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

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!

Revision #1
Created 2022-03-11 06:24:38 UTC by gasick
Updated 2022-03-11 17:11:17 UTC by gasick