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

# Real World Haskell

A lot of people think day-to-day tasks like running a web app are difficult or impossible in Haskell! But of course this isn't true! In our Real World Haskell series, we'll take you through a whole slew of libraries that allow you to write a web backend. These libraries use Haskell's features to approach things like database queries and API building in unique ways.

- Databases and Persistent
- Building an API with Servant!
- Redis Caching
- Testing with Docker
- Esqueleto and Complex Queries

# Databases and Persistent

Welcome to our Real World Haskell Series! In these tutorials, we'll explore a bunch of different libraries you can use for some important tasks in backend web development. We'll start by looking at how we store our Haskell data in databases. If you're already familiar with this, feel free to move on to part 2, where we'll look at building an API using Servant.

If you want a larger listing of the many different libraries available to you, be sure to download our Production Checklist! It'll tell you about other options for databases, APIs and more!

As a final note, all the code for this series is on Github! For this first part, most of the code lives in the Basic Schema module and the Database module.

# THE PERSISTENT LIBRARY

There are many Haskell libraries that allow you to make a quick SQL call. But Persistent does much more than that. With Persistent, you can link your Haskell types to your database definition. You can also make type-safe queries to save yourself the hassle of decoding data. All in all, it's a very cool system. Let's start our journey by defining the type we'd like to store.

# OUR BASIC TYPE

Consider a simple user type that looks like this:

```
data User = User
  { userName :: Text
  , userEmail :: Text
  , userAge :: Int
  , userOccupation :: Text
  }
```

Imagine we want to store objects of this type in an SQL database. We'll first need to define the table to store our users. We could do this with a manual SQL command or through an editor. But regardless, the process will be at least a little error prone. The command would look something like this:

```
create table users (
  name varchar(100),
  email varchar(100),
  age bigint,
  occupation varchar(100)
)
```

When we do this, there's nothing linking our Haskell data type to the table structure. If we update the Haskell code, we have to remember to update the database. And this means writing another error-prone command.

From our Haskell program, we'll also want to make SQL queries based on the structure of the user. We could write out these raw commands and execute them, but the same issues apply. There would be a high probability of errors. Persistent helps us solve these problems.

# PERSISTENT AND TEMPLATE HASKELL

We can get these bonuses from Persistent without all that much extra code! To do this, we're going to use Template Haskell (TH). There are a few pros and cons of TH. It does allow us to avoid writing some boilerplate code. But it will make our compile times longer as well. It will also make our code less accessible to inexperienced Haskellers. With Persistent however, the amount of code generated is substantial, so the pros out-weigh the cons.

To generate our code, we'll use a language construct called a "quasi-quoter". This is a block of code that follows some syntax designed by the programmer or in a library, rather than normal Haskell syntax. It is often used in libraries that do some sort of foreign function interface. We delimit a quasi-quoter by a combination of brackets and pipes. Here's what the Template Haskell call looks

like. The quasi-quoter is the final argument:

```
import qualified Database.Persist.TH as PTH

PTH.share [PTH.mkPersist PTH.sqlSettings, PTH.mkMigrate "migrateAll"] [PTH.persistLowerCase|

|]
```

The share function takes a list of settings and then the quasi-quoter itself. It then generates the necessary Template Haskell for our data schema. Within this section, we'll define all the different types our database will use. We notate certain settings about those types. In particular we specify sqlSettings, so everything we do here will focus on an SQL database. More importantly, we also create a migration function, migrateAll. After this Template Haskell gets compiled, this function will allow us to migrate our DB. This means it will create all our tables for us!

But before we see this in action, we need to re-define our user type. Instead of defining User in the normal Haskell way, we're going to define it within the quasi-quoter. Note that this level of Template Haskell requires many compiler extensions. Here's our definition:

```
{-# LANGUAGE TemplateHaskell            #-}
{-# LANGUAGE QuasiQuotes                #-}
{-# LANGUAGE TypeFamilies               #-}
{-# LANGUAGE MultiParamTypeClasses      #-}
{-# LANGUAGE GADTs                      #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE RecordWildCards            #-}
{-# LANGUAGE FlexibleInstances          #-}
{-# LANGUAGE OverloadedStrings          #-}
{-# LANGUAGE DerivingStrategies         #-}
{-# LANGUAGE StandaloneDeriving         #-}
{-# LANGUAGE UndecidableInstances       #-}



PTH.share [PTH.mkPersist PTH.sqlSettings, PTH.mkMigrate "migrateAll"] [PTH.persistLowerCase|
  User sql=users
    name Text
    email Text
    age Int
```

```
    occupation Text
    UniqueEmail email
    deriving Show Read
|]
```

There are a lot of similarities to a normal data definition in Haskell. We've changed the formatting and reversed the order of the types and names. But you can still tell what's going on. The field names are all there. We've still derived basic instances like we would in Haskell.

But we've also added some new directives. For instance, we've stated what the table name should be (by default it would be user, not users). We've also created a UniqueEmail constraint. This tells our database that each user has to have a unique email. The migration will handle creating all the necessary indices for this to work!

This Template Haskell will generate the normal Haskell data type for us. All fields will have the prefix user and will be camel-cased, as we specified. The compiler will also generate certain special instances for our type. These will enable us to use Persistent's type-safe query functions. Finally, this code generates lenses that we'll use as filters in our queries, as we'll see later.

# ENTITIES AND KEYS

Persistent also has a construct allowing us to handle database IDs. For each type we put in the schema, we'll have a corresponding Entity type. An Entity refers to a row in our database, and it associates a database ID with the object itself. The database ID has the type SqlKey and is a wrapper around Int64. So the following would look like a valid entity:

```
import Database.Persist (Entity(..))

sampleUser :: Entity User
sampleUser = Entity (toSqlKey 1) $ User
  { userName = "admin"
  , userEmail = "admin@test.com"
  , userAge = 23
  , userOccupation = "System Administrator"
  }
```

This nice little abstraction that allows us to avoid muddling our user type with the database ID. This allows our other code to use a more pure User type.

# THE SQLPERSISTT MONAD

So now that we have the basics of our schema, how do we actually interact with our database from Haskell code? As a specific example, we'll be accessing a Postgresql database. This requires the SqlPersistT monad. All the query functions return actions in this monad. The monad transformer has to live on top of a monad that is MonadIO, since we obviously need IO to run database queries.

If we're trying to make a database query from a normal IO function, the first thing we need is a ConnectionString. This string encodes information about the location of the database. The connection string generally has 4-5 components. It has the host/IP address, the port, the database username, and the database name. So for instance if you're running Postgres on your local machine, you might have something like:

```
{-# LANGUAGE OverloadedStrings #-}

import Database.Persist.Postgresql (ConnectionString)

connString :: ConnectionString
connString = "host=127.0.0.1 port=5432 user=postgres dbname=postgres password=password"
```

Now that we have the connection string, we're set to call withPostgresqlConn. This function takes the string and then a function requiring a backend:

```
-- Also various constraints on the monad m
withPostgresqlConn :: (IsSqlBackend backend) => ConnectionString -> (backend -> m a) -> m a
``
The IsSqlBackend constraint forces us to use a type that conforms to Persistent's guidelines. The SqlPersistT monad is only a synonym for ReaderT backend. So in general, the only thing we'll do with this backend is use it as an argument to runReaderT. Once we've done this, we can pass any action within SqlPersistT as an argument to run.
```haskell
import Control.Monad.Logger (runStdoutLoggingT)
import Database.Persist.Postgresql (ConnectionString, withPostgresqlConn, SqlPersistT)
```
```

```
...

runAction :: ConnectionString -> SqlPersistT a ->  IO a
runAction connectionString action = runStdoutLoggingT $ withPostgresqlConn connectionString $
\backend ->
  runReaderT action backend
```

Note we add in a call to runStdoutLoggingT so that our action can log its results, as Persistent expects. This is necessary whenever we use withPostgresqlConn. Here's how we would run our migration function:

```
migrateDB :: IO ()
migrateDB = runAction connString (runMigration migrateAll)
``
This will create the users table, perfectly to spec with our data definition!

# QUERIES
Now let's wrap up by examining the kinds of queries we can run. The first thing we could do is insert a new user into our database. For this, Persistent has the insert function. When we insert the user, we'll get a key for that user as a result. Here's the type signature for insert specified to our particular User type:
```haskell
insert :: (MonadIO m) => User -> SqlPersistT m (Key User)
``
Then of course we can also do things in reverse. Suppose we have a key for our user and we want to get it out of the database. We'll want the get function. Of course this might fail if there is no corresponding user in the database, so we need a Maybe.
```haskell
get :: (MonadIO m) => Key User -> SqlPersistT m (Maybe User)
``
We can use these functions for any type satisfying the PersistRecordBackend class. This is included for free when we use the template Haskell approach. So you can use these queries on any type that lives in your schema.

But SQL allows us to do much more than query with the key. Suppose we want to get all the users that meet certain criteria. We'll want to use the selectList function, which replicates the behavior of the SQL SELECT command. It takes a couple different arguments for the different ways to run a selection. The two list types look a little complicated, but we'll
```

examine them in more detail:

```haskell
selectList
  :: PersistRecordBackend backend val
  => [Filter val]
  -> [SelectOpt val]
  -> SqlPersistT m [val]
```

As before, the PersistRecordBackend constraint is satisfied by any type in our TH schema. So we know our User type fits. So let's examine the first argument. It provides a list of different filters that will determine which elements we fetch. For instance, suppose we want all users who are younger than 25 and whose occupation is "Teacher". Remember the lenses I mentioned that get generated? We'll create two different filters on this by using these lenses.

```haskell
selectYoungTeachers :: (MonadIO m, MonadLogger m) => SqlPersistT m [User]
selectYoungTeachers = select [UserAge <. 25, UserOccupation ==. "Teacher"] []
```

We use the UserAge lens and the UserOccupation lens to choose the fields to filter on. We use a "less-than" operator to state that the age must be smaller than 25. Similarly, we use the ==. operator to match on the occupation. Then we provide an empty list of SelectOpts.

The second list of selection operations provides some other features we might expect in a select statement. First, we can provide an ordering on our returned data. We'll also use the generated lenses here. For instance, Asc UserEmail will order our list by email. Here's an ordered query where we also limit ourselves to 100 entries.

```haskell
selectYoungTeachers' :: (MonadIO m) => SqlPersistT m [User]
selectYoungTeachers' = selectList [UserAge <=. 25, UserOccupation ==. "Teacher"] [Asc
UserEmail]
```

The other types of SelectOpts include limits and offsets. For instance, we can further modify this query to exclude the first 5 users (as ordered by email) and then limit our selection to 100:

```haskell
selectYoungTeachers' :: (MonadIO m) => SqlPersistT m [Entity User]
selectYoungTeachers' = selectList
  [UserAge <. 25, UserOccupation ==. "Teacher"] [Asc UserEmail, OffsetBy 5, LimitTo 100]
```

And that's all there is to making queries that are type-safe and sensible. We know we're actually filtering on values that make sense for our types. We don't have to worry about typos ruining our code at runtime.

# CONCLUSION

Persistent gives us some excellent tools for interacting with databases from Haskell. The Template Haskell mechanisms generate a lot of boilerplate code that helps us. For instance, we can migrate our database to create the correct tables for our Haskell types. We also can perform queries that filter results in a type-safe way. All in all, it's a fantastic experience.

You should now move on to part 2 of this series, where we'll make a Web API using Servant. If you want to check out some more potential libraries for all your production needs, take a look at our Production Checklist!

# Building an API with Servant!

In part 1, we began our series on production Haskell techniques by learning about Persistent. We created a schema that contained a single User type that we could store in a Postgresql database. We examined a couple functions allowing us to make SQL queries about these users.

In this part, we'll see how we can expose this database to the outside world using an API. We'll construct our API using the Servant library. If you've already experienced Servant for yourself, you can move on to part 3, where you'll learn about caching and using Redis from Haskell.

Now, Servant involves some advanced type level constructs, so there's a lot to wrap your head around. There are definitely simpler approaches to HTTP servers than what Servant uses. But I've found that the power Servant gives us is well worth the effort. On the other hand, if you want some simpler approaches, you can take a look at our Production Checklist. It'll give you ideas for some other Web API libraries and a host of other tasks!

As a final note, make sure to look at the Github Repository! For this part, you'll mainly want to look at the Basic Server module.

# DEFINING OUR API

The first step in writing an API for our user database is to decide what the different endpoints are. We can decide this independently of what language or library we'll use. For this article, our API will have two different endpoints. The first will be a POST request to /users. This request will contain a "user" definition in its body, and the result will be that we'll create a user in our database. Here's a sample of what this might look like:

```
POST /users
{
  userName : "John Doe",
  userEmail : "john@doe.com",
  userAge : 29,
  userOccupation: "Teacher"
```

```
}
```

It will then return a response containing the database key of the user we created. This will allow any clients to fetch the user again. The second endpoint will use the ID to fetch a user by their database identifier. It will be a GET request to /users/:userid. So for instance, the last request might have returned us something like 16. We could then do the following:

```
GET /users/16
```

And our response would look like the request body from above.

# AN API AS A TYPE

So we've got our very simple API. How do we actually define this in Haskell, and more specifically with Servant? Well, Servant does something pretty unique. In Servant we define our API by using a type. Our type will include sub-types for each of the endpoints of our API. We combine the different endpoints by using the (:<|>) operator. I'll sometimes refer to this as "E-plus", for "endpoint-plus". This is a type operator, so remember we'll need the TypeOperators language extension. Here's the blueprint of our API:

```
type UsersAPI =
  fetchEndpoint
  :<|> createEndpoint
```

Now let's define what we mean by fetchEndpoint and createEndpoint. Endpoints combine different combinators that describe different information about the endpoint. We link combinators together with the (:>) operator, which I call "C-plus" (combinator plus). Here's what our final API looks like. We'll go through what each combinator means in the next section:

```
type UsersAPI =
       "users" :> Capture "userid" Int64 :> Get '[JSON] User
  :<|> "users" :> ReqBody '[JSON] User :> Post '[JSON] Int64
```

# COMBINATORS

Both of these endpoints have three different combinators. Let's start by examining the fetch endpoint. It starts off with a string combinator. This is a path component, allowing us to specify what url extension the caller should use to hit the endpoint. We can use this combinator multiple times to have a more complicated path for the endpoint. If we instead wanted this endpoint to be at /api/users/:userid then we'd change it to:

```
"api" :> "users" :> Capture "userid" Int64 :> Get '[JSON] User
```

The second combinator (Capture) allows us to get a value out of the URL itself. We give this value a name and then we supply a type parameter. We won't have to do any path parsing or manipulation ourselves. Servant will handle the tricky business of parsing the URL and passing us an Int64. If you want to use your own custom class as a piece of HTTP data, that's not too difficult. You'll just have to write an instance of the FromHttpApiData class. All the basic types like Int64 already have instances.

The final combinator itself contains three important pieces of information for this endpoint. First, it tells us that this is in fact a GET request. Second, it gives us the list of content-types that are allowable in the response. This is a type level list of content formats. Each type in this list must have different classes for serialization and deserialization of our data. We could have used a more complicated list like '[JSON, PlainText, OctetStream]. But for the rest of this article, we'll just use JSON. This means we'll use the ToJSON and FromJSON typeclasses for serialization.

The last piece of this combinator is the type our endpoint returns. So a successful request will give the caller back a response that contains a User in JSON format. Notice this isn't a Maybe User. If the ID is not in our database, we'll return a 401 error to indicate failure, rather than returning Nothing.

Our second endpoint has many similarities. It uses the same string path component. Then its final combinator is the same except that it indicates it is a POST request instead of a GET request. The second combinator then tells us what we can expect the request body to look like. In this case, the request body should contain a JSON representation of a User. It also requires a list of acceptable content types, and then the type we want, like the Get and Post combinators.

That completes the "definition" of our API. We'll need to add ToJSON and FromJSON instances of our User type in order for this to function. You can take a look at those on Github, and check out this article for more details on creating those instances!

# WRITING HANDLERS

Now that we've defined the type of our API, we need to write handler functions for each endpoint. This is where Servant's awesomeness kicks in. We can map each endpoint up to a function that has a particular type based on the combinators in the endpoint. So, first let's remember our endpoint for fetching a user:

```
"users" :> Capture "userid" Int64 :> Get '[JSON] User
```

The string path component doesn't add any arguments to our function. The Capture component will result in a parameter of type Int64 that we'll need in our function. Then the return type of our function should be User. This almost completely defines the type signature of our handler. We'll note though that it needs to be in the Handler monad. So here's what it'll look like:

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

We can also look at the type for our create endpoint:

```
"users" :> ReqBody '[JSON] User :> Post '[JSON] Int64
```

The parameter for a ReqBody parameter is just the type argument. So it will resolve this endpoint into the handler monad like so:

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

Now, we'll need to be able to access our Postgres database through both of these handlers. So they'll each get an extra parameter referring to the connection string (recall the PGInfo type alias). We'll pass that from our code so that by the time Servant is resolving the types, the parameter is accounted for:

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

# THE HANDLER MONAD

Before we go any further, we should discuss the Handler monad. This is a wrapper around the monad ExceptT ServantErr IO. In other words, each of these requests might fail. To make it fail, we can throw errors of type ServantErr. Then of course we can also call IO functions, because these are network operations.

Before we implement these functions, let's first write a couple simple helpers. These will use the runAction function from the last part to run database actions:

```
fetchUserPG :: PGInfo -> Int64 -> IO (Maybe User)
fetchUserPG connString uid = runAction connString (get (toSqlKey uid))

createUserPG :: PGInfo -> User -> IO Int64
createUserPG connString user = fromSqlKey <$> runAction connString (insert user)
```

For completeness (and use later in testing), we'll also add a simple delete function. We need the signature on the where clause for type inference:

```
deleteUserPG :: ConnectionString -> Int64 -> IO ()
deleteUserPG connString uid = runAction connString (delete userKey)
  where
    userKey :: Key User
    userKey = toSqlKey uid
```

Now we'll call these two functions from our Servant handlers. This will completely cover the case of the create endpoint. But we'll need a little bit more logic for the fetch endpoint. Since our functions are in the IO monad, we have to lift them up to Handler.

```
fetchUsersHandler :: ConnectionString -> Int64 -> Handler User
fetchUserHandler connString uid = do
  maybeUser <- liftIO $ fetchUserPG connString uid
  ...
```

```
createUserHandler :: ConnectionString -> User -> Handler Int64
createuserHandler connString user = liftIO $ createUserPG connString user
```

To complete our fetch handler, we need to account for a non-existent user. Instead of making the type of the whole endpoint a Maybe, we'll throw a ServantErr in this case. We can use one of the built-in Servant error functions, which correspond to normal error codes. Then we can update the body. In this case, we'll throw a 401 error. Here's how we do that:

```
fetchUsersHandler :: ConnectionString -> Int64 -> Handler User
fetchUserHandler connString uid = do
  maybeUser <- lift $ fetchUserPG connString uid
  case maybeUser of
    Just user -> return user
    Nothing -> Handler $ (throwE $ err401 { errBody = "Could not find user with ID: " ++ (show
uid)})

createUserHandler :: ConnectionString -> User -> Handler Int64
createuserHandler connString user = lift $ createUserPG connString user
```

And that's it! We're done with our handler functions!

# COMBINING IT ALL INTO A SERVER

Our next step is to create an object of type Server over our API. This is actually remarkably simple. When we defined the original type, we combined the endpoints with the (:<|>) operator. To make our Server, we do the same thing but with the handler functions:

```
usersServer :: ConnectionString -> Server UsersAPI
usersServer pgInfo =
  (fetchUsersHandler pgInfo) :<|>
  (createUserHandler pgInfo)
```

And Servant does all the work of ensuring that the type of each endpoint matches up with the type of the handler! Suppose we changed the type of our fetchUsersHandler so that it took a Key User instead of an Int64. We'd get a compile error:

```
fetchUsersHandler :: ConnectionString -> Key User -> Handler User
...

-- Compile Error!
• Couldn't match type ‘Key User' with ‘Int64'
      Expected type: Server UsersAPI
        Actual type: (Key User -> Handler User)
                     :<|> (User -> Handler Int64)
```

There's now a mismatch between our API definition and our handler definition. So Servant knows to throw an error! The one issue is that the error messages can be rather difficult to interpret sometimes. This is especially the case when your API becomes very large! The "Actual type" section of the above error will become massive! So always be careful when changing your endpoints! Frequent compilation is your friend!

# BUILDING THE APPLICATION

The final piece of the puzzle is to actually build an Application object out of our server. The first step of this process is to create a Proxy for our API. Remember that our API is a type, and not a term. But a Proxy allows us to represent this type at the term level. The concept is a little complicated, but the code is not!

```
import Data.Proxy

...

usersAPI :: Proxy UsersAPI
usersAPI = Proxy :: Proxy UsersAPI
```

Now we can make our runnable Application like so (assuming we have a Postgres connection):

```
serve usersAPI (usersServer connString)
```

We'll run this server from port 8000 by using the run function, again from Network.Wai. (See Github for a full list of imports). We'll fetch our connection string, and then we're good to go!

```
runServer :: IO ()
runServer = run 8000 (serve usersAPI (usersServer localConnString))
```

# CONCLUSION

The Servant library offers some truly awesome possibilities. We're able to define a web API at the type level. We can then define handler functions using the parameters the endpoints expect. Servant handles all the work of marshalling back and forth between the HTTP request and the native Haskell types. It also ensures a match between the endpoints and the handler function types!

Now you're ready for part 3 of our Real World Haskell series! You'll learn how we can modify our API to be faster by employing a Redis cache!

This part of the series gave a brief overview on Servant. But if you want a more in-depth introduction, you should check out my talk from Bayhac from April 2017! That talk was more exhaustive about the different combinators you can use in your APIs. It also showed authentication techniques, client functions and documentation. You can also check out the slides and code for that presentation!

On the other hand, Servant is also quite involved. If you want some ideas for a simpler solution, check out our Production Checklist! It'll give a couple other suggestions for Web API libraries and so much more!

# Redis Caching

In part 1 of this series we used Persistent to store a User type in a Postgresql database. Then in part 2 we used Servant to create a very simple API that exposed this database to the outside world. This week, we're going to look at how we can improve the performance of our API using a Redis cache.

If you've already read this part, you can move onto part 4 and learn how we test this system! You can also download our Production Checklist for a lot more ideas on useful tools to use in your production Haskell applications!

You can follow along the code for this part by looking at our Github repository! You'll specifically want to look at two files. First, the Cache module for the Redis specific code, and then the CacheServer module for an updated version of our Servant server.

# CACHING 101

One cannot overstate the importance of caching in both software and hardware. There's a hierarchy of memory types from registers to RAM, to the File system, to a remote database, and so on. Accessing each of these gets progressively slower (by orders of magnitude). But the faster means of storage are more expensive, so we can't always have as much as we'd like.

But memory usage operates on a very important principle. When we use a piece of memory once, we're very likely to use it again in the near-future. So when we pull something out of long-term memory, we can temporarily store it in short-term memory as well. This way when we need it again, we can get it faster. After a certain point, that item will be overwritten by other more urgent items. This is the essence of caching.

# REDIS

Redis is an application that allows us to create a key-value store of items. It functions like a database, except it only uses these keys. It lacks the sophistication of joins, foreign table references and indices. So we can't run the kinds of more complex queries that are possible on an SQL database. But we can run simple key lookups, and we can do them faster. In this article, we'll use Redis as a short-term cache for our user objects.

For this article, we've got one main goal for cache integration. Whenever we "fetch" a user using the GET endpoint in our API, we want to store that user in our Redis cache. Then the next time someone requests that user from our API, we'll grab them out of the cache. This will save us the trouble of making a longer call to our Postgres database.

#CONNECTING TO REDIS Haskell's Redis library has a lot of similarities to Persistent and Postgres. First, we'll need some sort of data that tells us where to look for our database. For Postgres, we used a simple ConnectionString with a particular format. Redis uses a full data type called ConnectInfo. (In our code, we alias this type as RedisInfo).

```
data ConnectInfo = ConnectInfo
  { connectHost :: HostName -- String
  , connectPort :: PortId   -- (Can just be a number)
  , connectAuth :: Maybe ByteString
  , connectDatabase :: Integer
  , connectMaxConnection :: Int
  , connectMaxIdleTime :: NominalDiffTime
  }
```

This has many of the same fields we stored in our PG string, like the host IP address, and the port number. The rest of this article assumes you are running a local Redis instance at port 6379. This means we can use defaultConnectInfo. As always, in a real system you'd want to grab this information out of a configuration, so you'd need IO.

```
fetchRedisConnection :: IO ConnectInfo
fetchRedisConnection = return defaultConnectInfo
```

With Postgres, we used withPostgresqlConn to actually connect to the database. With Redis, we do this with the connect function. We'll get a Connection object that we can use to run Redis actions.

```
connect :: ConnectInfo -> IO Connection
```

With this connection, we simply use runRedis, and then combine it with an action. Here's the wrapper runRedisAction we'll write for that:

```
runRedisAction :: ConnectInfo -> Redis a -> IO a
runRedisAction redisInfo action = do
  connection <- connect redisInfo
  runRedis connection action
```

# THE REDIS MONAD

Just as we used the SqlPersistT monad with Persist, we'll use the Redis monad to interact with our Redis cache. Our API is simple, so we'll stick to three basic functions. The real types of these functions are a bit more complicated. But this is because of polymorphism related to transactions, and we won't be using those.

```
get :: ByteString -> Redis (Either x (Maybe ByteString))
set :: ByteString -> ByteString -> Redis (Either x ())
setex :: ByteString -> ByteString -> Int -> Redis (Either x ())
```

Redis is a key-value store, so everything we set here will use ByteString values. The get function takes a ByteString of the key and delivers the value as another ByteString. The set function takes both the serialized key and value and stores them in the cache. The setex function does the same thing as set except that it also sets an expiration time for the item we're storing.

Expiration is a very useful feature, since most relational databases don't have this. The nature of a cache is that it's only supposed to store a subset of our information at any given time. If we never expire or delete anything, it might eventually store our whole database. That would defeat the purpose of using a cache! It's memory footprint should remain low compared to our database. So we'll use setex in our API.

# SAVING A USER IN REDIS

So now let's move on to the actions we'll actually use in our API. First, we'll write a function that will actually store a key-value pair of an Int64 key and the User in the database. Here's how we start:

```
cacheUser :: ConnectInfo -> Int64 -> User -> IO ()
cacheUser redisInfo uid user = runRedisAction redisInfo $ setex ??? ??? ???
```

All we need to do now is convert our key and our value to ByteString values. We'll keep it simple and use Data.ByteString.Char8 combined with our Show and Read instances. Then we'll create a Redis action using setex and expire the key after 3600 seconds (one hour).

```
import Data.ByteString.Char8 (pack, unpack)

...

cacheUser :: ConnectInfo -> Int64 -> User -> IO ()
cacheUser redisInfo uid user = runRedisAction redisInfo $ void $
  setex (pack . show $ uid) 3600 (pack . show $ user)
```

(We use void to ignore the result of the Redis call).

# FETCHING FROM REDIS

Fetching a user is a similar process. We'll take the connection information and the key we're looking for. The action we'll create uses the bytestring representation and calls get. But we can't ignore the result of this call like we could before! Retrieving anything gives us Either e (Maybe ByteString). A Left response indicates an error, while Right Nothing indicates the key doesn't exist. We'll ignore the errors and treat the result as Maybe User though. If any error comes up, we'll return Nothing. This means we run a simple pattern match:

```
fetchUserRedis :: ConnectInfo -> Int64 -> IO (Maybe User)
fetchUserRedis redisInfo uid = runRedisAction redisInfo $ do
  result <- Redis.get (pack . show $ uid)
  case result of
    Right (Just userString) -> return $ Just (read . unpack $ userString)
    _ -> return Nothing
```

If we do find something for that key, we'll read it out of its ByteString format and then we'll have our final User object.

# UPDATING OUR API

Now that we're all set up with our Redis functions, we have to update the fetchUsersHandler to use this cache. First, we now need to pass the Redis connection information as another parameter. For ease of reading, we'll refer to these using our type synonyms (PGInfo and RedisInfo) from now on:

```
type PGInfo = ConnectionString
type RedisInfo = ConnectInfo


...

fetchUsersHandler :: PGInfo -> RedisInfo -> Int64 -> Handler User
fetchUsersHandler pgInfo redisInfo uid = do
  ...
```

The first thing we'll try is to look up the user by their ID in the Redis cache. If the user exists, we'll immediately return that user.

```
fetchUsersHandler :: PGInfo -> RedisInfo -> Int64 -> Handler User
fetchUsersHandler pgInfo redisInfo uid = do
  maybeCachedUser <- liftIO $ fetchUserRedis redisInfo uid
  case maybeCachedUser of
    Just user -> return user
    Nothing -> do
      ...
```

If the user doesn't exist, we'll then drop into the logic of fetching the user in the database. We'll replicate our logic of throwing an error if we find that user doesn't actually exist. But if we find the user, we need one more step. Before we return it, we should call cacheUser and store it for the future.

```
fetchUsersHandler :: PGInfo -> RedisInfo -> Int64 -> Handler User
fetchUsersHandler pgInfo redisInfo uid = do
```

```
  maybeCachedUser <- liftIO $ fetchUserRedis redisInfo uid
  case maybeCachedUser of
    Just user -> return user
    Nothing -> do
      maybeUser <- liftIO $ fetchUserPG pgInfo uid
      case maybeUser of
        Just user -> liftIO (cacheUser redisInfo uid user) >> return user
        Nothing -> Handler $ (throwE $ err401 { errBody = "Could not find user with that ID"
})
```

Since we changed our type signature, we'll have to make a few other updates as well, but these are quite simple:

```
usersServer :: PGInfo -> RedisInfo -> Server UsersAPI
usersServer pgInfo redisInfo =
  (fetchUsersHandler pgInfo redisInfo) :<|>
  (createUserHandler pgInfo)


runServer :: IO ()
runServer = run 8000 (serve usersAPI (usersServer localConnString defaultConnectInfo))
```

And that's it! We have a functioning cache with expiring entries. This means that repeated queries to our fetch endpoint should be faster!

# CONCLUSION

Caching is a vitally important way for us to write software that is much faster for our users. Redis is a key-value store we can use as a cache for our most commonly used data. We can use it instead of forcing every single API call to hit our database. In Haskell, the Redis API requires everything to be a ByteString. So we have to deal with some logic surrounding encoding and decoding. But otherwise it operates in a very similar way to Persistent and Postgres. Next, you should move onto part 4, where we'll get into how we test this complicated system!

We're starting to get to the point where we're using a lot of different libraries in our Haskell application! It pays to know how to organize everything, so package management is vital! I tend to

use Stack for all my package management. It makes it quite easy to bring all these different libraries together. If you want to learn how to use Stack, check out our free Stack mini-course!

If you're already an expert in package management, perhaps you just want to expand your horizon of different libraries you know! In that case, take a look at our Production Checklist for some ideas!

# 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!

# Esqueleto and Complex Queries

In this series so far, we've done a real whirlwind tour of Haskell libraries. We created a database schema using Persistent and used it to write basic SQL queries in a type-safe way. We saw how to expose this database via an API with Servant. We also went ahead and added some caching to that server with Redis. Finally, we wrote some basic tests around the behavior of this API. By using Docker, we made those tests reproducible.

In this last part, we're going to review this whole process by adding another type to our schema. We'll write some new endpoints for an Article type, and link this type to our existing User type with a foreign key. Then we'll learn one more library: Esqueleto. Esqueleto improves on Persistent by allowing us to write type-safe SQL joins.

As with the previous articles, you can follow along with this code on the Github repository for this series. We'll be re-working our Schema a bit, so there are different files to reference for this part. Here are links to the new Schema, new Database library and updated server. Note that there is no caching nor any tests for this part.

If this series has whet your appetite for awesome Haskell libraries, download our Production Checklist for more ideas!

#ADDING ARTICLE TO OUR SCHEMA So our first step is to extend our schema with an Article type. We're going to give each article a title, some body text, and a timestamp for its publishing time. One new feature we'll see is that we'll add a foreign key referencing the user who wrote the article. Here's what it looks like within our schema:

```
PTH.share [PTH.mkPersist PTH.sqlSettings, PTH.mkMigrate "migrateAll"] [PTH.persistLowerCase|
 User sql=users
   ...

 Article sql=articles
   title Text
```

```
    body Text
    publishedTime UTCTime
    authorId UserId
    UniqueTitle title
    deriving Show Read Eq
|]
```

We can use UserId as a type in our schema. This will create a foreign key column when we create the table in our database. In practice, our Article type will look like this when we use it in Haskell:

```
data Article = Article
 { articleTitle :: Text
 , articleBody :: Text
 , articlePublishedTime :: UTCTime
 , articleAuthorId :: Key User
 }
```

This means it doesn't reference the entire user. Instead, it contains the SQL key of that user. Since we'll be adding the article to our API, we need to add ToJSON and FromJSON instances as well. These are pretty basic as well, so you can check them out in the schema definition if you're curious.

# ADDING ENDPOINTS

Now we're going to extend our API to expose certain information about these articles. First, we'll write a couple basic endpoints for creating an article and then fetching it by its ID:

```
type FullAPI =
      "users" :> Capture "userid" Int64 :> Get '[JSON] User
 :<|> "users" :> ReqBody '[JSON] User :> Post '[JSON] Int64
 :<|> "articles" :> Capture "articleid" Int64 :> Get '[JSON] Article
 :<|> "articles" :> ReqBody '[JSON] Article :> Post '[JSON] Int64
Now, we'll write a couple special endpoints. The first will take a User ID as a key and then
it will provide all the different articles the user has written. We'll do this endpoint as
/articles/author/:authorid.

...
```

```
 :<|> "articles" :> "author" :> Capture "authorid" Int64 :> Get '[JSON] [Entity Article]
```

Our last endpoint will fetch the most recent articles, up to a limit of 3. This will take no parameters and live at the /articles/recent route. It will return tuples of users and their articles, both as entities.

```
...
 :<|> "articles" :> "recent" :> Get '[JSON] [(Entity User, Entity Article)]
ADDING QUERIES (WITH ESQUELETO!)
```

Before we can actually implement these endpoints, we'll need to write the basic queries for them. For creating an article, we use the standard Persistent insert function:

```
createArticlePG :: PGInfo -> Article -> IO Int64
createArticlePG connString article = fromSqlKey <$> runAction connString (insert article)
```

We could do the same for the basic fetch endpoint. But we'll write this basic query using Esqueleto in the interest of beginning to learn the syntax. With Persistent, we used list parameters to specify different filters and SQL operations. Esqueleto instead uses a special monad to compose the different type of query. The general format of an esqueleto select call will look like this:

fetchArticlePG :: PGInfo -> Int64 -> IO (Maybe Article) fetchArticlePG connString aid = runAction connString selectAction where selectAction :: SqlPersistT (LoggingT IO) (Maybe Article) selectAction = select . from $ \articles -> do ... We use select . from and then provide a function that takes a table variable. Our first queries will only refer to a single table, but we'll see a join later. To complete the function, we'll provide the monadic action that will incorporate the different parts of our query.

The most basic filtering function we can call from within this monad is where_. This allows us to provide a condition on the query, much as we could with the filter list from Persistent. Esqueleto's filters also use the lenses generated by our schema.

```
selectAction :: SqlPersistT (LoggingT IO) (Maybe Article)
   selectAction = select . from $ \articles -> do
     where_ (articles ^. ArticleId ==. val (toSqlKey aid))
```

First, we use the ArticleId lens to specify which value of our table we're filtering. Then we specify the value to compare against. We not only need to lift our Int64 into an SqlKey, but we also need to lift that value using the val function.

But now that we've added this condition, all we need to do is return the table variable. Now, select returns our results in a list. But since we're searching by ID, we only expect one result. We'll use listToMaybe so we only return the head element if it exists. We'll also use entityVal once again to unwrap the article from its entity.

```
selectAction :: SqlPersistT (LoggingT IO) (Maybe Article)
selectAction = ((fmap entityVal) . listToMaybe) <$> (select . from $ \articles -> do
  where_ (articles ^. ArticleId ==. val (toSqlKey aid))
  return articles)
```

Now we should know enough that we can write out the next query. It will fetch all the articles that have been written by a particular user. We'll still be querying on the articles table. But now instead checking the article ID, we'll make sure the ArticleAuthorId is equal to a certain value. Once again, we'll lift our Int64 user key into an SqlKey and then again with val to compare it in "SQL-land".

```
fetchArticleByAuthorPG :: PGInfo -> Int64 -> IO [Entity Article]
fetchArticleByAuthorPG connString uid = runAction connString fetchAction
 where
   fetchAction :: SqlPersistT (LoggingT IO) [Entity Article]
   fetchAction = select . from $ \articles -> do
     where_ (articles ^. ArticleAuthorId ==. val (toSqlKey uid))
     return articles
```

And that's the full query! We want a list of entities this time, so we've taken out listToMaybe and entityVal.

Now let's write the final query, where we'll find the 3 most recent articles regardless of who wrote them. We'll include the author along with each article. So we're returning a list of of these different tuples of entities. This query will involve our first join. Instead of using a single table for this query, we'll use the InnerJoin constructor to combine our users table with the articles table.

```
fetchRecentArticlesPG :: PGInfo -> IO [(Entity User, Entity Article)]
fetchRecentArticlesPG connString = runAction connString fetchAction
 where
   fetchAction :: SqlPersistT (LoggingT IO) [(Entity User, Entity Article)]
   fetchAction = select . from $ \(users `InnerJoin` articles) -> do
```

Since we're joining two tables together, we need to specify what columns we're joining on. We'll use the on function for that:

```
fetchAction :: SqlPersistT (LoggingT IO) [(Entity User, Entity Article)]
fetchAction = select . from $ \(users `InnerJoin` articles) -> do
  on (users ^. UserId ==. articles ^. ArticleAuthorId)
```

Now we'll order our articles based on the timestamp of the article using orderBy. The newest articles should come first, so we'll use a descending order. Then we limit the number of results with the limit function. Finally, we'll return both the users and the articles, and we're done!

```
fetchAction :: SqlPersistT (LoggingT IO) [(Entity User, Entity Article)]
fetchAction = select . from $ \(users `InnerJoin` articles) -> do
  on (users ^. UserId ==. articles ^. ArticleAuthorId)
  orderBy [desc (articles ^. ArticlePublishedTime)]
  limit 3
  return (users, articles)
```

# CACHING DIFFERENT TYPES OF ITEMS

We won't go into the details of caching our articles in Redis, but there is one potential issue we want to observe. Currently we're using a user's SQL key as their key in our Redis store. So for instance, the string "15" could be such a key. If we try to naively use the same idea for our articles, we'll have a conflict! Trying to store an article with ID "15" will overwrite the entry containing the User!

But the way around this is rather simple. What we would do is that for the user's key, we would make the string something like users:15. Then for our article, we'll have its key be articles:15. As long as we deserialize it the proper way, this will be fine.

# FILLING IN THE SERVER HANDLERS

Now that we've written our database query functions, it is very simple to fill in our Server handlers. Most of them boil down to following the patterns we've already set with our other two endpoints:

```
fetchArticleHandler :: PGInfo -> Int64 -> Handler Article
fetchArticleHandler pgInfo aid = do
 maybeArticle <- liftIO $ fetchArticlePG pgInfo aid
 case maybeArticle of
   Just article -> return article
   Nothing -> Handler $ (throwE $ err401 { errBody = "Could not find article with that ID" })

createArticleHandler :: PGInfo -> Article -> Handler Int64
createArticleHandler pgInfo article = liftIO $ createArticlePG pgInfo article

fetchArticlesByAuthorHandler :: PGInfo -> Int64 -> Handler [Entity Article]
fetchArticlesByAuthorHandler pgInfo uid = liftIO $ fetchArticlesByAuthorPG pgInfo uid

fetchRecentArticlesHandler :: PGInfo -> Handler [(Entity User, Entity Article)]
fetchRecentArticlesHandler pgInfo = liftIO $ fetchRecentArticlesPG pgInfo
Then we'll complete our Server FullAPI like so:

fullAPIServer :: PGInfo -> Server FullAPI
fullAPIServer pgInfo =
 (fetchUsersHandler pgInfo) :<|>
 (createUserHandler pgInfo) :<|>
 (fetchArticleHandler pgInfo) :<|>
 (createArticleHandler pgInfo) :<|>
 (fetchArticlesByAuthorHandler pgInfo) :<|>
 (fetchRecentArticlesHandler pgInfo)
```

One interesting thing we can do is that we can compose our API types into different sections. For instance, we could separate our FullAPI into two parts. First, we could have the UsersAPI type from before, and then we could make a new type for ArticlesAPI. We can glue these together with the e-

plus operator just as we could individual endpoints!

```
type FullAPI = UsersAPI :<|> ArticlesAPI

type UsersAPI =
      "users" :> Capture "userid" Int64 :> Get '[JSON] User
 :<|> "users" :> ReqBody '[JSON] User :> Post '[JSON] Int64

type ArticlesAPI =
 "articles" :> Capture "articleid" Int64 :> Get '[JSON] Article
 :<|> "articles" :> ReqBody '[JSON] Article :> Post '[JSON] Int64
 :<|> "articles" :> "author" :> Capture "authorid" Int64 :> Get '[JSON] [Entity Article]
 :<|> "articles" :> "recent" :> Get '[JSON] [(Entity User, Entity Article)]
```

If we do this, we'll have to make similar adjustments in other areas combining the endpoints. For example, we would need to update the server handler joining and the client functions.

# CONCLUSION

This completes our overview of Real World Haskell skills. Over the course of this series, we've constructed a small web API from scratch. We've seen some awesome abstractions that let us deal with only the most important pieces of the project. Both Persistent and Servant generated a lot of extra boilerplate for us. This article showed the power of the Esqueleto library in allowing us to do type-safe joins. We also saw an end-to-end process of adding a new type and endpoints to our API.

But we've only scratched the surface of potential libraries to use in our Haskell code! Download our Production Checklist to get a glimpse of some of the possibilities!

Also be sure to check out our Haskell Stack mini-course!! It'll show you how to use Stack, so you can incorproate all the libraries from this series!