The Simple Haskell Handbook

Build a Continuous Integration Server from Scratch

Marco Sampellegrini

Contents

Part I

Exploring the Domain

We need a rough idea of what we're going to build.

At the heart of any CI server is the *Build*. A Build runs some commands in a sandboxed environment and reports the result.

We specify the commands to run in a *Pipeline*. Pipelines are made of different Jobs or *Steps*. A Step specifies which commands should be run. All *Commands* will be run in Docker containers. So a Step needs to provide an *Image* as well.

This seems like a good place to start! Let's create a new file `src/Core.hs` and model what we said so far.

```haskell
-- src/Core.hs

module Core where

import RIO

data Pipeline
  = Pipeline
      { steps :: [Step]
      }
  deriving (Eq, Show)

data Step
  = Step
      { name :: StepName
      , commands :: [Text]
      , image :: Image
      }
  deriving (Eq, Show)

data Build
  = Build
      { pipeline :: Pipeline
      }
  deriving (Eq, Show)

newtype StepName = StepName Text
  deriving (Eq, Show)

newtype Image = Image Text
  deriving (Eq, Show)
```

When defining newtypes, we'll also define helper functions to unwrap them and get to the inner value.

```haskell
stepNameToText :: StepName -> Text
stepNameToText (StepName step) = step

imageToText :: Image -> Text
imageToText (Image image) = image
```

Ok great. How does the model evolve through time?

Pipelines and Steps are going to stay static. But Builds will transition into different *States* until eventually they'll either *Succeed* or *Fail*.

```haskell
data Build
  = Build
      { pipeline :: Pipeline
      , state :: BuildState
      }

data BuildState
  = BuildReady
  | BuildRunning
  | BuildFinished BuildResult
  deriving (Eq, Show)

data BuildResult
  = BuildSucceeded
  | BuildFailed
  deriving (Eq, Show)
```

We can start playing around with this! Let's create the file `test/Spec.hs` and define some test values.

```haskell
-- test/Spec.hs

module Main where

import RIO
import Core

-- Helper functions
makeStep :: Text -> Text -> [Text] -> Step
makeStep name image commands
  = Step
      { name = StepName name
      , image = Image image
      , commands = commands
      }

makePipeline :: [Step] -> Pipeline
makePipeline steps =
  Pipeline { steps = steps }

-- Test values
testPipeline :: Pipeline
testPipeline = makePipeline
  [ makeStep "First step" "ubuntu" ["date"]
  , makeStep "Second step" "ubuntu" ["uname -r"]
  ]

testBuild :: Build
testBuild = Build
  { pipeline = testPipeline
  , state = BuildReady
  }

main :: IO ()
main = pure ()
```

This is looking good, but there's one thing we can improve right away. We're using plain lists to define `steps` and `commands`. This means an empty list would be a valid value.

```haskell
Pipeline { steps = [] } -- Valid
```

But does it make sense? A Pipeline with no steps isn't really useful. And the same is true for a Step with no commands.

We can leverage the type system to ensure that both lists have *at least* one element.

```haskell
-- src/Core.hs

data Pipeline
  = Pipeline
      { steps :: NonEmpty Step
      }

data Step
  = Step
      { name :: StepName,
        commands :: NonEmpty Text,
        image :: Image
      }
```

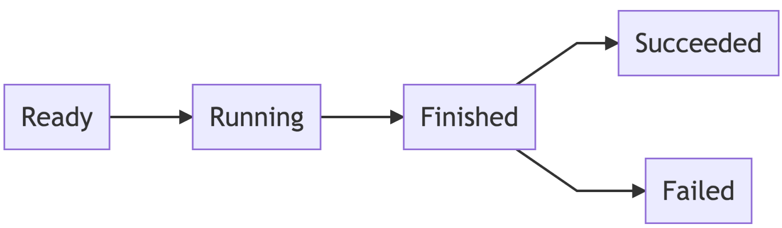
That's much better! We can now use the `NonEmpty.Partial.fromList` function to construct a NonEmpty list. This function will throw an exception if given empty lists, but we know that in our test code we're supplying lists with at least one element so we should be good.

```haskell
-- test/Spec.hs

import qualified RIO.NonEmpty.Partial as NonEmpty.Partial

makeStep :: Text -> Text -> [Text] -> Step
makeStep name image commands
  = Step
      { name: StepName name
       , image: Image image
       , commands: NonEmpty.Partial.fromList commands
       }

makePipeline :: [Step] -> Pipeline
makePipeline steps =
  Pipeline { steps = NonEmpty.Partial.fromList steps }

testPipeline :: Pipeline
testPipeline = makePipeline
  [ makeStep "First step" "ubuntu" ["date"]
  , makeStep "Second step" "ubuntu" ["uname -r"]
  ]
```

All about the State Machine

Let's start thinking about *state transitions*. What needs to happen for a Build to transition to the Running state? How do we know that a Build can be considered Finished?

State machines are the perfect tool to come up with a satisfying Domain model. In Haskell, we can think of a State machine as a function that goes from *OldState* to *NewState*.



In our case, we could introduce a `progress` function that, given a `Build`, determines what needs to happen next. This function will perform some side effects and return an updated version of the build. We can then call it recursively, feeding the output back as the input, until we get to the final state.

```haskell
-- src/Core.hs

progress :: Build -> IO Build
progress build =
  case build.state of
    BuildReady -> undefined -- TODO
    BuildRunning -> undefined -- TODO
    BuildFinished _ ->
      pure build
```

The most trivial case is that of a finished build – not much else we can do there.

But what should happen in the other cases? How do we transition from the Ready state to the Running one?

Well, let's think about it.

We'll transition to Running if there are any steps left to execute.
We'll transition to Finished if all steps have run or any step failed.

This immediately uncovers some data we're not capturing in our model. We don't know which steps have run and whether they failed or succeeded!

Recall that all our steps will run in containers. The way we tell if a step ran successfully or not is by looking at the container exit status. So let's define a `ContainerExitCode` type for that. We'll also define a `StepResult` type to indicate whether a step succeeded or failed.

```haskell
data Build
  = Build
      { pipeline :: Pipeline
      , state :: BuildState
      , completedSteps :: Map StepName StepResult
      }

data StepResult
  = StepFailed ContainerExitCode
  | StepSucceeded
  deriving (Eq, Show)

newtype ContainerExitCode = ContainerExitCode Int
  deriving (Eq, Show)

exitCodeToInt :: ContainerExitCode -> Int
exitCodeToInt (ContainerExitCode code) = code

-- Add `Ord` so that it can be used as Map key
newtype StepName = StepName Text
  deriving (Eq, Show, Ord)
```

Given a `ContainerExitCode` value, we should be able to turn it into a `StepResult`.

```haskell
exitCodeToStepResult :: ContainerExitCode -> StepResult
exitCodeToStepResult exit =
  if exitCodeToInt exit == 0
    then StepSucceeded
    else StepFailed exit
```

Whenever we run a Step, we'll add it to the `completedSteps` Map, considering it successful only when the `exitCode` is `0`.

Let's update the test and initialize the build with an empty Map.

```haskell
-- test/Spec.hs

testBuild :: Build
testBuild = Build
  { pipeline = testPipeline
  , state = BuildReady
  , completedSteps = mempty
  }
```

We can now write a function that determines what's next for a Build. Is there a step to run next? Has the build failed (meaning the current step failed)? Has the build succeeded (all steps completed successfully)?

```haskell
buildHasNextStep :: Build -> Either BuildResult Step
buildHasNextStep build = undefined
```

Note how the return type encodes these invariants – we *either* have a build result *or* there must be a step that we should execute next. Feel free to take a crack at implementing this! We'll write the implementation in the coming pages.

With this function available, we can have a look at the `BuildReady` case in `progress`.

```haskell
progress :: Build -> IO Build
progress build =
  case build.state of
    BuildReady ->
      case buildHasNextStep build of
        Left result ->
          pure $ build{state = BuildFinished result}
        Right step ->
          pure $ build{state = BuildRunning}

    BuildRunning -> -- TODO
    BuildFinished _ -> -- [...]
```

Nice! So if there are no more steps available, the build can transition to the Finished state. Otherwise it'll be Running.

Well… running what?

This is certainly an oversight in our original design. But that's the beauty of uncovering the model through types! We probably want to carry extra data in the Running state – at the very least the name of the step that's currently running.

```
data BuildState
  = BuildReady
  | BuildRunning BuildRunningState
  | BuildFinished BuildResult

data BuildRunningState
  = BuildRunningState
      { step :: StepName
      }
  deriving (Eq, Show)
```

Now it's clear from our model that for a Build to be running, a Step needs to be running.

Once again, we'll implement a piece of the `progress` function with what we just discovered. We also need to add a `state` argument to the `BuildRunning` constructor in the pattern match.

```
progress :: Build -> IO Build
progress build =
  case build.state of
    BuildReady ->
      case buildHasNextStep build of
        Left result ->
          pure $ build{state = BuildFinished result}
        Right step -> do
          let s = BuildRunningState { step = step.name }
          pure $ build{state = BuildRunning s}


    BuildRunning state ->
      -- TODO

    BuildFinished _ -> -- [...]
```

Before transitioning to the Running state, we should perform some side-effects. This is where we create a Docker container and set it up so that it runs the commands specified in the Step.

We'll get to Docker in the next chapter. For now, we will assume that a step in the Running state will succeed automatically. This is so we can move on with our modelling and see if there is anything else we are missing.

So once in the Running state, let's go ahead and move the step to the `completedSteps` Map.

```haskell
import qualified RIO.Map as Map

progress :: Build -> IO Build
progress build =
  case build.state of
    BuildReady ->
      -- [...]

    BuildRunning state -> do
      -- We'll assume the container exited with a 0 status code.
      let exit = ContainerExitCode 0
          result = exitCodeToStepResult exit

      pure build
        { state = BuildReady
        , completedSteps
            = Map.insert state.step result build.completedSteps
        }

    BuildFinished _ ->
      -- [...]
```

As we said, we're hardcoding the container `exitCode` (it will be grabbed from the Docker api later on). We're also transitioning the build back to the Ready state. This will ensure that the logic we implemented a while ago (`buildHasNextStep`) will run next.

It seems like our model is holding up so far. We should go on and implement the `buildHasNextStep` function we left behind.

```haskell
import qualified RIO.List as List

buildHasNextStep :: Build -> Either BuildResult Step
buildHasNextStep build =
  if allSucceeded
    then case nextStep of
      Just step -> Right step
      Nothing -> Left BuildSucceeded
    else Left BuildFailed
  where
    allSucceeded = List.all ((==) StepSucceeded) build.completedSteps
    nextStep = List.find f build.pipeline.steps
    f step = not $ Map.member step.name build.completedSteps
```

Brief explanation: if any steps failed, then we can consider the build to have failed as well. Otherwise we go through the steps in the pipeline and find one which hasn't run yet (not in the `completedSteps` Map). If we can't find a step, then they all succeeded so the build is successful.

Talking to Docker

Each Build step will run in its own Docker container. We can use the Docker api (via HTTP) to create and start containers programmatically.

We'll implement this in a separate module. Let's create a new file `src/Docker.hs` and add a `createContainer` function.

```haskell
-- src/Docker.hs

module Docker where

import RIO

data CreateContainerOptions
  = CreateContainerOptions
      { image :: Image
      }

createContainer :: CreateContainerOptions -> IO ()
createContainer options = undefined
```

The `Image` type is currently defined in the `Core` module. We should move it (along with `ContainerExitCode`) to the `Docker` module as it seems more appropriate there. We'll get a few errors in `Core` – we just need to update `Image` to `Docker.Image`.

```haskell
-- src/Docker.hs

-- These types and functions were previously defined in `Core.hs`.
Delete them from there and move them here.

newtype Image = Image Text
  deriving (Eq, Show)

newtype ContainerExitCode = ContainerExitCode Int
  deriving (Eq, Show)

exitCodeToInt :: ContainerExitCode -> Int
exitCodeToInt (ContainerExitCode code) = code

imageToText :: Image -> Text
imageToText (Image image) = image
```

```haskell
-- src/Core.hs

import qualified Docker

data Step
  = Step
      { name :: StepName,
      , commands :: [Text],
      , image :: Docker.Image
      }

data StepResult
  = StepFailed Docker.ContainerExitCode
  | StepSucceeded
  deriving (Eq, Show)

exitCodeToStepResult :: Docker.ContainerExitCode -> StepResult
exitCodeToStepResult exit =
  if Docker.exitCodeToInt exit == 0
    then StepSucceeded
    else StepFailed exit
```

```haskell
progress :: Build -> IO Build
progress build =
  case build.state of
    -- [...]

    BuildRunning state -> do
      let exit = Docker.ContainerExitCode 0
          result = exitCodeToStepResult exit

      -- [...]
```

Let's also make sure to update the tests.

```haskell
-- test/Spec.hs

import qualified Docker

makeStep name image commands
  = Step
      { name = StepName name
      , commands = NonEmpty.Partial.fromList commands
      , image = Docker.Image image
      }
```

Great! So now we'll need to talk to the Docker Engine API (we'll use version `1.40` if you want to have a look at the docs). To send HTTP requests we'll use the `http-conduit` package. The endpoint we're interested in is `/containers/create`.

```haskell
-- src/Docker.hs

import qualified Network.HTTP.Simple as HTTP

createContainer :: CreateContainerOptions -> IO ()
createContainer options = do
  let body = () -- TODO figure out actual request body
  let req = HTTP.defaultRequest
          & HTTP.setRequestPath "/v1.40/containers/create"
          & HTTP.setRequestMethod "POST"
          & HTTP.setRequestBodyJSON body

  res <- HTTP.httpBS req

  -- Dump the response to stdout to check what we're getting back.
  traceShowIO res
```

Let's open a repl so we can play around with this a bit. The code is clearly incomplete so this request won't have any effect yet. Let's use the `ubuntu` image and see what happens.

```
$ stack repl
Ok, five modules loaded.

> createContainer $ CreateContainerOptions (Image "ubuntu")
*** Exception: HttpExceptionRequest Request {
  host                 = "localhost"
  port                 = 80
  secure               = False
  requestHeaders       = [("Content-Type","application/json;
charset=utf-8")]
  path                 = "/v1.40/containers/create"
  queryString          = ""
  method               = "POST"
  proxy                = Nothing
  rawBody              = False
  redirectCount        = 10
  responseTimeout      = ResponseTimeoutDefault
  requestVersion       = HTTP/1.1
}
 (ConnectionFailure Network.Socket.connect: <socket: 12>:
 does not exist (Connection refused))
```

As we expected, the request failed. First thing we'll need to define is the host to connect to. The Docker Engine API is a bit peculiar in that it doesn't expose a traditional HTTP api. Instead, we have to connect to the Docker *socket*.

So before we can send any requests, we need to configure the HTTP library to talk over sockets. All requests go through a `Manager` that is responsible for managing and sending the actual requests. We'll provide a new `Manager` that knows how to communicate via sockets.

Setting up the socket is fairly low level and honestly not very interesting. This is the only piece of code we won't go over – we'll just stick it in a new file `src/Socket.hs` and forget about it.

```haskell
-- src/Socket.hs

module Socket where

import qualified Network.HTTP.Client as Client
import qualified Network.HTTP.Client.Internal as Client.Internal
import qualified Network.Socket as S
import qualified Network.Socket.ByteString as SBS
import RIO

newManager :: FilePath -> IO Client.Manager
newManager fp =
  Client.newManager $
    Client.defaultManagerSettings
      { Client.managerRawConnection = pure makeSocket
      }
  where
    makeSocket _ _ _ = do
      s <- S.socket S.AF_UNIX S.Stream S.defaultProtocol
      S.connect s (S.SockAddrUnix fp)
      Client.Internal.makeConnection
        (SBS.recv s 8096)
        (SBS.sendAll s)
        (S.close s)
```

Don't sweat about understanding the code above, just assume it works. Now that we have a way of creating a request `Manager` that can talk to a socket, let's use it in our function. We'll also use the `aeson` package to create a `null` JSON value to send as the request body. This is just to check it works, we'll update the body in the next section.

```haskell
-- src/Docker.hs

import qualified Data.Aeson as Aeson
import qualified Socket

createContainer :: CreateContainerOptions -> IO ()
createContainer options = do
  manager <- Socket.newManager "/var/run/docker.sock"

  let body = Aeson.Null -- TODO

  let req = HTTP.defaultRequest
          & HTTP.setRequestManager manager
          & HTTP.setRequestPath "/v1.40/containers/create"
          & HTTP.setRequestMethod "POST"
          & HTTP.setRequestBodyJSON body

  res <- HTTP.httpBS req
  traceShowIO res
```

We should be able to send a request now. Before going any further, make sure the Docker daemon is running on your machine!

Let's reload the repl (you can use the `:r` command) and call `createContainer` once again.

```
> :r
Compiling ...

> createContainer $ CreateContainerOptions (Image "ubuntu")
Response {
  responseStatus = Status {
    statusCode = 400, statusMessage = "Bad request"
  },
  responseVersion = HTTP/1.1,
  responseHeaders = [
    ("Api-Version","1.40"),
    ("Content-Type","application/json"),
    ("Docker-Experimental","false"),
    ....
  ],
  responseBody =
    "{\"message\":\"Config cannot be empty in order to create a
container\"}\n"
}
```

With any luck, we're going to get a response back from the Docker api! Note how it's complaining about the body of the request not matching what it expected (we're sending `null`). But at least we can talk to it now!

Building JSON values

Let's get to the request body next. We need to send some JSON data, which we can build up with the `aeson` package. At the very least, we must send an object with an `Image` field, ie. `{ "Image": "ubuntu" }`

```
createContainer :: CreateContainerOptions -> IO ()
createContainer options = do
  let image = imageToText options.image
  let body = Aeson.object
               [ ("Image", Aeson.toJSON image)
               ]
```

Make sure the `ubuntu` image is available locally otherwise launching the container won't work. This is because our code doesn't take care of pulling images just yet. Just to be sure, run the following:

```
$ docker pull ubuntu
```

If we now run `createContainer` in the repl once again, we should see the container starting!

```
> createContainer $ CreateContainerOptions (Image "ubuntu")
Response {
  responseStatus = Status {
    statusCode = 201, statusMessage = "Created"
  },

  ...
}
```

And we can verify that a container has been created (but not started).

```
$ docker ps -a

CONTAINER ID    IMAGE    COMMAND
e8d5c24427d8    ubuntu   "/bin/bash"
```

To make our container a bit more interesting, let's use `echo "hello"` as the command. We'll also label it with the `quad` tag so it's easier to clean up all of our tests later on.

```
createContainer :: CreateContainerOptions -> IO ()
createContainer options = do
  let image = imageToText options.image
  let body = Aeson.object
               [ ("Image", Aeson.toJSON image)
               , ("Tty", Aeson.toJSON True)
               , ("Labels", Aeson.object [("quad", "")])
               , ("Cmd", "echo hello")
               , ("Entrypoint", Aeson.toJSON [Aeson.String "/bin/sh",
"-c"])
               ]
```

Let's call `createContainer` once again and manually grab the container ID from the `responseBody` (`968f`).

```
> createContainer $ CreateContainerOptions (Image "ubuntu")
Response {
  responseStatus = Status {
    statusCode = 201, statusMessage = "Created"
  },
  responseVersion = HTTP/1.1,
  responseHeaders = [
    ("Api-Version","1.40"),
    ("Content-Type","application/json"),
    ("Docker-Experimental","false"),
    ....
  ],
  responseBody = "{\"Id\":\"968f...\",\"Warnings\":[]}\n"
}
```

We can grab the container id and start the container manually to see if it's actually working.

```
$ docker start 968f
968f

$ docker logs 968f
hello
```

Yes it's alive!

Parsing JSON

As we saw from playing around with the Docker CLI, in order to start a container we first need to grab its id. So it would be helpful if our `createContainer` function returned the id of the freshly created container.

First, let's introduce a new type `ContainerId` and change the function return type.

```haskell
-- src/Docker.hs

newtype ContainerId = ContainerId Text
  deriving (Eq, Show)

containerIdToText :: ContainerId -> Text
containerIdToText (ContainerId c) = c


-- Return type was IO ()
createContainer :: CreateContainerOptions -> IO ContainerId
createContainer options = do
  -- [...]
```

For debugging purposes, we've been tracing the response we got back from the api. Now that we know it's working, we can get rid of the `traceShowIO` call and create a function `parser` of type `Aeson.Value -> Aeson.Parser ContainerId` that can decode the response into a `ContainerId` value.

Our `parser` is simply looking for a field `Id` in the response.

```haskell
import Data.Aeson ((.:))

createContainer :: CreateContainerOptions -> IO ContainerId
createContainer options = do
  -- [...]

  let parser = Aeson.withObject "create-container" $ \o -> do
        cId <- o .: "Id"
        pure $ ContainerId cId

  res <- HTTP.httpBS req
  parseResponse res parser
```

We're also going to define a `parseResponse` function that tries to decode an HTTP Response with the given parser, or fails by throwing an exception.

```haskell
import qualified Data.Aeson.Types as Aeson.Types

parseResponse
  :: HTTP.Response ByteString
  -> (Aeson.Value -> Aeson.Types.Parser a)
  -> IO a
parseResponse res parser = do
  let result = do
        value <- Aeson.eitherDecodeStrict (HTTP.getResponseBody res)
        Aeson.Types.parseEither parser value

  case result of
    Left e -> throwString e
    Right status -> pure status
```

This is mostly boilerplate, but we're done with it now. Note how we throw an exception in case we can't parse the response. Not much else we can do in case of error.

Testing the `createContainer` function now gives back a `ContainerId` value. Let's try it out in the repl.

```
> createContainer $ CreateContainerOptions (Image "ubuntu")
ContainerId "ac186..."
```

Which means we can get to the `startContainer` function!

Starting a Container

The request setup looks very similar. We don't send any data this time and we discard the response body. Note how we must use `encodeUtf8` (which is provided by `RIO`) to go from `Text` to `ByteString`.

```
startContainer :: ContainerId -> IO ()
startContainer container = do
  manager <- Socket.newManager "/var/run/docker.sock"

  let path
        = "/v1.40/containers/" <> containerIdToText container <>
"/start"

  let req = HTTP.defaultRequest
          & HTTP.setRequestManager manager
          & HTTP.setRequestPath (encodeUtf8 path)
          & HTTP.setRequestMethod "POST"

  void $ HTTP.httpBS req
```

So now for the final (manual) test. We first create the container.

```
> container <- createContainer $ CreateContainerOptions (Image
"ubuntu")
> container
ContainerId "cf5e7..."
```

Then start it.

```
> startContainer container
```

And finally check with `docker logs` whether it ran successfully.

```
$ docker logs cf5e7
hello
```

Yes it worked! We've been doing manual testing while working on this feature. Once we have some more infrastructure in place we can start writing proper integration tests.

Services and Dependency injection

So far, the `Docker` module contains two functions: `createContainer` and `startContainer`. There are a few issues with this module that I'd like to highlight.

For starters, there's a lot of code repetition. We're initializing the HTTP Manager multiple times, whereas we could share it among all requests.

But the real issue is that this module is (ironically) not very *modular*. It would be very hard for us to use it as a *dependency*. Ideally, we would have a type that describes which *operations* the Docker module supports, so that clients can depend on it.

Let's try to use it and see if it makes sense. We'll go back to the `progress` function in `Core` and add some code to create and start a container. As a reminder, this is where we transition the Build state (`Ready -> Running -> Finished`).

```haskell
-- src/Core.hs

progress :: Build -> IO Build
progress build =
  case build.state of
    BuildReady ->
      case buildHasNextStep build of
        Left result ->
          pure $ build{state = BuildFinished result}
        Right step -> do
          let options = Docker.CreateContainerOptions step.image
          container <- Docker.createContainer options
          Docker.startContainer container

          let s = BuildRunningState { step = step.name }
          pure $ build{state = BuildRunning s}

    BuildRunning state -> ...
    BuildFinished _ -> ...
```

To recap, when the build is in the Ready state, we need to create and start the container so that we can then transition the build to Running.

Is there anything wrong with this code? Do we *always* want to call the Docker api?

As it is, this function would be quite hard to test. There would be no way for us to provide *a different implementation* for the Docker module. That's usually an indicator we're not using the right abstraction.

What would an ideal solution look like then?

Well, it's easy really. What we need is some form of *Dependency injection*. The great thing about Haskell is that we don't need any framework to achieve that, it's just *function application*! We provide dependencies as function arguments so that we're not tied to a specific implementation.

Let's see how. We're going to add another parameter `Docker.Service` (the type doesn't exist yet) to the `progress` function.

```
progress :: Docker.Service -> Build -> IO Build
progress docker build =
  -- [...]
```

Now we can change the body to use `docker` (the injected service) in place of the actual module `Docker`.

```
progress :: Docker.Service -> Build -> IO Build
progress docker build =
  case build.state of
    -- [...]

    let options = Docker.CreateContainerOptions step.image
    container <- docker.createContainer options
    docker.startContainer container
```

This solves the problem because we're not depending on a specific implementation anymore.

What is this `Service` type then? Going back to the Docker module, we can define it as such:

```
-- src/Docker.hs

data Service
  = Service
      { createContainer :: CreateContainerOptions -> IO ContainerId,
        startContainer :: ContainerId -> IO ()
      }
```

It's just a simple record type!

Whenever we want to talk to the Docker api, we'll pass around a `Docker.Service` value which contains a bunch of functions for the job.

This is now very easy to test, because we can swap out the real `Docker.Service` implementation for another one. We won't dive into unit testing for this project, but if we wanted to test `Core.progress` in isolation, we'd be able to do so by providing a mocked `Docker.Service` value.

Initializing the Service is just a matter of using the functions we already have. We need to rename the existing functions to avoid name clashes with the record fields we just defined. That's an easy fix, we can just append an underscore `_` at the end.

So let's add a `createService` function with the real implementation and update the existing function names.

```haskell
-- src/Docker.hs

createService :: IO Service
createService = do
  pure Service
    { createContainer = createContainer_
    , startContainer = startContainer_
    }

-- update name
createContainer_ :: CreateContainerOptions -> IO ContainerId
createContainer_ options = do
  -- [...]

-- update name
startContainer_ :: ContainerId -> IO ()
startContainer_ container = do
  -- [...]
```

With this approach, we have removed the hard dependency on the *implementation* of `createContainer` and `startContainer` in the `Core` module. Instead, we now have a nice *interface* we can work with.

Setting up tests

This is a good time to write some basic integration tests. Let's set up `hspec` in our `Spec` module and get the ball rolling.

```haskell
-- test/Spec.hs

import Test.Hspec

main :: IO ()
main = hspec do
  describe "Quad CI" do
    it "should run a build (success)" do
      1 `shouldBe` 1 -- TODO
```

Let's also define a helper function `runBuild` which will call `Core.progress` recursively until the build finishes. We can then run some checks on its final state.

```haskell
-- test/Spec.hs

runBuild :: Docker.Service -> Build -> IO Build
runBuild docker build = do
  newBuild <- Core.progress docker build
  case newBuild.state of
    BuildFinished _ ->
      pure newBuild
    _ -> do
      threadDelay (1 * 1000 * 1000)
      runBuild docker newBuild
```

We call `Core.progress` over and over until we get the build in the Finished state. Note how we wait one second in between calls to `Core.progress` – this is so we don't spam the Docker api too much. With this, we can write our first test!

```
import qualified RIO.Map as Map

-- Existing test values

testPipeline :: Pipeline
testPipeline = makePipeline
  [ makeStep "First step" "ubuntu" ["date"]
  , makeStep "Second step" "ubuntu" ["uname -r"]
  ]

testBuild :: Build
testBuild = Build
  { pipeline = testPipeline
  , state = BuildReady
  , completedSteps = mempty
  }


-- First test

testRunSuccess :: Docker.Service -> IO ()
testRunSuccess docker = do
  result <- runBuild docker testBuild
  result.state `shouldBe` BuildFinished BuildSucceeded
  Map.elems result.completedSteps `shouldBe` [StepSucceeded,
StepSucceeded]
```

In the first assertion, we expect the build to finish successfully. In the second one we're turning `completedSteps` (which has type `Map StepName StepResult`) into `[StepResult]` and checking that both steps did indeed succeed.

We need to pass a `Docker.Service` value to `testRunSuccess` – let's initialize it once at the top so that all tests can make use of it.

```haskell
-- test/Spec.hs

main :: IO ()
main = hspec do
  docker <- runIO Docker.createService
  describe "Quad CI" do
    it "should run a build (success)" do
      testRunSuccess docker
```

Once again, the tests rely on the `ubuntu` image to be available locally. This is something we're going to be automating shortly.

Ok let's see if the tests pass.

```
$ stack test

quad> test (suite: quad-test)

Progress 1/2: quad
Quad CI
  should run a build (success)

  Finished in 2.7079 seconds
  1 example, 0 failures
```

Way to go!

We can't yet assert that the container is doing the right thing (ie. inspect the logs), but this is a good start.

Everytime we run the tests, it will create and start two containers. As we add more tests, this will generate quite a bit of cruft. Thankfully, all of our containers have a `quad` label, so they're easily cleaned up.

We can add a `beforeAll` hook to our test suite so that existing containers are cleaned up on every test run. There's no need to deal with the Docker API for this, we can just invoke the Docker CLI. Let's use the `typed-process` package to do that.

```haskell
import qualified System.Process.Typed as Process

main :: IO ()
main = hspec do
  docker <- runIO Docker.createService
  beforeAll cleanupDocker $ describe "Quad CI" do
    it "should run a build (success)" do
      testRunSuccess docker

cleanupDocker :: IO ()
cleanupDocker = void do
  Process.readProcessStdout "docker rm -f $(docker ps -aq --filter
\"label=quad\")"
```

If we now run the tests and check which containers are present in our system (with `docker ps -a`), we should only see containers created by the last test run. This is useful for debugging purposes. Containers from previous runs are deleted by the `cleanupDocker` hook.

End of sample

Thanks for checking out the book! I'd love to know what you think.

You can subscribe to the newsletter and I'll send you an email when the book is finished.

https://alpacaaa.net/simple-haskell-book/