- `return checkPayMeBack(this)();` calls the function returned from the `checkPayMeBack` function to transfer out the donated amount to the caller, remove the address from the set and mapping, and update the values of `fundsRemaining` and `numFundersRemaining`.

  You might notice that this parallel reduce doesn't have the `.timeout()` function that the parallel reduce we wrote in step 4 does. That is because we don't want to have a deadline for funders to claim their donations. This parallel reduce only exits when `while( fundsRemaining > 0 && numFundersRemaining > 0)` evaluates to false.

---

7. Finishing index.rsh

We're almost done with index.rsh. There is just one loose end to tie up.
Insert the following after the parallel reduce we just wrote but before the closing `});`:

```
  // This transfers out any remaining
balance to ensure
  // that no funds are permanently locked
in the contract.
  transfer(balance()).to(Receiver);
  commit();
  exit();
```

permanently locked in the contract.

- Then we `commit();` to publish onto the blockchain and finally `exit();` our program. index.rsh is done!

  Wait, why do we have to transfer out the remaining balance of the contract? Shouldn't the parallel reduce we wrote above handle the distribution of all donations?

  Since a contract is just an address on a blockchain, anyone can send funds to it at any time. Any funds sent to this contract's address without using the donateToFund function aren't tracked and thus nobody can re-claim them during the "pay me back" period if the fund doesn't reach its goal. We view sending funds in this manner as a free donation to the receiver that isn't expecting to be paid back. When this transfer is reached, we have already exited the parallel reduce to pay funders back, so we know that any remaining balance in the contract doesn't belong to any funder (i.e. weren't donated using `donateToFund`).

---

8. Start the frontend index.mjs

Now we will build out the front end. For this tutorial we are just using the frontend for a terminal demo.
Replace the contents of index.mjs with the following:

```
import {loadStdlib} from '@reach-sh/stdlib';
import * as backend from './build/index.main.mjs';
```

```
const runDemo = async (GOAL) => {

  const startingBalance =
stdlib.parseCurrency(100);

  // Set for testing purposes.
  const deadline = 50;
  const FUNDGOAL =
stdlib.parseCurrency(GOAL);

  // Helper function for holding the
balance of a participant
  const getBalance = async (who) =>
stdlib.parseCurrency(await
stdlib.balanceOf(who), 4,);

  // Prints to console the fund goal.
  console.log(`Fund goal is set to
${stdlib.formatCurrency(FUNDGOAL)}`);

  // Creates receiver and 2 funder test
accounts with the starting balance.
  const receiver = await
stdlib.newTestAccount(startingBalance);
  const users = await
stdlib.newTestAccounts(2,
startingBalance);

  // Prints initial balance of the 3
accounts
  for ( const who of [ receiver, ...users
]) {

console.warn(stdlib.formatAddress(who),
'has',
    stdlib.formatCurrency(await
```

```
await runDemo(10);
await runDemo(20);
```

- We're going to run a demo with two different goal amounts (10 and 20). `runDemo` takes a goal amount and parses that amount into the network currency to later pass as a fund parameter.
- `const startingBalance = stdlib.parseCurrency(100);` defines a starting currency of 100 network tokens.
- `const deadline = 50;` is a very short deadline period for demo purposes.
- `const getBalance = async (who) => stdlib.parseCurrency(await stdlib.balanceOf(who), 4,);` is a helper function for getting the balance of a participant
- `const receiver = await stdlib.newTestAccount(startingBalance);` creates 1 receiver test account with a balance of `startingBalance`
- `const users = await stdlib.newTestAccounts(2, startingBalance);` creates 2 funder test account with a balance of `startingBalance`. Since we're creating 2 test accounts, `users` is an array. Ex: access the first user with `users[0]`.
- The for loop prints initial balance of the 3 test accounts we created.
- Then we run the demo with a goal of 10, then again with a goal of 20.

9. Deploying contract from front-end

Add the following to the end of the `runDemo` function, inside the closing `}` but after the for loop:

```
  // Receiver deploys the contract
  const ctcReceiver =
receiver.contract(backend);

  // Since the receiver doesn't need to do
anything after
  // deploying the contract and fund, we
can shut off their thread.
  // This can be done in Reach by using a
try/catch block
  // and throwing an arbitrary error.
  // Inspiration from Jay McCarthy's
session at Reach Summit 2022:
  // https://www.youtube.com/watch?
v=rhgEUFjiI2s&t=5158s
  try {
    await ctcReceiver.p.Receiver({
      receiverAddr:
receiver.networkAccount,
      deadline: deadline,
      goal: FUNDGOAL,
      ready: () => {
        // Defines the receivers ready()
function.
        console.log('The contract is
ready');
        // Arbitrary error.
        throw 42;
      },
```

```
      throw e;
    }
  }
```

- `const ctcReceiver = receiver.contract(backend);` receiver deploys the contract.
- Since the receiver doesn't need to do anything after deploying the contract and fund, we can shut off their thread. This can be done in Reach by using a try/catch block and throwing an arbitrary error. This is inspired by Reach founder Jay McCarthy's session at Reach Summit 2022 (This video is a great resource check it out! McCarthy builds a dapp in under 30 minutes while explaining his thought process).
- `await ctcReceiver.p.Receiver({...})` deploys the contract with the given fund parameters.
- `ready: () => {...}` defines the logic of the receiver's `ready()` function that we call in the back end.

> ### 10. Helper functions

Add the following to the end of the `runDemo` function, inside the closing `}` but after the try/catch:

```
  // Helper function to connect and
address to the contract.
  const ctcWho = (whoi) =>
```

```
  // Helper function to connect an address
to the contract and call
  // the contracts donateToFund function.
  const donate = async (whoi, amount) => {
    const who = users[whoi];
    // Attatches the funder to the backend
that the receiver deployed.
    const ctc = ctcWho(whoi);
    // Calls the donateToFund function
from backend.
    console.log(stdlib.formatAddress(who),
`donated ${stdlib.formatCurrency(amount)}
to fund`);
    await
ctc.apis.Funder.donateToFund(amount);
  };

  // Helper function to call the
contract's timesUp function.
  const timesup = async () => {
    await
ctcReceiver.apis.Bystander.timesUp();
    console.log('Deadline reached');
  };

  // Helper function to call the
contract's getOutcome function
  // and publish it to the frontend.
  const getoutcome = async () => {
    const outcome = await
ctcReceiver.apis.Bystander.getOutcome();
    console.log(`Fund ${outcome? `did` :
`did not`} meet its goal`);
    return outcome;
  };
```

```
function.
  const paymeback = async (whoi) => {
    const who = users[whoi];
    // Attatches the funder to the backend
that the receiver deployed.
    const ctc = ctcWho(whoi);
    // Calls the donateToFund function
from backend.
    await ctc.apis.Funder.payMeBack();
    console.log(stdlib.formatAddress(who),
`got their funds back`);
  };
```

- `const ctcWho = (whoi) => users[whoi].contract(backend, ctcReceiver.getInfo());` helper function to connect an address to the contract. Addresses have to be connected to the contract before calling functions.
- `const donate = async (whoi, amount) => {...}` helper function for a funder to call to donate to the contract during the funding period. Takes an address and the amount the funder wants to donate. `await ctc.apis.Funder.donateToFund(amount);` Calls the donateToFund function from backend.
- `const who = users[whoi];` and `const ctc = ctcWho(whoi);` Attatches the funder (`who`) to the backend that the receiver deployed.
- `const timesup = async () => {...};` is a helper function for a bystander to call the contract's timesUp function.
- `const getoutcome = async () => {...};` is a helper function for a bystander to call the contract's getOutcome function and publish it to the frontend.

function. `await ctc.apis.Funder.paymeBack();` calls the `donateToFund` function from backend.

---

11. Create the tests

Add the following to the end of the `runDemo` function, inside the closing `}` but after the `paymeback` function:

```
  // Test account user 0 donates 5
currency to fund.
  await donate(0,
stdlib.parseCurrency(5));

  // Test account user 1 donates 10
currency to fund.
  await donate(1,
stdlib.parseCurrency(10));

  // Waits for the fund to mature
  console.log(`Waiting for the fund to
reach the deadline`);
  await stdlib.wait(deadline);

  // Anyone calls the timesUp function to
indicate the
  // contract has reached the deadline.
  await timesup();

  // Gets the outcome of the fund.
```

```
  // If the fund didn't meet its goal,
funders call
  // function to get their funds back.
  if(!outcome){
    // Test account user 0 requests their
donation back.
    await paymeback(0);
    // Test account user 1 requests their
donation back.
    await paymeback(1);
  }

  // Prints the final balances of all
accounts
  for ( const who of [ receiver, ...users
]) {

console.warn(stdlib.formatAddress(who),
'has',
    stdlib.formatCurrency(await
stdlib.balanceOf(who)));
  }

  console.log(`\n`);
```

- `await donate(0, stdlib.parseCurrency(5));` Test account user 0 donates 5 currency to fund.
- `await donate(1, stdlib.parseCurrency(10));` Test account user 1 donates 10 currency to fund.
- `await timesup();` A bystander calls the timesUp function to indicate the contract has reached the deadline.
- `const outcome = await getoutcome();` Gets the outcome of the fund. True if fund met its goal, false

funders call function to get their funds back: `await paymeback(0);` (Test account user 0 requests their donation back) and `await paymeback(1);` (Test account user 0 requests their donation back).

- Finally, we print the final balances of all accounts.

---

12. Run the tests

Add the following after the `runDemo` function (outside the closing `}`):

```
// Runs the demo with a fund goal of 10.
await runDemo(10);
// Runs the demo with a fund goal of 20.
await runDemo(20);
```

- `await runDemo(10);` Runs the demo with a fund goal of 10. This will pass since user 0 donates 5 currency and user 1 donates 10 currency.
- `await runDemo(20);` Runs the demo with a fund goal of 20. This will fail since only 15 currency is donated and 20 is the goal. In this case, users call the `paymeback` function to reclaim their funds.

Go ahead and run your crowdfunding dApp.
Output should look like this:

```
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

> index
> node --experimental-modules --unhandled-rejections=strict index.mjs

Fund goal is set to 10
0xECF127ad12e0c8576014FA618fCa3F0CD3890128 has 100
0x418682F9E4F8f2F1eD3aeFBE381b596C97FD8232 has 100
0xB0e5CC81Cb6DA4A91C820210803E3B7d31DaCE6B has 100
The contract is ready
0x418682F9E4F8f2F1eD3aeFBE381b596C97FD8232 donated 5 to fund
0xB0e5CC81Cb6DA4A91C820210803E3B7d31DaCE6B donated 10 to fund
Waiting for the fund to reach the deadline
Deadline reached
Fund did meet its goal
0xECF127ad12e0c8576014FA618fCa3F0CD3890128 has 114.996889461983410464
0x418682F9E4F8f2F1eD3aeFBE381b596C97FD8232 has 94.99972387999852736
0xB0e5CC81Cb6DA4A91C820210803E3B7d31DaCE6B has 89.99980082999893776


Fund goal is set to 20
0xfCdC7f2D5BcB857E94Ab91Dcc3A8Df725C6a5D16 has 100
0x81DD2DA75207b21Da940F324504384f50F1F5C02 has 100
0x7f0d21E68C645ac93392Ce99cc46c589d40047d7 has 100
The contract is ready
0x81DD2DA75207b21Da940F324504384f50F1F5C02 donated 5 to fund
0x7f0d21E68C645ac93392Ce99cc46c589d40047d7 donated 10 to fund
Waiting for the fund to reach the deadline
Deadline reached
Fund did not meet its goal
0x81DD2DA75207b21Da940F324504384f50F1F5C02 got their funds back
0x7f0d21E68C645ac93392Ce99cc46c589d40047d7 got their funds back
0xfCdC7f2D5BcB857E94Ab91Dcc3A8Df725C6a5D16 has 99.996895605983443232
0x81DD2DA75207b21Da940F324504384f50F1F5C02 has 99.99962735499801256
0x7f0d21E68C645ac93392Ce99cc46c589d40047d7 has 99.99969612249837932
```

If you're running in ALGO mode you might notice a ton of warning messages. Don't worry! These are warnings from Reach that can't be turned off and have to be logically protected against (which we do 💪). Explanation on Github

Other warning messages can be turned off by setting REACH_NO_WARN in your reach config file:

```
# Automatically generated with `reach config` at 2022-10-20T18:51:29.9372512
export REACHC_ID=5C4CDE03DE65636B9FC702FB97F96C
export REACH_CONNECTOR_MODE=ETH
export REACH_NO_WARN=1
~
```

Finished Dapp repo: https://github.com/JossDuff/reach-crowdfund/
If this tutorial is helpful at all please leave a star :)

Some ideas that you can implement to improve your Reach and dapp building skills!

- Allows funders to donate multiple times.
- Create more parameters for each fund (name, image, description, link, etc) and display them.
- Allow the receiver to specify an address to send all the funds to if the goal is reached.
- Send to the receiver a list of all addresses that donated to their fund.
- Create a website frontend.
- Allow wallet connection.
- Refactor. This is just one solution, but it’s definitely not the best solution. Improve it or write a better solution!