Если вы видите что-то необычное, просто сообщите мне.

Testing with Docker

In first three parts of this series, we've combined several useful Haskell libraries to make a small web app. In part 1 we used Persistent to create a schema with automatic migrations for our database. Then in part 2 we used Servant to expose this database as an API through a couple simple queries. Finally in part 3, we used Redis to act as a cache so that repeated requests for the same user happen faster.

For our next step, we'll tackle a thorny issue: testing. How do we test a system that has so many moving parts? There are a couple general approaches we could take.

On one end of the spectrum we could mock out most of our API calls and services. This helps gives our testing deterministic behavior. This is desirable since we would want to tie deployments to test results. But we also want to be faithfully testing our API. So on the other end, there's the approach we'll try in this article. We'll set up functions to run our API and other services, and then use before and after hooks to run them. We'll make our lives easier in the end by using Docker to handle running our outside services. On the Github repository for this series, you can find the code for this series. For this part, you'll mainly want to look at our testing code. This module has the test specification and this file has some utilities.

And don't miss the final part of this series! We'll make our schema more complex and use Esqueleto to perform type safe joins! And for some more cool library ideas, be sure to check out our Production Checklist!

CREATING CLIENT FUNCTIONS FOR OUR API

Calling our API from our tests means we'll want a way to make API calls programmatically. We can do this with amazing ease for a Servant API by using the servant-client library. This library has one primary function: client. This function takes a proxy for our API and generates programmatic client functions. Let's remember our basic endpoint types (after resolving connection information parameters):

```
fetchUsersHandler :: Int64 -> Handler User
createUserHandler :: User -> Handler Int64
```

We'd like to be able to call these API's with functions that use the same parameters. Those types might look something like this:

```
fetchUserClient :: Int64 -> m User
createUserClient :: User -> m Int64
```

Where m is some monad. And in this case, the ServantClient library provides such a monad, ClientM. So let's re-write these type signatures, but leave them seemingly unimplemented:

```
fetchUserClient :: Int64 -> ClientM User
createUserClient :: User -> ClientM Int64
```

Now we'll construct a pattern match that combines these function names with the :<|> operator. As always, we need to make sure we do this in the same order as the original API type. Then we'll set this pattern to be the result of calling that primary client function on a proxy for our API:

```
fetchUserClient :: Int64 -> ClientM User
createUserClient :: User -> ClientM Int64
(fetchUserClient :<|> createUserClient) = client (Proxy :: Proxy UsersAPI)
```

And that's it! The Servant library fills in the details for us and implements these functions! Soon we'll see how we can actually call these functions.

SETTING UP THE TESTS

We'd like to get to the business of deciding on our test cases and writing them. But first we need to make sure that our tests have a proper environment. This means 3 things. First we need to fetch

the connection information for our data stores and API. This means the PGInfo, the RedisInfo, and the ClientEnv we'll use to call the client functions we wrote. Second, we need to actually migrate our database so it has the proper tables. Third, we need to make sure our server is actually running. Let's start with the connection information, as this is easy:

```
setupTests = do
  let pgInfo = localConnString
  let redisInfo = defaultConnectInfo
  ...
```

Now to create our client environment, we'll need two main things. We'll need a manager for the network connections and the base URL for the API. Since we're running the API locally, we'll use a localhost URL. The default manager from the Network library will work fine for us. There's an optional third argument for storing cookies, but we can leave that as Nothing.

```
import Network.HTTP.Client (newManager)
import Network.HTTP.Client.TLS (tlsManagerSettings)
import Servant.Client (ClientEnv(..))

setupTests = do
  pgInfo <- fetchPostgresConnection
  redisInfo <- fetchRedisConnection
  mgr <- newManager tlsManagerSettings
  baseUrl <- parseBaseUrl "http://127.0.0.1:8000"
  let clientEnv = ClientEnv mgr baseUrl Nothing
```

Now we can run our migration, which will ensure that our users table exists:

```
import Schema (migrateAll)

setupTests = do
  let pgInfo = localConnString
  ...
  runStdoutLoggingT $ withPostgresqlConn pgInfo $ \dbConn ->
    runReaderT (runMigrationSilent migrateAll) dbConn
```

Last of all, we'll start our server with runServer from our API module. We'll fork this off to a separate thread, as otherwise it will block the test thread! We'll wait for a second afterward to

make sure it actually loads before the tests run (there are less hacky ways to do this of course). But then we'll return all the important information we need, and we're done with test setup:

```
setupTests :: IO (PGInfo, RedisInfo, ClientEnv, ThreadID)
setupTests = do
  pgInfo <- fetchPostgresConnection
  redisInfo <- fetchRedisConnection
  mgr <- newManager tlsManagerSettings
  baseUrl <- parseBaseUrl "http://127.0.0.1:8000"
  let clientEnv = ClientEnv mgr baseUrl
  runStdoutLoggingT $ withPostgresqlConn pgInfo $ \dbConn ->
    runReaderT (runMigrationSilent migrateAll) dbConn
  threadId <- forkIO runServer
  threadDelay 1000000
  return (pgInfo, redisInfo, clientEnv, serverThreadId)
```

ORGANIZING OUR 3 TEST CASES

Now that we're all set up, we can decide on our test cases. We'll look at 3 of them. First, if we have an empty database and we fetch a user by some arbitrary ID, we'll expect an error. Further, we should expect that the user does not exist in the database or in the cache, even after calling fetch.

In our second test case, we'll look at the effects of calling the create endpoint. We'll save the key we get from this endpoint. Then we'll verify that this user exists in the database, but NOT in the cache. Finally, our third case will insert the user with the create endpoint and then fetch the user. We'll expect at the end of this that in fact the user exists in both the database AND the cache.

We organize each of our tests into up to three parts: the "before hook", the test assertions, and the "after hook". A "before hook" is some IO code we'll run that will return particular results to our test assertion. We want to make sure it's done running BEFORE any test assertions. This way, there's no interleaving of effects between our test output and the API calls. Each before hook will first make the API calls we want. Then they'll investigate our different databases and determine if certain users exist.

We also want our tests to be database-neutral. That is, the database and cache should be in the same state after the test as they were before. So we'll also have "after hooks" that run after our tests have finished (if we've actually created anything). The after hooks will delete any new entries. This means our before hooks also have to pass the keys for any database entities they create. This way the after hooks know what to delete.

Last of course, we actually need the testing code that makes assertions about the results. These will be pretty straightforward as we'll see below.

TEST #1

For our first test, we'll start by making a client call to our API. We use runClientM combined with our clientEnv and the fetchUserClient function. Next, we'll determine that the call in fact returns an error as it should. Then we'll add two more lines checking if there's an entry with the arbitrary ID in our database and our cache. Finally, we return all three boolean values:

```
beforeHook1 :: ClientEnv -> PGInfo -> RedisInfo -> IO (Bool, Bool, Bool)
beforeHook1 clientEnv pgInfo redisInfo = do
  callResult <- runClientM (fetchUserClient 1) clientEnv
  let throwsError = isLeft callResult
  inPG <- isJust <$> fetchUserPG pgInfo 1
  inRedis <- isJust <$> fetchUserRedis redisInfo 1
  return (throwsError, inPG, inRedis)
```

Now we'll write our assertion. Since we're using a before hook returning three booleans, the type of our Spec will be SpecWith (Bool, Bool, Bool). Each it assertion will take this boolean tuple as a parameter, though we'll only use one for each line.

```
spec1 :: SpecWith (Bool, Bool, Bool)
spec1 = describe "After fetching on an empty database" $ do
  it "The fetch call should throw an error" $ \(throwsError, _, _) -> throwsError `shouldBe`
True
  it "There should be no user in Postgres" $ \(_, inPG, _) -> inPG `shouldBe` False
  it "There should be no user in Redis" $ \(_, _, inRedis) -> inRedis `shouldBe` False
```

And that's all we need for the first test! We don't need an after hook since it doesn't add anything to our database.

TESTS 2 AND 3

Now that we're a little more familiar with how this code works, let's look at the next before hook. This time we'll first try creating our user. If this fails for whatever reason, we'll throw an error and stop the tests. Then we can use the key to check out if the user exists in our database and Redis. We return the boolean values and the key.

```
beforeHook2 :: ClientEnv -> PGInfo -> RedisInfo -> IO (Bool, Bool, Int64)
beforeHook2 clientEnv pgInfo redisInfo = do
  userKeyEither <- runClientM (createUserClient testUser) clientEnv
  case userKeyEither of
    Left _ -> error "DB call failed on spec 2!"
    Right userKey -> do
      inPG <- isJust <$> fetchUserPG pgInfo userKey
      inRedis <- isJust <$> fetchUserRedis redisInfo userKey
      return (inPG, inRedis, userKey)
```

Now our spec will look similar. This time we expect to find a user in Postgres, but not in Redis.

```
spec2 :: SpecWith (Bool, Bool, Int64)
spec2 = describe "After creating the user but not fetching" $ do
  it "There should be a user in Postgres" $ \(inPG, _, _) -> inPG `shouldBe` True
  it "There should be no user in Redis" $ \(_, inRedis, _) -> inRedis `shouldBe` False
```

Now we need to add the after hook, which will delete the user from the database and cache. Of course, we expect the user won't exist in the cache, but we include this since we'll need it in the final example:

```
afterHook :: PGInfo -> RedisInfo -> (Bool, Bool, Int64) -> IO ()
afterHook pgInfo redisInfo (_, _, key) = do
  deleteUserCache redisInfo key
  deleteUserPG pgInfo key
```

Last, we'll write one more test case. This will mimic the previous case, except we'll throw in a call to fetch in between. As a result, we expect the user to be in both Postgres and Redis:

```
beforeHook3 :: ClientEnv -> PGInfo -> RedisInfo -> IO (Bool, Bool, Int64)
beforeHook3 clientEnv pgInfo redisInfo = do
  userKeyEither <- runClientM (createUserClient testUser) clientEnv
  case userKeyEither of
    Left _ -> error "DB call failed on spec 3!"
    Right userKey -> do
      _ <- runClientM (fetchUserClient userKey) clientEnv
      inPG <- isJust <$> fetchUserPG pgInfo userKey
      inRedis <- isJust <$> fetchUserRedis redisInfo userKey
      return (inPG, inRedis, userKey)

spec3 :: SpecWith (Bool, Bool, Int64)
spec3 = describe "After creating the user and fetching" $ do
  it "There should be a user in Postgres" $ \(inPG, _, _) -> inPG `shouldBe` True
  it "There should be a user in Redis" $ \(_, inRedis, _) -> inRedis `shouldBe` True
```

And it will use the same after hook as case 2, so we're done!

HOOKING IN AND RUNNING THE TESTS

The last step is to glue all our pieces together with hspec, before, and after. Here's our main function, which also kills the thread running the server once it's done:

```
main :: IO ()
main = do
  (pgInfo, redisInfo, clientEnv, tid) <- setupTests
  hspec $ before (beforeHook1 clientEnv pgInfo redisInfo) spec1
  hspec $ before (beforeHook2 clientEnv pgInfo redisInfo) $ after (afterHook pgInfo redisInfo)
$ spec2
  hspec $ before (beforeHook3 clientEnv pgInfo redisInfo) $ after (afterHook pgInfo redisInfo)
$ spec3
```

```
  killThread tid
  return ()
```

And now our tests should pass!

```
After fetching on an empty database
  The fetch call should throw an error
  There should be no user in Postgres
  There should be no user in Redis

Finished in 0.0410 seconds
3 examples, 0 failures

After creating the user but not fetching
  There should be a user in Postgres
  There should be no user in Redis

Finished in 0.0585 seconds
2 examples, 0 failures

After creating the user and fetching
  There should be a user in Postgres
  There should be a user in Redis

Finished in 0.0813 seconds
2 examples, 0 failures
```

USING DOCKER

So when I say, "the tests pass", they now work on my system, after I start up Postgres and Redis. But if you were to clone the code as is and try to run them, you would get failures. The tests depend on Postgres and Redis, so if you don't have them running, they fail! It is quite annoying to have your tests depend on these outside services. This is the weakness of devising our tests as we have. It increases the on-boarding time for anyone coming into your codebase. The new person has to figure out which things they need to run, install them, and so on.

So how do we fix this? One answer is by using Docker. Docker allows you to create containers that have particular services running within them. This spares you from worrying about the details of setting up the services on your local machine. Even more important, you can deploy a docker image to your remote environments. So develop and production will match your local system. To setup this process, we'll create a description of the services we want running on our Docker container. We do this with a docker-compose file. Here's what ours looks like:

```
version: '2'

services:
  postgres:
    image: postgres:10.12
    container_name: real-world-haskell-postgres
    ports:
      - "5432:5432"

  redis:
    image: redis:5.0
    container_name: real-world-haskell-redis
    ports:
      - "6379:6379"
```

Then, you can start these services for your Docker machines with docker-compose up. Granted, you do have to install and run Docker. But if you have several different services, this is a much easier on-boarding process. Better yet, the "compose" file ensures everyone uses the same versions of these services.

Even with this container running, the tests will still fail! That's because you also need the tests themselves to be running on your Docker cluster. But with Stack, this is easy! We'll add the following flag to our stack.yaml file:

```
docker:
  enable: true
```

Now, whenever you build and test your program, you will do so on Docker. The first time you do this, Docker will need to set everything up on the container. This means it will have to download Stack and ALL the different packages you use. So the first run will take a while. But subsequent

runs will be normal. So after all that finishes, NOW the tests should work!

CONCLUSION

Testing integrated systems is hard. We can try mocking out the behavior of external services. But this can lead to a test representation of our program that isn't faithful to the production system. But using the before and after hooks from Hspec is a great way make sure all your external events happen first. Then you can pass those results to simpler test assertions.

When it comes time to run your system, it helps if you can bring up all your external services with one command! Docker allows you to do this by listing the different services in the docker-compose file. Then, Stack makes it easy to run your program and tests on a docker container, so you can use the services!

Stack is key to all this integration. If you've never used Stack before, you should check out our free mini-course. It will teach you all the basics of organizing a Haskell project using Stack.

Don't miss the final part of this series! We'll use the awesome library Esqueleto to perform type safe joins in SQL!

We've seen a lot of libraries in this series, but it's only the tip of the iceberg of all that Haskell has to offer! Check out our Production Checklist for some more ideas!

Revision #1
Created 2022-03-11 06:22:01 UTC by gasick
Updated 2022-03-11 17:11:17 UTC by gasick