

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

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.

```

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:

```

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](#)!

Revision #1

Created 11 March 2022 06:13:54 by gasick

Updated 11 March 2022 17:11:17 by gasick