Tutorial 1: Transactions and contracts

This first tutorial will introduce the most basic concepts of Fae, namely transactions and contracts. The sample code will use the conventions of the postTX utility and the sample command lines use the docker images. After following the instructions there, you should execute this first to have a local server running:

```
./faeServer.sh
```

Hello, world!

Transaction source file: HelloWorld.hs

```
body :: FaeTX String
body = return "Hello, world!"
```

Transaction message: HelloWorld

```
body = HelloWorld
```

Command line

```
./postTX.sh HelloWorld
```

This transaction does only one thing: declare its "results" to be the string `Hello, world!`. This result is saved within the Fae storage in association with the transaction's unique identifier (the hash of this source file plus some identity data), which can be examined by anyone running a Fae client by linking an executable to this storage and reading its contents. `faeServer` and `postTX` together form such a client, and will print an informational message containing this result and other data.

Transaction results

```
Transaction e94750fde381b3fb523b173eb2341e778a693d88947ecb988de1dd1fe425aeea
  result: "Hello, world!"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
```

The lengthy hex strings are not reproducible, but the format is. The return value, "`Hello, world!`", is displayed as the `result`, while the list of transaction signers (a topic for the next tutorial) is given below. The transaction ID, here, is the hex string at the top, `e947...`.

TRANSACTION TYPE

The transaction must declare its type, which is of the form `argType1 -> argType2 -> .. -> FaeTX resultType`. Here, there are no arguments; later, we will see how to supply them to a transaction. The result type is `String` because the `return` value is a string. The transaction body itself is always a function called `body`.

The adjective `FaeTX` modifying `String` is an environment providing the Fae API for a transaction, none of which is actually used in this transaction. (Formally, this environment is a *monad*, which is an intimidating word with an intimidating reputation and which is often poorly explained. A monad is what is described here: an execution environment providing an API.)

Creating and calling a contract

Transaction source file: HelloWorld2.hs

```
body :: FaeTX ()
body = newContract c where
  c :: Contract () String
  -- The underscore here means "there is an argument, which is ignored"
  c _ = spend "Hello, world!"
```

Transaction message: HelloWorld2

```
body = HelloWorld2
```

Command line

```
./postTX.sh HelloWorld2
```

Transaction results

```
Transaction 106d05538dfe7b25c0c91e43abb26a3a4697e6f018a934dd675f769e1af8e68f
  result: ()
  outputs:
    0: 106d05538dfe7b25c0c91e43abb26a3a4697e6f018a934dd675f769e1af8e68f
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
```

This transaction also does only one thing, which is entirely useless in isolation: it creates a new contract that, when called with the empty argument, "spends" the value `Hello, world!`; this "output contract" is shown under `outputs`, numbered `0` (if there were a second contract created, that would be `1`, and so on) and decorated with yet another hex string representing the "version" of this contract after creation (versions are a topic for a later tutorial).

The contract is created but not actually called, nor does the `newContract` function return any value itself. We must therefore provide a second transaction:

Transaction source file: CallHelloWorld2.hs

```
body :: String -> FaeTX String
body = return
```

Transaction message: CallHelloWorld2

```
body = CallHelloWorld2
inputs
  $txID/Body/0/Current = ()
```

Command line

```
./postTX.sh -e txID=<transaction ID from HelloWorld2> CallHelloWorld2
```

Transaction results

```
Transaction cd9b7b039dc0add991e25fd105360d5e090d26212234b408c655cd7e9749ea2d
  result: "Hello, world!"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: 106d.../Body/0/Current (Deleted)
```

This transaction *calls* the contract denoted by its ID with an argument given as a string literal. The return value of that transaction is then provided to `body` as an argument, which it passes through to `return` (pass-through is denoted by assigning the `return` function directly to `body`, not even writing an argument), displaying it as the `result`, as previously.

The effects of the call are summarized in the `input #0` line; the number here indicates the order in the list of `inputs` in the transaction message. After it is the contract ID, described below and with the hex string to which `$txID` expands shortened. Note that, although the transaction ID will look different when you run these transactions, the one displayed after `Transaction` for `HelloWorld2` will still match the summarized one in the contract ID in `CallHelloWorld2`. Finally, this contract is annotated as being `Deleted`, which agrees with the specification of `spend` given later.

THE `()` TYPE

The type `()` appears both in the new contract as an argument and the first transaction as a return value. This type has only one value, which is also called `()`, and is used to denote a "value" that is actually absent. Conceptually, the new contract takes no arguments, but it is required to take an argument, so we give it `()`. Likewise, the `newContract` function returns no value, and therefore the transaction `body` also returns no value, so we call the return type `()`.

CONTRACT CREATION

A new contract is created with the first of the two API functions seen here: the `newContract` function, which takes one argument: here, a function whose type is of the form `Contract argType valType` that forms the body of the contract. When called, the contract takes an argument of type `argType` and "spends" a value of type `valType`. In the definition of `c`, we elide the (unused, empty) argument with a `_` argument and return a `String`.

CONTRACT TYPE

The type `Contract argType valType` is (nearly) a synonym for the function type `argType -> Fae argType valType valType`, where the three-part "adjective" type `Fae argType valType` is an environment (monad) for a contract taking an `argType` and returning a `valType`. It differs from `FaeTX` in two ways: first, by specifying the argument and return types; and second, by possessing a slightly larger set of API functions. The function type therefore denotes a function taking an `argType` and returning a `valType` in the `Fae argType valType` monad; although the parameters to `Fae` seem redundant here, the `Fae` monad can contain any type, not just `valType`, and not all API functions deal with the `argType`, while some of them need to know the actual argument and return types declared for the contract, so the parameters are present.

SPENDING VALUES

The other API function we see here is `spend`, which is much like `return`, and in fact in this context is no different, but in general deals with value-bearing return types appropriately. We will see what this means in a later tutorial; suffice it to say, here, that a contract *must* end with `spend`. This API function is one of two that are only available in `Fae` but not in `FaeTX`; it can only be used by contract code and not in a transaction. After a contract spends a value, it is deleted from Fae storage and cannot be called again.

INPUT LIST

The second transaction employs an *input list* to call the contract created in the first one. Contract calls are denoted by a *contract ID* and an *argument*. The contract ID in this case is `$txID/Body/0/Current` meaning the first (indexed-0) contract created by the body of the transaction with ID `$txID`. The `Body` and `Current` components have variants: `Body` can also be `InputCall n`, which identifies the n'th contract called by transaction `$txID`; `Current` can also be `Version <32-byte hex string>`, which identifies a particular incarnation of the contract and will be discussed in a later tutorial.

The `newContract` function does *not* return the contract ID because it is impossible to use it meaningfully within Fae code; the only way a contract ID can be used is in the list of transaction inputs, which is part of the transaction message, not the source code.

INPUT ARGUMENTS

The input contract is called with a string-literal argument `()`. All arguments to input contracts are string literals but are parsed behind the scenes into the expected argument type for the contract as it was created. This scheme is the first instance of Fae's contract security guarantees: it ensures that the caller of a contract cannot provide it with a computationally expensive argument, thus performing a denial-of-service attack on that contract. The parser is declared along with the contract's argument type (the type `()` has its parser declared in the language definition) and, therefore, is completely under the control of the contract's *author*, who can design it to handle input strings in a safe way or at least know to expect the possible negative consequences of parsing.

Contract names

In the previous example, the contract was created by passing the contract function directly to `newContract`, which was convenient for dashing off a simple contract; this is, however, a limited technique that is not viable for any contracts that need to hold valuables (a subject that is the topic of Tutorial 2). In addition, it promotes verbose function definitions that must be decorated with lengthy `where` clauses or other inline function definition syntax (arguably, it is actually more verbose than this style, but the verbosity is decoupled from other code, so is cleaner). Fae provides a more organized, more modular, and more descriptive method: *contract names*. We can rewrite `HelloWorld2.hs` thus:

Transaction source file: HelloWorld2a.hs

```
body :: FaeTX ()
body = newContract (Echo "Hello, world!")

data Echo = Echo String deriving (Generic)

instance ContractName Echo where
  type ArgType Echo = ()
  type ValType Echo = String
  theContract (Echo s) = \_ -> spend s
```

Transaction message: HelloWorld2a

```
body = HelloWorld2a
```

Command line

```
./postTX.sh HelloWorld2a
```

The transaction results are exactly the same as for HelloWorld2.

DATA DEFINITIONS

The contract name is a data type that we define, appropriately, with the keyword `data`. It is called `Echo`, and its values are *also* called `Echo`, except they also contain the data, which is (according to the definition) a `String`. You can see this in the `newContract` call, where the argument is `Echo "Hello, world!"`. The `Echo` here is a *constructor* for this type, and is effectively a function taking the data fields as arguments.

DERIVING GENERIC

The phrase `deriving (Generic)` at the end of the `Echo` definition is one of the only pieces of boilerplate that Fae requires. `Generic` is a property that allows introspection into data types, and is used widely in Fae's internals, which reduce every other required property to just this one, so that the boilerplate is only two words on each data definition. Every contract name should be `Generic`.

INSTANCES

In order to use `Echo` as a contract name, it must be an *instance* of the interface `ContractName` (formally called a *typeclass*, as in a class (i.e. collection) of types, namely the ones with the designated interface), which is defined by Fae to have three members: the two types `ArgType` and `ValType`, and the method `theContract`.

The type members, when declared, use those words as “adjectives” (like `FaeTX`, the environment for transactions) applied to, in this case, `Echo`; you can see the pattern that would hold for other contract names. That is, `ArgType` and `ValType` are not types but *type families*, and we are instantiating them for `Echo`. Since they are declared with `type` rather than `data`, they are not given constructors but simply set equal to other, existing types, in this case `()` and `String`.

The method `theContract` is the link between the data type `Echo` and its intended meaning as a contract. Each value `Echo s`, for various strings `s`, is the “name” for a different contract, one that simply returns `s`. The function here is the same one that would have gone in the `where` clause in the previous contract style; it takes an `ArgType Echo` and returns a `ValType Echo`.

PATTERN MATCHING

Our definition of `theContract` uses the constructor to *deconstruct* a value of type `Echo`: we “match” the value with its constructor expression `Echo s`, and now we can use `s`, which is a `String`, in the function's code.

LAMBDAS

We chose the style `theContract (Echo s) = \_ -> spend s` because it emphasizes that *the contract* for an `Echo` is a particular function; the construct `\_ -> <code>` is *lambda syntax* for defining a function. We already saw, previously, that the underscore `_` as an argument means “ignore this”; it plays the same role here, as the arguments to this anonymous function follow the `\`, which represents the letter lambda (λ). It would also have been possible to write `theContract (Echo s) _ = spend s`, using the first argument of the function that was previously the value as an additional argument to the function that returned that value, but this seems (to me) to be a little unclear.

UNNAMED CONTRACTS

To connect this concept back to the old syntax where the contract function was given inline, we observe that by passing a function of type `Contract () String` to `newContract`, we are using that function as a contract name! Indeed, every type `Contract argType valType` is an instance of `ContractName` implicitly, with the obvious `ArgType` and `ValType`, and where the `theContract` for a contract function is, of course, that function. Therefore, the contract, when defined this way, has a name of type `Contract () String`, as opposed to `Echo`. This is less descriptive

and, as we will see in Tutorial 2, also less capable. However, it is useful to remember that this is the case: contracts always have names.

Repeatable contracts

Transaction source file: HelloWorld3.hs

```
body :: FaeTX ()
body = newContract (Echo2 "Hello, world!" "Goodbye!")

data Echo2 = Echo2 String String deriving (Generic)

instance ContractName Echo2 where
  type ArgType Echo2 = ()
  type ValType Echo2 = String
  theContract (Echo2 s1 s2) = \_ -> do
    release s1
    spend s2
```

Transaction message: HelloWorld3

```
body = HelloWorld3
```

Command line

```
./postTX.sh HelloWorld3
```

The transaction results look the same as for HelloWorld2. To call this contract, the following transaction may be run repeatedly.

Transaction source file: CallHelloWorld3.hs

```
body :: String -> FaeTX String
body = return
```

Transaction message: CallHelloWorld3

```
body = CallHelloWorld3
inputs
  $txID/Body/0/Current = ()
```

Command line:

```
./postTX.sh -e txID=<transaction ID from HelloWorld3> CallHelloWorld3
```

The first transaction creates another new contract that does a little more than before; the second transaction calls it. If we do so twice, we get the following results:

Transaction #1 results

```
Transaction 463657a4a9818c193fc232e6fa77cd0c03dbac6c287b2c5b7904c41653f8cf73
  result: "Hello, world!"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: a569.../Body/0/Current (Updated)
    version: a569ffeebcd79a3ebddcb2fcf0c4aaf9b06f47978db6aea12a2b682fc909c234
```

Transaction #2 results

```
Transaction bab93c3f950e83d1bb10c252d69ff3d1c233d3026be68045ac901da5fd4925a8
  result: "Goodbye!"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: a569.../Body/0/Current (Deleted)
```

As expected, the successive calls returned the two messages "`Hello, world!`" and "`Goodbye!`". Since for the first time we are using a contract that does not immediately `spend`, we see the annotation `Updated` in Transaction #1, as well as a new `version` line (which, again, will not be explained here). In Transaction #2, the contract does call `spend`, and is therefore `Deleted` (since it no longer exists, it does not get a new version).

EXTENDED CONTRACT BODIES

The `theContract` here contains more than one line and, therefore, employs a syntactical construct called the *do block*. A do block is simply an expression in an environment, here the `Fae () String` monad; each line is a separate action, which are all executed sequentially.

RELEASING VALUES

In this contract we encounter the `release` API function, which is similar to `spend` but with two important differences: first, it does not *terminate* the contract but merely *suspends* it, i.e. ends the current contract call but allows it to be called again; and second, it returns the argument of the subsequent call. Here, the argument type is `()`, which can be ignored in a do block statement, so we do not reference it. This API function is the other one that is available in `Fae` but not in `FaeTX`.

CALLING A CONTRACT REPEATEDLY

This contract is called twice. The first time, it will return `Hello, world!`, which is therefore the result of Transaction 2. After this call, the contract is updated in storage to begin at the second line, so on the second call, it returns `Goodbye!`, the result of Transaction 3. After *this* call, the contract is "spent" and removed from Fae storage, so a third call will fail with an error that the contract does not exist.

Endless contracts

Transaction source code: HelloWorld4.hs

```
import Control.Monad (forever)

body :: FaeTX ()
body = newContract (EchoForever "Hello, world!")

data EchoForever = EchoForever String deriving (Generic)

instance ContractName EchoForever where
```

```
  type ArgType EchoForever = ()
  type ValType EchoForever = String
  theContract (EchoForever s) = \_ -> forever (release s)
```

Transaction message: HelloWorld4

```
body = HelloWorld4
```

Command line:

```
./postTX.sh HelloWorld4
```

The transaction results look the same as HelloWorld2. To call this new contract, run the following transaction repeatedly:

Transaction source code: CallHelloWorld4.hs

```
body :: String -> FaeTX String
body = return
```

Transaction message: CallHelloWorld4

```
body = CallHelloWorld4
inputs
  $txID/Body/0/Current = ()
```

Command line:

```
./postTX.sh -e txID=<transaction ID from HelloWorld4> CallHelloWorld4
```

The first two calls produce results like:

Transaction #1 results

```
Transaction 1535b638c0ddb0b97c82f730d7d4cdf9b7b89911e606e1b46c9bd572c6d08e61
  result: "Hello, world!"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: 46bf.../Body/0/Current (Updated)
    version: 46bfee243a90afd489c722839decd31b91eb86780f134f235bf0c0493770c616
```

Transaction #2 results

```
Transaction ee91892aa49ca819204694f38a6bc63fc8c152ade35b726c8e335bd75a6acd19
  result: "Hello, world!"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: 46bf.../Body/0/Current (Updated)
    version: 706f5f3ec89d2c32b7ebb9ce639cdd4e158e6b459d672e72829f43c5a1b1834e
```

Each time, the contract is `Updated`; you can see that the version changes, even though the contract ID and the status message are the same, so this does indicate the contract passing each time into a new state.

RUNNING FOREVER

The `forever` function is an infinite loop that performs its argument on each iteration; it has to be imported from the `Control.Monad` module (part of the Haskell standard library `base`). By placing a `release` inside it, we construct a contract that can be called indefinitely, each time returning the same string `Hello, world!`. This contract need not "end with `spend`" because it never ends.

Summary

This first tutorial demonstrated the basic features of Fae:

- Transactions: input and body
- Contracts: creation, spending, suspending, and calling as inputs
- Do blocks and the two Fae monads `FaeTX` and `Fae`.

Although this set of features is enough to employ the *language* of smart contracts, it is not enough to write contracts that have serious content. In the next tutorial, we will explore the mechanism for denoting and handling value in contracts, which allows authors to create and use such essentials as currencies, identity tokens, and abstract deals.