Haskell Tutorials Documentation

Release 1.0

Saurabh Nanda

Oct 09, 2017

Contents

Contents:

CHAPTER 1

Opaleye Tutorials

Contents:

Instant Gratification

Overview

We'll start by quickly running through the following DB operations, which should give you a sense of "instant gratification" (as the title says!) However, **do not** start writing apps with Opaleye just after reading this. As they say, a little knowledge is a dangerous thing! We **strongly encourage** you to read all the chapters in this tutorial before using Opaleye in any serious project.

- Connecting to the Postgres DB
- Selecting multiple rows
- Selecting a row
- Inserting a row
- Updating a row
- Selecting a single row

Preliminaries

- Install PostgreSQL. Create a database. Run the table creation script given below.

```
create table users(
   id serial primary key
  ,name text not null
  ,email text not null
);
```

```
insert into users(name, email) values ('John', 'john@mail.com');
insert into users(name, email) values ('Bob', 'bob@mail.com');
insert into users(name, email) values ('Alice', 'alice@mail.com');
```

- Install `opaleye` using your favourite package management tool
- Fire up your favourite text editor and copy-paste the code snippet below, and make sure it compiles without any errors.

```
{-# LANGUAGE Arrows                #-}
module Main where

import Opaleye
import Database.PostgreSQL.Simple
import Data.Profunctor.Product (p3)
import Control.Arrow

userTable :: Table
  (Column PGInt4, Column PGText, Column PGText)  -- read type
  (Column PGInt4, Column PGText, Column PGText) -- write type
userTable = Table "users" (p3 (required "id",
                              required "name",
                              required "email"))


selectAllRows :: Connection -> IO [(Int, String, String)]
selectAllRows conn = runQuery conn $ queryTable userTable

insertRow :: Connection -> (Int, String, String) -> IO ()
insertRow conn row = do
  runInsertMany conn userTable [(constant row)]
  return ()

selectByEmail :: Connection -> String -> IO [(Int, String, String)]
selectByEmail conn email = runQuery conn $ proc () ->
    do
      row@(_, _, em) <- queryTable userTable -< ()
      restrict -< (em .== constant email)
      returnA -< row

updateRow :: Connection -> (Int, String, String) -> IO ()
updateRow conn row@(key, name, email) = do
  runUpdate
    conn
    userTable
    (\_ -> constant row) -- what should the matching row be updated to
    (\ (k, _, _) -> k .== constant key) -- which rows to update?
  return ()

main :: IO ()
main = do
  conn <- connect ConnectInfo{connectHost="localhost"
                             ,connectPort=5432
                             ,connectDatabase="opaleye_tutorial"
                             ,connectPassword="opalaye_tutorial"
                             ,connectUser="opaleye_tutorial"
                             }
```

```
allRows <- selectAllRows conn
print allRows

insertRow conn (4, "Saurabh", "saurabhnanda@gmail.com")

row <- selectByEmail conn "saurabhnanda@gmail.com"
print row

updateRow conn (4, "Don", "corleone@puzo.com")

allRows <- selectAllRows conn
print allRows

return ()
```

Now read on to understand what this code is doing...

Teaching your table schema to Opaleye

Let's tackle the cryptic `userTable` definition at the very beginning of this code.

```
userTable :: Table
  (Column PGInt4, Column PGText, Column PGText)  -- read type
  (Column PGInt4, Column PGText, Column PGText) -- write type
userTable = Table "users" (p3 (required "id",
                               required "name",
                               required "email"))
```

Here's what it is basically teaching Opaleye:

- We will be reading rows of the type `(Column PGInt4, Column PGText, Column PGText)` from the table. The `Column a` type is what Opaleye uses to represent Postgres columns in Haskell-land. So `integer` columns become `Column PGInt4`, `varchar` columns become `Column PGText` and so on.
- We will be writing rows of the same type to the table. (Opaleye allows you to read and write rows of *different* types for very valid reasons. Read *Basic DB mappings* for more details on this.)
- The table's name is `users`
- The first column in the table is called `id`; it is *required*; and it maps to the first value of the tuple. Marking a column *required* means that you will have to specify a value for it whenever you are inserting or updating a row via Opaleye. You can mark a column as *optional* as well, but we talk about the subtle differences between *required*, *optional*, `NULL` and `NOT NULL` in the *Basic DB mappings* chapter.
- The second column in the table is called `name`; it is *required*; and it maps to the second value of the tuple.
- The third column in the table is called `email`; it is *required*; and it maps to the third value of the tuple.

We will need to use `userTable` to SELECT, INSERT, UPDATE, or DELETE from the `users` table via Opaleye.

To learn more about mapping different types of DB schemas to Opaleye's `Table` types, please read *Basic DB mappings* and *Advanced DB Mappings* chapters.

Connecting to the Postgresql database

Opaleye uses postgresql-simple to actually talk to the database.So, we first start by getting hold of a DB `Connection` using postgres-simples's `connect` function:

```
conn <- connect ConnectInfo{connectHost="localhost"
                           ,connectPort=5432
                           ,connectDatabase="opaleye_tutorial"
                           ,connectPassword="opalaye_tutorial"
                           ,connectUser="opaleye_tutorial"
                           }
```

Warning: Please take care to change the DB connection settings based on your local system.

Selecting all rows

Next we fetch and print all the rows from the `users` table:

```
allRows <- selectAllRows conn
print allRow
```

which calls `selectAllRows`:

```
selectAllRows :: Connection -> IO [(Int, String, String)]
selectAllRows conn = runQuery conn $ queryTable userTable
```

This uses `runQuery`, which is basically `SELECT` in Opaleye. Please take **special note** of the type signature of this function. It evaluates to `IO [(Int, String, String)]`, whereas we clearly told Opaleye that we will be reading rows of type `(Column PGInt4, Column PGText, ColumnPGText)`. So, why doesn't this function evaluate to `IO [(Column PGInt4, Column PGText, ColumnPGText)]`?

This is because Opaleye knows how to convert most basic data types from DB => Haskell (eg. `PGInt4` => `Int`). And vice versa.

However, here's a **gotcha!** Try compiling ths function *without* the type signature. The compiler will fail to infer the types. This is also due to the underlying infrastructure that Opaleye uses to convert DB => Haskell types. To understand this further, please read *Advanced DB Mappings*.

Inserting a row

```
insertRow :: Connection -> (Int, String, String) -> IO ()
insertRow conn row = do
  runInsertMany conn userTable [(constant row)]
  return ()
```

This function uses `runInsertMany` which is basically Opaleye's version of `INSERT`, **but** it only supports inserting *multiple rows*. This is why it is called `runInsertMany` instead of `runInsert` and the third argument is a *list* of rows.

Note: So, what does `constant row` do? It converts Haskell types => DB types, i.e. `(Int, String, String)` => `(Column PGInt4, Column PGText, Column PGText)` This is because we clearly told Opaleye that we will be writing rows of type `(Column PGInt4, Column PGText, Column PGText)` to `userTable`. However, our program doesn't deal with values of type `Column PGText` or `Column PGInt4` directly. So, this function - `insertRow` - gets a regular `(Int, String, String)` tuple and uses `constant` to convert it to `(Column PGInt4, Column PGText, Column PGText)` before handing it over to Opaleye.

Note: Strangely, while `runQuery` converts DB => Haskell types automagically, `runInsertMany` and `runUpdate` refuse to do Haskell => DB conversions on their own. Hence the need to do it explicitly when using these functions.

Updating a row

```
updateRow :: Connection -> (Int, String, String) -> IO ()
updateRow conn row@(key, name, email) = do
  runUpdate
    conn
    userTable
    (\_ -> constant row) -- what should the matching row be updated to
    (\ (k, _, _) -> k .== constant key) -- which rows to update?
  return ()
```

- As you can see from this function, updating rows in Opaleye is not very pretty! The biggest pain is that you cannot specify only a few columns from the row – you are forced to update the **entire row**. More about this in *Updating rows*.
- You already know what `constant row` does - it converts a Haskell datatype to its corresponding PG data type, which for some strange reason, Opaleye refuses to do here automagically.
- The comparison operator `.==` is what gets translated to equality operator in SQL. We cannot use Haskell's native equality operator because it represents equality in Haskell-land, whereas we need to represent equality when it gets converted to SQL-land. You will come across a lot of such special operators that map to their correspnding SQL parts.

Selecting a single row

Warning: Caution! Extreme hand-waving lies ahead. This is probably an incorrect explanation, but should work well-enough to serve your intuition for some time.

```
selectByEmail :: Connection -> String -> IO [(Int, String, String)]
selectByEmail conn email = runQuery conn $ proc () ->
    do
      row@(_, _, em) <- queryTable userTable -< ()
      restrict -< (em .== constant email)
      returnA -< row
```

And finally, the last section of this chapter introduces you to a weird arrow notation `-<`, which we have absolutely no clue about! All we know is that it works... mostly!

Check the type of `row@(_, _, em)` in your editor. It should be `(Column PGInt4, Column PGText, Column PGText)`, which means that if we do some hand-waving, here's what's happening in this function:

- `queryTable userTable -< ()` maps to a `SELECT` clause in SQL-land.
- The columns selected are *conceptually* capurted in `row@(_, _, em)` in SQL-land (which is why the row is a PG type instead of a Haskell type).
- `restrict` maps to `WHERE` in SQL.

- The `WHERE` condition, i.e. `em .== constant email` needs to convert `email`, which is of type `String`, to `Column PGText` (through the `constant` function) before it can compare it with `em`
- Finally `returnA` does some magic to return the row back to Haskell-land. Notice, that we don't have to do a DB => Haskell conversion here, because, as mentioned earlier, `runQuery` does that conversion automagically.

Basic DB mappings

Overview

In this chapter we will configure the DB<=>Haskell mapping for the following table:

- `tenants` - the master table of "tenants" in a typical multi-tenant SaaS app. You can think of a tenant as a "company account", where no two company accounts share any data.

At the end of the mapping process, we would like to have a schema as close to the following, as possible.

```
--
-- Tenants
--

create table tenants(
       id serial primary key
       ,created_at timestamp with time zone not null default current_
↪timestamp
       ,updated_at timestamp with time zone not null default current_
↪timestamp
       ,name text not null
       ,first_name text not null
       ,last_name text not null
       ,email text not null
       ,phone text not null
       ,status text not null default 'inactive'
       ,owner_id integer
       ,backoffice_domain text not null
       constraint ensure_not_null_owner_id check (status!='active' or owner_
↪id is not null)
);
create unique index idx_index_owner_id on tenants(owner_id);
create index idx_status on tenants(status);
create index idx_tenants_created_at on tenants(created_at);
create index idx_tenants_updated_at on tenants(updated_at);
create unique index idx_unique_tenants_backoffice_domain on␣
↪tenants(lower(backoffice_domain));
```

Further, we will see how Opaleye deals with the following four cases:

- Non-nullable columns without DB-specified defaults
- Non-nullable columns with DB-specified defaults
- Nullable columns without DB-specified defaults
- Nullable columns with DB-specified defaults - TODO: What's a good use-case for such a column?

Creating the DB

Since Opaleye does not have any support for migrations, setting up the DB schema is done by simply issuing SQL statement directly.

```
$ createdb vacationlabs
$ psql vacationlabs < includes/db-mappings/schema.sql
```

Now, to setup the DB<=>Haskell mapping for the `tenants` table, we'll walk down the following code:

```haskell
module DB where

import Opalaye
import Data.Text
import Data.Time (UTCTime)

data TenantPoly key createdAt updatedAt name status ownerId backofficeDomain = Tenant
↪{
  tenantKey :: key
  ,tenantCreatedAt :: createdAt
  ,tenantUpdatedAt :: updatedAt
  ,tenantName :: name
  ,tenantStatus :: status
  ,tenantOwnerId :: ownerId
  ,tenantBackofficeDomain :: backofficeDomain
  } deriving Show

type TenantPGWrite = TenantPoly
  (Maybe (Column PGInt8)) -- key
  (Maybe (Column PGTimestamptz)) -- createdAt
  (Column PGTimestamptz) -- updatedAt
  (Column PGText) -- name
  (Column PGText) -- status
  (Column (Nullable PGInt8)) -- ownerId
  (Column PGText) -- backofficeDomain

type TenantPGRead = TenantPoly
  (Column PGInt8) -- key
  (Column PGTimestamptz) -- createdAt
  (Column PGTimestamptz) -- updatedAt
  (Column PGText) -- name
  (Column PGText) -- status
  (Column (Nullable PGInt8)) -- ownerId
  (Column PGText) -- backofficeDomain

type Tenant = TenantPoly
  Integer -- key
  UTCTime -- createdAt
  UTCTime -- updatedAt
  Text -- name
  Text -- status
  (Maybe Integer) -- ownerId
  Text -- backofficeDomain

$(makeAdaptorAndInstance "pTenant" ''TenantPoly)
$(makeLensesWith abbreviatedFields ''TenantPoly)
```

```
tenantTable :: Table TenantPGWrite TenantPGRead
tenantTable = Table "tenants" (pTenant Tenant{
                                  tenantKey = optional "id"
                                  ,tenantCreatedAt = optional "created_at"
                                  ,tenantUpdatedAt = required "updated_at"
                                  ,tenantName = required "name"
                                  ,tenantStatus = required "status"
                                  ,tenantOwnerId = required "owner_id"
                                  ,tenantBackofficeDomain = required "backoffice_
↪domain"
                                  })
```

That's quite a **lot of code** to setup mappings for just one table! Most of it is just boilerplate that can easily be abstracted away using type-families or some TemplateHaskell. In fact there are libraries, such as, SilkOpaleye and dbrecord-opaleye which try to give Opaleye an easier-to-use API.

Strange polymorphic records

Firstly, let's tackle the strangely polymorphic `TenantPoly`.

```
data TenantPoly key createdAt updatedAt name status ownerId backofficeDomain = Tenant
↪{
  tenantKey :: key
  ,tenantCreatedAt :: createdAt
  ,tenantUpdatedAt :: updatedAt
  ,tenantName :: name
  ,tenantStatus :: status
  ,tenantOwnerId :: ownerId
  ,tenantBackofficeDomain :: backofficeDomain
  } deriving Show
```

This is a **base type** which defines the **shape** of a set of related record-types (namely `TenantPGRead`, `TenantPGWrite`, and `Tenant`). `TenantPoly` is polymorphic over every single field of the record. This allows us to easily change the type of each field, while ensuring that the *shape* of all these related records is always the same. (*Why* would we want records with similar shapes, but different types, will get clearer in a moment - hang in there!) Generally, `TenantPoly` is never used directly in any Opaleye operation. The concrete types - `TenantPGRead` `TenantPGWrite` and `Tenant` - are used instead.

At the the time of writing, Opalaye does **not do any reflection** on the DB schema whatsoever. This is very different from something like Rails (in the Ruby world) and HRR (in the Haskell world), which generate the DB<=>Haskell mappings on the basis of schema reflection). So, Opaleye does not know what data-types to expect for each column when talking to the DB. Therefore, we have to teach it by essentially duplicating the SQL column definitions in Haskell. This is precisely what `TenantPGRead`, `TenantPGWrite`, `makeAdaptorAndInstance` and `tenantTable` do, and this is what we absolutely hate about Opaleye!

Note: We've scratched our own itch here and are working on Opaleye Helpers to help remove this duplication and boilerplate from Opaleye.

```
type TenantPGWrite = TenantPoly
  (Maybe (Column PGInt8)) -- key
  (Maybe (Column PGTimestamptz)) -- createdAt
  (Column PGTimestamptz) -- updatedAt
  (Column PGText) -- name
  (Column PGText) -- status
```

```
  (Column (Nullable PGInt8)) -- ownerId
  (Column PGText) -- backofficeDomain

type TenantPGRead = TenantPoly
  (Column PGInt8) -- key
  (Column PGTimestamptz) -- createdAt
  (Column PGTimestamptz) -- updatedAt
  (Column PGText) -- name
  (Column PGText) -- status
  (Column (Nullable PGInt8)) -- ownerId
  (Column PGText) -- backofficeDomain

$(makeAdaptorAndInstance "pTenant" ''TenantPoly)

tenantTable :: Table TenantPGWrite TenantPGRead
tenantTable = Table "tenants" (pTenant Tenant{
                                  tenantKey = optional "id"
                                  ,tenantCreatedAt = optional "created_at"
                                  ,tenantUpdatedAt = optional "updated_at"
                                  ,tenantName = required "name"
                                  ,tenantStatus = required "status"
                                  ,tenantOwnerId = required "owner_id"
                                  ,tenantBackofficeDomain = required "backoffice_
↪domain"
                                  })
```

Different types for read & write

With this, we witness another quirk (and power) of Opaleye. It allows us to define different types for the read (SELECT) and write (INSERT/UPDATE) operations. In fact, our guess is that, to achieve type-safety, it is forced to do this. Let us explain. If you're using standard auto-increment integers for the primary key (which most people do), you essentially end-up having two different types for the `INSERT` and `SELECT` operations. In the `INSERT` operation, you *should not* be specifying the `id` field/column. Whereas, in the `SELECT` operation, you will always be reading it. (Look at Persistent if you want to see another approach of solving this problem.)

One way to avoid having separate types for read & write operations, is to allow the PK field to be `undefined` in Haskell, being careful not to evaluate it when dealing with a record that has not yet been saved to the DB. We haven't tried this approach yet, but we're very sure it would require us to teach Opalaye how to map `undefined` values to SQL. Nevertheless, depending upon partially defined records for something as common as `INSERT` operations does not bode too well for a language that prides itself on type-safety and correctness.

Therefore, the need for two separate types: `TenantPGRead` and `TenantPGWrite`, with subtle differences. But, before we discuss the differences, we need to understand how Opaleye deals with `NULL` values and "omitted columns".

Handling `NULL` and database defaults

Let's look at the types of a few fields from `TenantPGWrite` and how they interact with `NULL` values and the `DEFAULT` value in the DB:

The (Column a) types

- `updatedAt` of type `(Column PGTimestamptz)` corresponding to `updated_at timestamp with time zone not null default current_timestamp`
- `name` of type `(Column PGText)` corresponding to `name text not null`

- `status` of type `(Column PGText)` corresponding to `status text not null default 'inactive'`

In each of these cases you **have to** specify the field's value whenever you are inserting or updating via Opaleye. Moreover, the type ensures that you cannot assign a `null` value to any of them at the Haskell-level. **Please note,** `null` is NOT the same as `Nothing`

The (Maybe (Column a)) types

- `key` of type `(Maybe (Column PGInt8))` corresponding to `id serial primary key`
- `createdAt` of type `(Maybe (Column PGTimestamptz))` corresponding to `created_at timestamp with time zone not null default current_timestamp`

In both these cases, during an INSERT, if the value is a `Nothing`, the entire column itself will be omitted from the INSERT statement and its fate will be left to the DB.

The (Column (Nullable a)) types

- `ownerId` of type `(Column (Nullable PGInt8))` corresponding to `owner_id integer`

In this case, while you **have to** specify a value at the Haskell level, you can specify a `null` as well.

For example, this is a possible INSERT operation:

```
runInsertMany
  conn  -- PG Connection
  userTable -- Opaleye table identifer
  [(
    TenantPGWrite
      {
        tenantKey              = Nothing -- column will be omitted from query; will␣
↪use DB's DEFAULT
      , tenantCreatedAt        = Just $ pgUTCTime someTime -- column will NOT be␣
↪omitted from query; will NOT use DB's DEFAULT
      , tenantUpdatedAt        = pgUTCTime someTime
      , tenantName             = pgText "Saurabh"
      , tenantStatus           = pgText "inactive"
      , tenantOwnerId          = null -- specfically store a NULL value
      , tenantBackofficeDomain = pgText "saurabh.vacationlabs.com"
      }
  )]
```

Note: Please make sure you understand the difference between `Maybe (Column a)` and `Column (Nullable a)`. And possibly `Maybe (Column (Nullable a))` - although we're not sure how useful the last one is!

Different types for read & write - again

Now, coming back to the subtle differences in `TenantPGWrite` and `TenantPGRead`:

- While writing, we may **omit** the `key` and `createdAt` columns (because their type is `(Maybe (Column x))` in `TenantPGWrite`)
- However, while reading, there is really no way to omit columns. You can, of course select 2 columns instead of 3, but that would result in completely different data types, eg: `(Column PGText, Column PGInt4)` vs `(Column PGText, Column PGInt4, Column PGTimestamptz)`.

- If your result-set is obtained from a LEFT JOIN, you can have a PGRead type of `(Column a, Column b, Column (Nullable c), Column (Nullable d))`, with the Nullable columns repreenting the result-set in a type-safe manner.

Note: Here are two small exercises:

What if `ownerId` had the following types. What would it mean? What is a possible use-case for having these types?

- `TenantPGWrite`: (Maybe (Column (Nullable PGInt8)))
- `TenantPGRead`: (Column (Nullable PGInt8))

And what about the following types for `onwerId`?

- `TenantPGWrite`: (Maybe (Column PGInt8))
- `TenantPGRead`: (Column (Nullable PGInt8))

Making things even more typesafe: If you notice, `TenantPGWrite` has the `key` field as `(Maybe (Column PGInt8))`, which makes it *omittable*, but it also makes it *definable*. Is there really any use of sending the primary-key's value from Haskell to the DB? In most cases, we think not. So, if we want to make this interface uber typesafe, Opaleye allows us to do the following as well (notice the type of `key`):

```
type TenantPGWrite = TenantPoly
  () -- key
  (Maybe (Column PGTimestamptz)) -- createdAt
  (Column PGTimestamptz) -- updatedAt
  (Column PGText) -- name
  (Column PGText) -- status
  (Column (Nullable PGInt8)) -- ownerId
  (Column PGText) -- backofficeDomain
```

See also:

You'll need to do some special setup for this to work as described in *Making columns read-only*

Wrapping-up

Coming to the last part of setting up DB<=>Haskell mapping with Opaleye, we need to issue these magic incantations:

```
$(makeAdaptorAndInstance "pTenant" ''TenantPoly)

tenantTable :: Table TenantPGWrite TenantPGRead
tenantTable = Table "tenants" (pTenant Tenant{
                                 tenantKey = optional "id"
                                 ,tenantCreatedAt = optional "created_at"
                                 ,tenantUpdatedAt = optional "updated_at"
                                 ,tenantName = required "name"
                                 ,tenantStatus = required "status"
                                 ,tenantOwnerId = required "owner_id"
                                 ,tenantBackofficeDomain = required "backoffice_
↪domain"
                                 })
```

The TH splice - `makeAdaptorAndInstance` - does two very important things:

- Defines the `pTenant` function, which is subsequently used in `tenantTable`

- Defines the `Default` instance for `TenantPoly` (this is not `Data.Default`, but the poorly named *Data.Profunctor.Product.Default*

Right now, we don't need to be bothered with the internals of `pTenant` and `Default`, but we *will* need them when we want to do some advanced DB<=>Haskell mapping. Right now, what we need to be bothered about is `tenantTable`. That is what we've been waiting for! This is what represents the `tenants` table in the Haskell land. Every SQL operation on the `tenants` table will need to reference `tenantsTable`. And while defining `tenantsTable` we've finally assembled the last piece of the puzzle: field-name <=> column-name mappings AND the name of the table! (did you happen to forget about them?)

Note: We're not really clear why we need to specify `optional` and `required` in the table definition when `TenantPGWrite` has already defined which columns are optional and which are required.

And, one last thing. We've been talking about `PGText`, `PGTimestamptz`, and `PGInt8` till now. These aren't the regular Haskell types that we generally deal with! These are representations of native PG types in Haskell. You would generally not build your app with these types. Instead, you would use something like `Tenant`, defined below:

```
type Tenant = TenantPoly
  Integer -- key
  UTCTime -- createdAt
  UTCTime -- updatedAt
  Text -- name
  Text -- status
  (Maybe Integer) -- ownerId
  Text -- backofficeDomain
```

Remember these three types and their purpose. We will need them when we're inserting, udpating, and selecting rows.

- `TenantPGWrite` defines the record-type that can be written to the DB in terms of PG types.
- `TenantPGRead` defines the record-type that can be read from the DB in terms of PG types.
- `Tenant` defines the records that represents rows of the `tenants` table, in terms of Haskell types. We haven't yet split this into separate read and write types.

Template Haskell expansion

If you're curious, this is what the TH splice expands to (not literally, but conceptually). You might also want to read the [documentation of Data.Profunctor.Product.TH](https://hackage.haskell.org/package/product-profunctors-0.7.1.0/docs/Data-Profunctor-Product-TH.html) to understand what's going on here.

```
pTenant :: ProductProfunctor p =>
  TenantPoly
    (p key0 key1)
    (p createdAt0 createdAt1)
    (p updatedAt0 updatedAt1)
    (p name0 name1)
    (p status0 status1)
    (p ownerId0 ownerId1)
    (p backofficeDomain0 backofficeDomain1)
  -> p  (TenantPoly key0 createdAt0 updatedAt0 name0 status0 ownerId0␣
↪backofficeDomain0)
        (TenantPoly key1 createdAt1 updatedAt1 name1 status ownerId1␣
↪backofficeDomain1)
pTenant = (((dimap toTuple fromTuple) . Data.Profunctor.Product.p7) . toTuple)
```

```
  where
      toTuple (Tenant key createdAt updatedAt name status ownerId backofficeDomain)
        = (key, createdAt, updatedAt, name, status, ownerId, backofficeDomain)
      fromTuple (key, createdAt, updatedAt, name, status, ownerId, backofficeDomain)
        = Tenant key createdAt updatedAt name status ownerId backofficeDomain


instance (ProductProfunctor p,
          Default p key0 key1,
          Default p createdAt0 createdAt1,
          Default p updatedAt0 updatedAt1,
          Default p name0 name1,
          Default p status0 status,
          Default p ownerId0 ownerId1,
          Default p backofficeDomain0 backofficeDomain1) =>
         Default p (TenantPoly key0 createdAt0 updatedAt0 name0 status0 ownerId0␣
↪backofficeDomain0)
                   (TenantPoly key1 createdAt1 updatedAt1 name1 status ownerId1␣
↪backofficeDomain1) where
  def = pTenant (Tenant def def def def def def def)
```

Advanced DB Mappings

Overview

In this chapter we'll build upon what we did in the last chapter:

- We'll modify the `tenants` table, to be a little more typesafe by changing the type of the `status` column to a Postgres `ENUM` (rather than a `text`) and mapping it to a Haskell ADT.
- We'll add a new table called `products` that will be used to store information of various products in our hypothetical ecommerce store
- We'll change the `id` and `createdAt` columns to be read-only, for greater type-safety while inserting records.
- We'll change the primary keys, `tenants.id` and `products.id` to `TenantId` and `ProductId` respecively. Again, for greater type-safety.

SQL for table creation

```
--
-- Tenants
--

create type tenant_status as enum('active', 'inactive', 'new');
create table tenants(
       id serial primary key
       ,created_at timestamp with time zone not null default current_
↪timestamp
       ,updated_at timestamp with time zone not null default current_
↪timestamp
       ,name text not null
       ,first_name text not null
       ,last_name text not null
```

```
        ,email text not null
        ,phone text not null
        ,status tenant_status not null default 'inactive'
        ,owner_id integer
        ,backoffice_domain text not null
        constraint ensure_not_null_owner_id check (status!='active' or owner_
↪id is not null)
);
create unique index idx_index_owner_id on tenants(owner_id);
create index idx_status on tenants(status);
create index idx_tenants_created_at on tenants(created_at);
create index idx_tenants_updated_at on tenants(updated_at);
create unique index idx_unique_tenants_backoffice_domain on␣
↪tenants(lower(backoffice_domain));

---
--- Products
---

create type product_type as enum('physical', 'digital');
create table products(
        id serial primary key
        ,created_at timestamp with time zone not null default current_
↪timestamp
        ,updated_at timestamp with time zone not null default current_
↪timestamp
        ,tenant_id integer not null references tenants(id)
        ,name text not null
        ,description text
        ,url_slug text not null
        ,tags text[] not null default '{}'
        ,currency char(3) not null
        ,advertised_price numeric not null
        ,comparison_price numeric not null
        ,cost_price numeric
        ,type product_type not null
        ,is_published boolean not null default false
        ,properties jsonb
);
create unique index idx_products_name on products(tenant_id, lower(name));
create unique index idx_products_url_sluf on products(tenant_id, lower(url_
↪slug));
create index idx_products_created_at on products(created_at);
create index idx_products_updated_at on products(updated_at);
create index idx_products_comparison_price on products(comparison_price);
create index idx_products_tags on products using gin(tags);
create index idx_product_type on products(type);
create index idx_product_is_published on products(is_published);
```

Code that we'll run through

```
{-# LANGUAGE Arrows                #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings     #-}
{-# LANGUAGE TemplateHaskell       #-}
```

```haskell

module Main where

import           Data.Aeson
import           Data.Profunctor.Product
import           Data.Profunctor.Product.Default
import           Data.Profunctor.Product.TH          (makeAdaptorAndInstance)
import           Data.Scientific
import           Data.ByteString hiding (putStrLn)
import           Data.Text
import           Data.Time
import           Opaleye

import           Database.PostgreSQL.Simple
import           Database.PostgreSQL.Simple.FromField (Conversion,
                                                       FromField (..),
                                                       ResultError (..),
                                                       returnError)

import           Control.Arrow
import           Prelude                             hiding (id)

-- Tenant stuff

newtype TenantId = TenantId Int deriving(Show)

data TenantStatus = TenantStatusActive | TenantStatusInActive | TenantStatusNew
  deriving (Show)

data TenantPoly key name fname lname email phone status b_domain = Tenant
  { tenant_id               :: key
  , tenant_name             :: name
  , tenant_firstname        :: fname
  , tenant_lastname         :: lname
  , tenant_email            :: email
  , tenant_phone            :: phone
  , tenant_status           :: status
  , tenant_backofficedomain :: b_domain
  } deriving (Show)

type Tenant = TenantPoly TenantId Text Text Text Text Text TenantStatus Text

type TenantTableW = TenantPoly
  (Maybe (Column PGInt4))
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)

type TenantTableR = TenantPoly
  (Column PGInt4)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
```

```
  (Column PGText)
  (Column PGText)
  (Column PGText)

-- Product stuff

newtype ProductId = ProductId Int deriving (Show)

data ProductType = ProductPhysical | ProductDigital deriving (Show)

data ProductPoly id created_at updated_at tenant_id name description url_slug tags␣
↪currency advertised_price comparison_price cost_price product_type is_published␣
↪properties = Product {
      product_id                 :: id
    , product_created_at         :: created_at
    , product_updated_at         :: updated_at
    , product_tenant_id          :: tenant_id
    , product_name               :: name
    , product_description        :: description
    , product_url_slug           :: url_slug
    , product_tags               :: tags
    , product_currency           :: currency
    , product_advertised_price   :: advertised_price
    , product_comparison_price   :: comparison_price
    , product_cost_price         :: cost_price
    , product_product_type       :: product_type
    , product_is_published       :: is_published
    , product_properties         :: properties
  } deriving (Show)

type Product = ProductPoly ProductId UTCTime UTCTime TenantId Text (Maybe Text) Text␣
↪[Text] Text Scientific Scientific (Maybe Scientific) ProductType Bool Value
type ProductTableW = ProductPoly
  (Maybe (Column PGInt4))
  (Maybe (Column PGTimestamptz))
  (Maybe (Column PGTimestamptz))
  (Column PGInt4)
  (Column PGText)
  (Maybe (Column (Nullable PGText)))
  (Column PGText)
  (Column (PGArray PGText))
  (Column PGText)
  (Column PGFloat8)
  (Column PGFloat8)
  (Maybe (Column (Nullable PGFloat8)))
  (Column PGText)
  (Column PGBool)
  (Column PGJsonb)

type ProductTableR = ProductPoly
  (Column PGInt4)
  (Column PGTimestamptz)
  (Column PGTimestamptz)
  (Column PGInt4)
  (Column PGText)
  (Column (Nullable PGText))
  (Column PGText)
  (Column (PGArray PGText))
```

```haskell
    (Column PGText)
    (Column PGFloat8)
    (Column PGFloat8)
    (Column (Nullable PGFloat8))
    (Column PGText)
    (Column PGBool)
    (Column PGJsonb)

-- Table defs

$(makeAdaptorAndInstance "pTenant" ''TenantPoly)
tenantTable :: Table TenantTableW TenantTableR
tenantTable = Table "tenants" (pTenant
   Tenant {
     tenant_id = (optional "id"),
     tenant_name = (required "name"),
     tenant_firstname = (required "first_name"),
     tenant_lastname = (required "last_name"),
     tenant_email = (required "email"),
     tenant_phone = (required "phone"),
     tenant_status = (required "status"),
     tenant_backofficedomain = (required "backoffice_domain")
   }
 )

$(makeAdaptorAndInstance "pProduct" ''ProductPoly)

productTable :: Table ProductTableW ProductTableR
productTable = Table "products" (pProduct
     Product {
       product_id = (optional "id"),
       product_created_at = (optional "created_at"),
       product_updated_at = (optional "updated_at"),
       product_tenant_id = (required "tenant_id"),
       product_name = (required "name"),
       product_description = (optional "description"),
       product_url_slug = (required "url_slug"),
       product_tags = (required "tags"),
       product_currency = (required "currency"),
       product_advertised_price = (required "advertised_price"),
       product_comparison_price = (required "comparison_price"),
       product_cost_price = (optional "cost_price"),
       product_product_type = (required "type"),
       product_is_published = (required "is_published"),
       product_properties = (required "properties") })

-- Instance declarations for custom types
-- For TenantStatus

instance FromField TenantStatus where
  fromField field mb_bytestring = makeTenantStatus mb_bytestring
    where
    makeTenantStatus :: Maybe ByteString -> Conversion TenantStatus
    makeTenantStatus (Just "active") = return TenantStatusActive
    makeTenantStatus (Just "inactive") = return TenantStatusInActive
    makeTenantStatus (Just "new") = return TenantStatusNew
    makeTenantStatus (Just _) = returnError ConversionFailed field "Unrecognized tenant status"
```

```haskell
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty tenant status"

instance QueryRunnerColumnDefault PGText TenantStatus where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For ProductType

instance FromField ProductType where
  fromField field mb_bytestring = makeProductType mb_bytestring
    where
    makeProductType :: Maybe ByteString -> Conversion ProductType
    makeProductType (Just "physical") = return ProductPhysical
    makeProductType (Just "digital") = return ProductDigital
    makeProductType (Just _) = returnError ConversionFailed field "Unrecognized␣
↪product type"
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty product type"

instance QueryRunnerColumnDefault PGText ProductType where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For productId

instance FromField ProductId where
  fromField field mb_bytestring = ProductId <$> fromField field mb_bytestring

instance QueryRunnerColumnDefault PGInt4 ProductId where
  queryRunnerColumnDefault = fieldQueryRunnerColumn
-- For TenantId
instance FromField TenantId where
  fromField field mb_bytestring = TenantId <$> fromField field mb_bytestring

instance QueryRunnerColumnDefault PGInt4 TenantId where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For Scientific we didn't have to implement instance of fromField
-- because it is already defined in postgresql-simple

instance QueryRunnerColumnDefault PGFloat8 Scientific where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- Default instance definitions for custom datatypes for converison to
-- PG types while writing into tables

-- For Tenant stuff

instance Default Constant TenantStatus (Column PGText) where
  def = Constant def'
    where
      def' :: TenantStatus -> (Column PGText)
      def' TenantStatusActive = pgStrictText "active"
      def' TenantStatusInActive = pgStrictText "inactive"
      def' TenantStatusNew = pgStrictText "new"

instance Default Constant TenantId (Maybe (Column PGInt4)) where
  def = Constant (\(TenantId x) -> Just $ pgInt4 x)

-- For Product stuff

```

```haskell
instance Default Constant ProductType (Column PGText) where
  def = Constant def'
    where
      def' :: ProductType -> (Column PGText)
      def' ProductDigital = pgStrictText "digital"
      def' ProductPhysical = pgStrictText "physical"

instance Default Constant ProductId (Maybe (Column PGInt4)) where
  def = Constant (\(ProductId x) -> Just $ constant x)

instance Default Constant Scientific (Column PGFloat8) where
  def = Constant (pgDouble.toRealFloat)

instance Default Constant Scientific (Column (Nullable PGFloat8)) where
  def = Constant (toNullable.constant)

instance Default Constant Text (Column (Nullable PGText)) where
  def = Constant (toNullable.pgStrictText)

instance Default Constant UTCTime (Maybe (Column PGTimestamptz)) where
  def = Constant ((Just).pgUTCTime)

instance Default Constant TenantId (Column PGInt4) where
  def = Constant (\(TenantId x) -> constant x)

getProducts :: IO [Product]
getProducts = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runQuery conn $ queryTable productTable

getTenants :: IO [Tenant]
getTenants = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runQuery conn $ queryTable tenantTable

insertTenant :: IO ()
insertTenant = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runInsertManyReturning conn tenantTable [constant getTestTenant] (\x -> x) :: IO [Tenant]
  return ()

insertProduct :: IO ()
insertProduct = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  product <- getTestProduct
  runInsertManyReturning conn productTable [constant product] (\x -> x) :: IO [Product]
  return ()

getTestTenant :: TenantIncoming
getTestTenant = Tenant {
  tenant_id = (),
  tenant_name = "Tenant Bob",
  tenant_firstname = "Bobby",
  tenant_lastname = "Bob",
  tenant_email = "bob@gmail.com",
  tenant_phone = "2255",
```

```
  tenant_status = TenantStatusInActive,
  tenant_backofficedomain = "bob.com"
}

getTestProduct :: IO Product
getTestProduct = do
  time <- getCurrentTime
  let (Just properties) =  decode "{\"weight\": \"200gm\"}" :: Maybe Value
  return $ Product {
    product_id = (ProductId 5),
    product_created_at = time,
    product_updated_at = time,
    product_tenant_id = (TenantId 5),
    product_name = "snacks",
    product_description = Just "",
    product_url_slug = "",
    product_tags = ["tag1", "tag2"],
    product_currency = "INR",
    product_advertised_price = 30,
    product_comparison_price = 45,
    product_cost_price = Nothing,
    product_product_type = ProductPhysical,
    product_is_published = False,
    product_properties = properties
  }

main :: IO ()
main = do
  insertTenant
  insertProduct
  tenants <- getTenants
  products <- getProducts
  putStrLn $ show tenants
  putStrLn $ show products

-- Output
--
-- [Tenant {tenant_id = TenantId 1, tenant_name = "Tenant John", tenant_firstname
-- = "John", tenant_lastname = "Honai", tenant_email = "john@mail.com", tenant_pho
-- ne = "2255", tenant_status = TenantStatusInActive, tenant_backofficedomain = "j
-- honhonai.com"}]
-- [Product {product_id = ProductId 1, product_created_at = 2016-11-27 10:24:31.60
-- 0244 UTC, product_updated_at = 2016-11-27 10:24:31.600244 UTC, product_tenant_i
-- d = TenantId 1, product_name = "Biscuits", product_description = Just "Biscuits
-- , you know..", product_url_slug = "biscuits", product_tags = ["bakery","snacks"
-- ], product_currency = "INR", product_advertised_price = 40.0, product_compariso
-- n_price = 55.0, product_cost_price = Just 34.0, product_product_type = ProductP
-- hysical, product_is_published = False, product_properties = Object (fromList [(
-- "weight",String "200gm")])}]
```

Warning: In the code given above, we are using `PGFloat8` to represent monetary values. This is a **bad idea** and absolutely **not recommended.** We are forced to do this because Opaleye's support for Postgres `NUMERIC` datatype is not really complete.

Core mechanism for mapping custom Haskell types to PG types

There are three typeclasses at play in converting values between Haskell types (like Int, Text and other user defined types) and PG types (like PGInt4, PGText etc). These are:

- `FromField`
- `QueryRunnerColumnDefault`
- `Default` (*not* `Data.Default`)

FromField

This is a typeclass defined by the postgresql-simple library. This typeclass decides how values read from database are converted to their Haskell counterparts. It is defined as:

```
class FromField a where
  fromField :: FieldParser a

type FieldParser a = Field -> Maybe ByteString -> Conversion a
```

The basic idea of this typeclass is simple. It wants you to define a function `fromField` which will be passed the following:

- `Field` - a record holding a lot of metadata about the underlying Postgres column
- `Maybe ByteString` - the raw value of that column

You are expected to return a `Conversion a` which is conceptually an *action*, which when evaluated will do the conversion from `Maybe ByteString` to your desired type `a`.

Diligent readers will immediately have the following questions:

What kind of metadata does Field have?

```
name :: Field -> Maybe ByteString
tableOid :: Field -> Maybe Oid
tableColumn :: Field -> Int
format :: Field -> Format
typeOid :: Field -> Oid
-- and more
```

How does one write a (Conversion a) action?

Good question! The answer is that we (the authors of this tutorial) don't know! And we didn't feel the need to find out as well. Because you already have the `fromField` functions for a lot of pre-defined Haskell types. In practice, you usually compose them to obtain your desired `Conversion` action. Read the other sections in this chapter to find exampler of how to do this.

QueryRunnerColumnDefault

This typeclass is used by Opaleye to do the conversion from postgres types defined by Opaleye, into Haskell types. It is defined as

```
class QueryRunnerColumnDefault pgType haskellType where
  queryRunnerColumnDefault :: QueryRunnerColumn pgType haskellType
```

Opaleye provides with a function

```
fieldQueryRunnerColumn:: FromField haskell => QueryRunnerColumn pgType haskell
```

As the type signature shows, *fieldQueryRunnerColumn* can return a value of type *QueryRunnerColumn a b* as long as *b* is an instance of *FromField* typeclass. So once we define an instance of FromField for our type, all we have to do is the following.

For the data type *TenantStatus* that we saw earlier,

```
instance QueryRunnerColumnDefault PGText TenantStatus where
  queryRunnerColumnDefault = fieldQueryRunnerColumn
```

Default

Note: This is **not** the `Data.Default` that you *may* be familiar with. This is `Data.Profunctor.Product.Default`

This is a typeclass that Opaleye uses to convert Haskell values to Postgres values while writing to the database. It is defined as:

```
class Default (p :: * -> * -> *) a b where
  def :: p a b
```

You see a type variable `p`, that this definition required. Opaleye provided with a type *Constant* that can be used here. It is defined as

```
newtype Constant haskells columns
  = Constant {constantExplicit :: haskells -> columns}
```

So if we are defining a Default instance for the *TenantStatus* we saw earlier, it would be something like this.

```
instance Default Constant TenantStatus (Column PGText) where
  def = Constant def'
    where
      def' :: TenantStatus -> (Column PGText)
      def' TenantStatusActive = pgStrictText "active"
      def' TenantStatusInActive = pgStrictText "inactive"
      def' TenantStatusNew = pgStrictText "new"
```

Newtypes for primary keys

Ideally, we would like to represent our primary keys using newtypes that wrap around an `Int`. For example:

```
newtype TenantId = TenantId Int
newtype ProductId = ProductId Int
```

This is generally done to extract greater type-safety out of the system. For instance, doing this would prevent the following class of errors:

- Comparing a `TenantId` to a `ProductId`, which would rarely make sense.
- Passing a `TenantId` to a function which is expecting a `ProductId`
- At an SQL level, joining the `tenantTable` with the `productTable` by matching `tenants.id` to `products.id`

But it seems that Opaleye's support for this feature is not really ready. So we will skip it for now.

Mapping ENUMs to Haskell ADTs

Here's what our ADT for `TenantStatus` looks like:

```
data TenantStatus = TenantStatusActive | TenantStatusInActive | TenantStatusNew
  deriving (Show)
```

Here's how we would setup the DB => Haskell conversion. If you notice, we didn't really need to bother with how to build `Conversion TenantStatus` because once we know what the incoming ByteString is, we know exactly which ADT value it should map to. We simply `return` that value, since `Conversion` is a Monad.

```
instance FromField TenantStatus where
  fromField field mb_bytestring = makeTenantStatus mb_bytestring
    where
    makeTenantStatus :: Maybe ByteString -> Conversion TenantStatus
    makeTenantStatus (Just "active") = return TenantStatusActive
    makeTenantStatus (Just "inactive") = return TenantStatusInActive
    makeTenantStatus (Just "new") = return TenantStatusNew
    makeTenantStatus (Just _) = returnError ConversionFailed field "Unrecognized␣
↪tenant status"
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty tenant status"

instance QueryRunnerColumnDefault PGText TenantStatus where
  queryRunnerColumnDefault = fieldQueryRunnerColumn
```

TODO: As we saw in the Typeclasses section, Opaleye requires the QueryRunnerColumnDefault typeclass instances for converting from data read from Database to Haskell values. the function *fieldQueryRunnerColumn* can return the value of the required type as long as there is a FromField instance for the required type.

Now, let's look at how to setup the Haskell => DB conversion.

```
instance Default Constant TenantStatus (Column PGText) where
  def = Constant def'
    where
      def' :: TenantStatus -> (Column PGText)
      def' TenantStatusActive = pgStrictText "active"
      def' TenantStatusInActive = pgStrictText "inactive"
      def' TenantStatusNew = pgStrictText "new"
```

Handing Postgres Arrays

Postgresql Array column are represented by the PGArray type. It can take an additional type to represent the kind of the array. So if the column is `text[]`, the type needs to be `PGArray PGText`.

If you look at the earlier code, you can see that the output contains a list for the `tag` fields.

Handling JSONB

The type that represents `jsonb` postgresql columns in Opaleye is `PGJsonb`. It will support any type that has a ToJSON/FromJSON instances defined for it.

ToJSON/FromJSON typeclasses are exported by the Aeson json library.

This is how it is done. Let us change the *properties* field of the *Product* type we saw earlier into a record in see how we can store it in a jsonb field.

```haskell
{-# LANGUAGE Arrows                #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings     #-}
{-# LANGUAGE TemplateHaskell       #-}

module Main where

import            Data.Aeson
import            Data.Aeson.Types
import            Data.Profunctor.Product
import            Data.Profunctor.Product.Default
import            Data.Profunctor.Product.TH          (makeAdaptorAndInstance)
import            Data.Scientific
import            Data.ByteString hiding (putStrLn)
import            Data.Text
import            Data.Time
import            Opaleye

import            Database.PostgreSQL.Simple
import            Database.PostgreSQL.Simple.FromField (Conversion,
                                                       FromField (..),
                                                       ResultError (..),
                                                       returnError)

import            Control.Arrow
import            Prelude                             hiding (id)


readOnly :: String -> TableProperties () (Column a)
readOnly = lmap (const Nothing) . optional

-- Tenant stuff

newtype TenantId = TenantId Int deriving(Show)

data TenantStatus = TenantStatusActive | TenantStatusInActive | TenantStatusNew
  deriving (Show)

data TenantPoly key name fname lname email phone status b_domain = Tenant
  { tenant_id               :: key
  , tenant_name             :: name
  , tenant_firstname        :: fname
  , tenant_lastname         :: lname
  , tenant_email            :: email
  , tenant_phone            :: phone
  , tenant_status           :: status
  , tenant_backofficedomain :: b_domain
  } deriving (Show)

type Tenant = TenantPoly TenantId Text Text Text Text Text TenantStatus Text

type TenantTableW = TenantPoly
  (Maybe (Column PGInt4))
  (Column PGText)
  (Column PGText)
```

```haskell
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)

type TenantTableR = TenantPoly
  (Column PGInt4)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)

-- Product stuff

newtype ProductId = ProductId Int deriving (Show)

data ProductType = ProductPhysical | ProductDigital deriving (Show)

data ProductProperties = ProductProperties { product_color :: String, product_weight :: String} deriving (Show)

data ProductPoly id created_at updated_at tenant_id name description url_slug tags currency advertised_price comparison_price cost_price product_type is_published properties = Product {
      product_id                :: id
    , product_created_at        :: created_at
    , product_updated_at        :: updated_at
    , product_tenant_id         :: tenant_id
    , product_name              :: name
    , product_description       :: description
    , product_url_slug          :: url_slug
    , product_tags              :: tags
    , product_currency          :: currency
    , product_advertised_price  :: advertised_price
    , product_comparison_price  :: comparison_price
    , product_cost_price        :: cost_price
    , product_product_type      :: product_type
    , product_is_published      :: is_published
    , product_properties        :: properties
  } deriving (Show)

type Product = ProductPoly ProductId UTCTime UTCTime TenantId Text (Maybe Text) Text [Text] Text Scientific Scientific (Maybe Scientific) ProductType Bool ProductProperties
type ProductTableW = ProductPoly
  (Maybe (Column PGInt4))
  (Maybe (Column PGTimestamptz))
  (Maybe (Column PGTimestamptz))
  (Column PGInt4)
  (Column PGText)
  (Maybe (Column (Nullable PGText)))
  (Column PGText)
  (Column (PGArray PGText))
  (Column PGText)
```

```haskell
  (Column PGFloat8)
  (Column PGFloat8)
  (Maybe (Column (Nullable PGFloat8)))
  (Column PGText)
  (Column PGBool)
  (Column PGJsonb)

type ProductTableR = ProductPoly
  (Column PGInt4)
  (Column PGTimestamptz)
  (Column PGTimestamptz)
  (Column PGInt4)
  (Column PGText)
  (Column (Nullable PGText))
  (Column PGText)
  (Column (PGArray PGText))
  (Column PGText)
  (Column PGFloat8)
  (Column PGFloat8)
  (Column (Nullable PGFloat8))
  (Column PGText)
  (Column PGBool)
  (Column PGJsonb)

-- Table defs

$(makeAdaptorAndInstance "pTenant" ''TenantPoly)
tenantTable :: Table TenantTableW TenantTableR
tenantTable = Table "tenants" (pTenant
   Tenant {
     tenant_id = (optional "id"),
     tenant_name = (required "name"),
     tenant_firstname = (required "first_name"),
     tenant_lastname = (required "last_name"),
     tenant_email = (required "email"),
     tenant_phone = (required "phone"),
     tenant_status = (required "status"),
     tenant_backofficedomain = (required "backoffice_domain")
   }
 )

$(makeAdaptorAndInstance "pProduct" ''ProductPoly)

productTable :: Table ProductTableW ProductTableR
productTable = Table "products" (pProduct
    Product {
      product_id = (optional "id"),
      product_created_at = (optional "created_at"),
      product_updated_at = (optional "updated_at"),
      product_tenant_id = (required "tenant_id"),
      product_name = (required "name"),
      product_description = (optional "description"),
      product_url_slug = (required "url_slug"),
      product_tags = (required "tags"),
      product_currency = (required "currency"),
      product_advertised_price = (required "advertised_price"),
      product_comparison_price = (required "comparison_price"),
      product_cost_price = (optional "cost_price"),
```

```haskell
      product_product_type = (required "type"),
      product_is_published = (required "is_published"),
      product_properties = (required "properties") })

-- Instance declarations for custom types
-- For TenantStatus

instance FromField TenantStatus where
  fromField field mb_bytestring = makeTenantStatus mb_bytestring
    where
    makeTenantStatus :: Maybe ByteString -> Conversion TenantStatus
    makeTenantStatus (Just "active") = return TenantStatusActive
    makeTenantStatus (Just "inactive") = return TenantStatusInActive
    makeTenantStatus (Just "new") = return TenantStatusNew
    makeTenantStatus (Just _) = returnError ConversionFailed field "Unrecognized tenant status"
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty tenant status"

instance QueryRunnerColumnDefault PGText TenantStatus where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For ProductType

instance FromField ProductType where
  fromField field mb_bytestring = makeProductType mb_bytestring
    where
    makeProductType :: Maybe ByteString -> Conversion ProductType
    makeProductType (Just "physical") = return ProductPhysical
    makeProductType (Just "digital") = return ProductDigital
    makeProductType (Just _) = returnError ConversionFailed field "Unrecognized product type"
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty product type"

instance QueryRunnerColumnDefault PGText ProductType where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For productId

instance FromField ProductId where
  fromField field mb_bytestring = ProductId <$> fromField field mb_bytestring

instance QueryRunnerColumnDefault PGInt4 ProductId where
  queryRunnerColumnDefault = fieldQueryRunnerColumn
-- For TenantId
instance FromField TenantId where
  fromField field mb_bytestring = TenantId <$> fromField field mb_bytestring

instance QueryRunnerColumnDefault PGInt4 TenantId where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For Scientific we didn't have to implement instance of fromField
-- because it is already defined in postgresql-simple

instance QueryRunnerColumnDefault PGFloat8 Scientific where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- Default instance definitions for custom datatypes for converison to
-- PG types while writing into tables
```

```haskell

-- For Tenant stuff

instance Default Constant TenantStatus (Column PGText) where
  def = Constant def'
    where
      def' :: TenantStatus -> (Column PGText)
      def' TenantStatusActive = pgStrictText "active"
      def' TenantStatusInActive = pgStrictText "inactive"
      def' TenantStatusNew = pgStrictText "new"

instance Default Constant TenantId (Maybe (Column PGInt4)) where
  def = Constant (\(TenantId x) -> Just $ pgInt4 x)

-- For Product stuff

instance Default Constant ProductType (Column PGText) where
  def = Constant def'
    where
      def' :: ProductType -> (Column PGText)
      def' ProductDigital = pgStrictText "digital"
      def' ProductPhysical = pgStrictText "physical"

instance Default Constant ProductId (Maybe (Column PGInt4)) where
  def = Constant (\(ProductId x) -> Just $ constant x)

instance Default Constant Scientific (Column PGFloat8) where
  def = Constant (pgDouble.toRealFloat)

instance Default Constant Scientific (Column (Nullable PGFloat8)) where
  def = Constant (toNullable.constant)

instance Default Constant Text (Column (Nullable PGText)) where
  def = Constant (toNullable.pgStrictText)

instance Default Constant UTCTime (Maybe (Column PGTimestamptz)) where
  def = Constant ((Just).pgUTCTime)

instance Default Constant TenantId (Column PGInt4) where
  def = Constant (\(TenantId x) -> constant x)

-- FromJSON/ToJSON instances for properties

instance FromJSON ProductProperties where
  parseJSON (Object v) = ProductProperties <$> v .: "color" <*> v .: "weight"
  parseJSON invalid = typeMismatch "Unrecognized format for product properties"␣
↪invalid

instance ToJSON ProductProperties where
  toJSON ProductProperties {product_color = color, product_weight = weight} = object [
↪"color" .= color, "weight" .= weight]

instance FromField ProductProperties where
  fromField field mb = do
    v <-  fromField field mb
    valueToProductProperties v
    where
      valueToProductProperties :: Value -> Conversion ProductProperties
```

```
    valueToProductProperties v = case fromJSON v of
      Success a -> return a
      Error err -> returnError ConversionFailed field "Cannot parse product properties"

instance QueryRunnerColumnDefault PGJsonb ProductProperties where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

instance Default Constant ProductProperties (Column PGJsonb) where
  def = Constant (\pp -> pgValueJSONB $ toJSON pp)

getProducts :: IO [Product]
getProducts = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runQuery conn $ queryTable productTable

getTenants :: IO [Tenant]
getTenants = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runQuery conn $ queryTable tenantTable

insertTenant :: IO ()
insertTenant = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runInsertManyReturning conn tenantTable [constant getTestTenant] (\x -> x) :: IO [Tenant]
  return ()

insertProduct :: IO ()
insertProduct = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  product <- getTestProduct
  runInsertManyReturning conn productTable [constant product] (\x -> x) :: IO [Product]
  return ()

getTestTenant :: Tenant
getTestTenant = Tenant (TenantId 5) "Tenant Bob" "Bobby" "Bob" "bob@mail.com" "2255" TenantStatusInActive "bob.com"

getTestProduct :: IO Product
getTestProduct = do
  time <- getCurrentTime
  let properties =  ProductProperties { product_color = "red", product_weight = "200gm"}
  return $ Product (ProductId 5) time time (TenantId 5) "snacks" (Just "") "" ["tag1", "tag2"] "INR" 30 45 Nothing ProductPhysical False properties

main :: IO ()
main = do
  insertTenant
  insertProduct
  tenants <- getTenants
  products <- getProducts
  putStrLn $ show tenants
  putStrLn $ show products

-- Output
```

```
--
--
-- [Tenant {tenant_id = TenantId 1, tenant_name = "Tenant John", tenant_firstname = "John", tenant_lastname = "Honai", te
-- nant_email = "john@mail.com", tenant_phone = "2255", tenant_status = TenantStatusInActive, tenant_backofficedomain = "
-- jhonhonai.com"},Tenant {tenant_id = TenantId 5, tenant_name = "Tenant Bob", tenant_firstname = "Bobby", tenant_lastnam
-- e = "Bob", tenant_email = "bob@mail.com", tenant_phone = "2255", tenant_status = TenantStatusInActive, tenant_backoffi
-- cedomain = "bob.com"}]
-- [Product {product_id = ProductId 5, product_created_at = 2016-11-28 12:31:40.085634 UTC, product_updated_at = 2016-11-
-- 28 12:31:40.085634 UTC, product_tenant_id = TenantId 5, product_name = "snacks", product_description = Just "", produc
-- t_url_slug = "", product_tags = ["tag1","tag2"], product_currency = "INR", product_advertised_price = 30.0, product_co
-- mparison_price = 45.0, product_cost_price = Nothing, product_product_type = ProductPhysical, product_is_published = Fa
-- lse, product_properties = ProductProperties {product_color = "red", product_weight = "200gm"}}]
```

In the emphasized lines in code above, we are defining instances to support json conversion. The binary operators *.:* and .= that you see are stuff exported by the Aeson json library. The basis of Json decoding/encoding is the aeson's Value type. This type can represent any json value. It is defined as

```
data Value
  = Object !Object
  | Array !Array
  | String !Text
  | Number !Scientific
  | Bool !Bool
  | Null
```

The Object type is an alias for a HashMap, and Array for a Vector and so on.

The instances are our usual type conversion instances. The *Value* type has the instances built in, so we will use them for defining instances for ProductProperties. So when we define a *FromField* instance for ProductProperties, we use the fromField instance of the *Value* type. We are also handling errors that might occur while parsing and reporting via postgresql's error reporting functions.

In the last instance, we are using the Default instance of the aforementioned *Value* type to implement instance for *ProductProperties*. The toJSON converts our ProductProperties to *Value* type, and since there are already built in Default instance for *Value* type, we were able to call the *constant* function on it, to return the appropriate opaleye's column type.

Making columns read-only

Sometimes we will want to make a certain column read only, accepting only values generated from the database. Here is how we can do it.

We have to define a new function *readOnly*, which will make the required field of type (), in the write types so we won't be able to provide a value for writing.

```
{-# LANGUAGE Arrows                #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE MultiParamTypeClasses #-}
```

```haskell
{-# LANGUAGE OverloadedStrings      #-}
{-# LANGUAGE TemplateHaskell        #-}

module Main where

import           Data.Aeson
import           Data.Aeson.Types
import           Data.Profunctor
import           Data.Profunctor.Product
import           Data.Profunctor.Product.Default
import           Data.Profunctor.Product.TH          (makeAdaptorAndInstance)
import           Data.Scientific
import           Data.ByteString hiding (putStrLn)
import           Data.Text
import           Data.Time
import           Opaleye

import           Database.PostgreSQL.Simple
import           Database.PostgreSQL.Simple.FromField (Conversion,
                                                       FromField (..),
                                                       ResultError (..),
                                                       returnError)

import           Control.Arrow
import           Prelude                             hiding (id)


readOnly :: String -> TableProperties () (Column a)
readOnly = lmap (const Nothing) . optional

-- Tenant stuff

newtype TenantId = TenantId Int deriving(Show)

data TenantStatus = TenantStatusActive | TenantStatusInActive | TenantStatusNew
  deriving (Show)

data TenantPoly key name fname lname email phone status b_domain = Tenant
  { tenant_id               :: key
  , tenant_name             :: name
  , tenant_firstname        :: fname
  , tenant_lastname         :: lname
  , tenant_email            :: email
  , tenant_phone            :: phone
  , tenant_status           :: status
  , tenant_backofficedomain :: b_domain
  } deriving (Show)

type Tenant = TenantPoly TenantId Text Text Text Text Text TenantStatus Text
type TenantIncoming = TenantPoly () Text Text Text Text Text TenantStatus Text

type TenantTableW = TenantPoly
  ()
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
```

```
  (Column PGText)
  (Column PGText)

type TenantTableR = TenantPoly
  (Column PGInt4)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)
  (Column PGText)

-- Product stuff

newtype ProductId = ProductId Int deriving (Show)

data ProductType = ProductPhysical | ProductDigital deriving (Show)

data ProductProperties = ProductProperties { product_color :: String, product_weight :: String} deriving (Show)

data ProductPoly id created_at updated_at tenant_id name description url_slug tags currency advertised_price comparison_price cost_price product_type is_published properties = Product {
      product_id                 :: id
    , product_created_at         :: created_at
    , product_updated_at         :: updated_at
    , product_tenant_id          :: tenant_id
    , product_name               :: name
    , product_description        :: description
    , product_url_slug           :: url_slug
    , product_tags               :: tags
    , product_currency           :: currency
    , product_advertised_price   :: advertised_price
    , product_comparison_price   :: comparison_price
    , product_cost_price         :: cost_price
    , product_product_type       :: product_type
    , product_is_published       :: is_published
    , product_properties         :: properties
  } deriving (Show)

type Product = ProductPoly ProductId UTCTime UTCTime TenantId Text (Maybe Text) Text [Text] Text Scientific Scientific (Maybe Scientific) ProductType Bool ProductProperties
type ProductTableW = ProductPoly
  (Maybe (Column PGInt4))
  (Maybe (Column PGTimestamptz))
  (Maybe (Column PGTimestamptz))
  (Column PGInt4)
  (Column PGText)
  (Maybe (Column (Nullable PGText)))
  (Column PGText)
  (Column (PGArray PGText))
  (Column PGText)
  (Column PGFloat8)
  (Column PGFloat8)
  (Maybe (Column (Nullable PGFloat8)))
```

```
  (Column PGText)
  (Column PGBool)
  (Column PGJsonb)

type ProductTableR = ProductPoly
  (Column PGInt4)
  (Column PGTimestamptz)
  (Column PGTimestamptz)
  (Column PGInt4)
  (Column PGText)
  (Column (Nullable PGText))
  (Column PGText)
  (Column (PGArray PGText))
  (Column PGText)
  (Column PGFloat8)
  (Column PGFloat8)
  (Column (Nullable PGFloat8))
  (Column PGText)
  (Column PGBool)
  (Column PGJsonb)

-- Table defs

$(makeAdaptorAndInstance "pTenant" ''TenantPoly)
tenantTable :: Table TenantTableW TenantTableR
tenantTable = Table "tenants" (pTenant
   Tenant {
     tenant_id = (readOnly "id"),
     tenant_name = (required "name"),
     tenant_firstname = (required "first_name"),
     tenant_lastname = (required "last_name"),
     tenant_email = (required "email"),
     tenant_phone = (required "phone"),
     tenant_status = (required "status"),
     tenant_backofficedomain = (required "backoffice_domain")
   }
 )

$(makeAdaptorAndInstance "pProduct" ''ProductPoly)

productTable :: Table ProductTableW ProductTableR
productTable = Table "products" (pProduct
    Product {
      product_id = (optional "id"),
      product_created_at = (optional "created_at"),
      product_updated_at = (optional "updated_at"),
      product_tenant_id = (required "tenant_id"),
      product_name = (required "name"),
      product_description = (optional "description"),
      product_url_slug = (required "url_slug"),
      product_tags = (required "tags"),
      product_currency = (required "currency"),
      product_advertised_price = (required "advertised_price"),
      product_comparison_price = (required "comparison_price"),
      product_cost_price = (optional "cost_price"),
      product_product_type = (required "type"),
      product_is_published = (required "is_published"),
      product_properties = (required "properties") })
```

```haskell

-- Instance declarations for custom types
-- For TenantStatus

instance FromField TenantStatus where
  fromField field mb_bytestring = makeTenantStatus mb_bytestring
    where
    makeTenantStatus :: Maybe ByteString -> Conversion TenantStatus
    makeTenantStatus (Just "active") = return TenantStatusActive
    makeTenantStatus (Just "inactive") = return TenantStatusInActive
    makeTenantStatus (Just "new") = return TenantStatusNew
    makeTenantStatus (Just _) = returnError ConversionFailed field "Unrecognized tenant status"
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty tenant status"

instance QueryRunnerColumnDefault PGText TenantStatus where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For ProductType

instance FromField ProductType where
  fromField field mb_bytestring = makeProductType mb_bytestring
    where
    makeProductType :: Maybe ByteString -> Conversion ProductType
    makeProductType (Just "physical") = return ProductPhysical
    makeProductType (Just "digital") = return ProductDigital
    makeProductType (Just _) = returnError ConversionFailed field "Unrecognized product type"
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty product type"

instance QueryRunnerColumnDefault PGText ProductType where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For productId

instance FromField ProductId where
  fromField field mb_bytestring = ProductId <$> fromField field mb_bytestring

instance QueryRunnerColumnDefault PGInt4 ProductId where
  queryRunnerColumnDefault = fieldQueryRunnerColumn
-- For TenantId
instance FromField TenantId where
  fromField field mb_bytestring = TenantId <$> fromField field mb_bytestring

instance QueryRunnerColumnDefault PGInt4 TenantId where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- For Scientific we didn't have to implement instance of fromField
-- because it is already defined in postgresql-simple

instance QueryRunnerColumnDefault PGFloat8 Scientific where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

-- Default instance definitions for custom datatypes for converison to
-- PG types while writing into tables

-- For Tenant stuff

```

```
instance Default Constant TenantStatus (Column PGText) where
  def = Constant def'
    where
      def' :: TenantStatus -> (Column PGText)
      def' TenantStatusActive = pgStrictText "active"
      def' TenantStatusInActive = pgStrictText "inactive"
      def' TenantStatusNew = pgStrictText "new"

instance Default Constant TenantId (Maybe (Column PGInt4)) where
  def = Constant (\(TenantId x) -> Just $ pgInt4 x)

-- For Product stuff

instance Default Constant ProductType (Column PGText) where
  def = Constant def'
    where
      def' :: ProductType -> (Column PGText)
      def' ProductDigital = pgStrictText "digital"
      def' ProductPhysical = pgStrictText "physical"

instance Default Constant ProductId (Maybe (Column PGInt4)) where
  def = Constant (\(ProductId x) -> Just $ constant x)

instance Default Constant Scientific (Column PGFloat8) where
  def = Constant (pgDouble.toRealFloat)

instance Default Constant Scientific (Column (Nullable PGFloat8)) where
  def = Constant (toNullable.constant)

instance Default Constant Text (Column (Nullable PGText)) where
  def = Constant (toNullable.pgStrictText)

instance Default Constant UTCTime (Maybe (Column PGTimestamptz)) where
  def = Constant ((Just).pgUTCTime)

instance Default Constant TenantId (Column PGInt4) where
  def = Constant (\(TenantId x) -> constant x)

-- FromJSON/ToJSON instances for properties

instance FromJSON ProductProperties where
  parseJSON (Object v) = ProductProperties <$> v .: "color" <*> v .: "weight"
  parseJSON invalid = typeMismatch "Unrecognized format for product properties" invalid

instance ToJSON ProductProperties where
  toJSON ProductProperties {product_color = color, product_weight = weight} = object ["color" .= color, "weight" .= weight]

instance FromField ProductProperties where
  fromField field mb = do
    v <-  fromField field mb
    valueToProductProperties v
    where
      valueToProductProperties :: Value -> Conversion ProductProperties
      valueToProductProperties v = case fromJSON v of
        Success a -> return a
        Error err -> returnError ConversionFailed field "Cannot parse product properties"
```

```haskell

instance QueryRunnerColumnDefault PGJsonb ProductProperties where
  queryRunnerColumnDefault = fieldQueryRunnerColumn

instance Default Constant ProductProperties (Column PGJsonb) where
  def = Constant (\pp -> pgValueJSONB $ toJSON pp)

getProducts :: IO [Product]
getProducts = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runQuery conn $ queryTable productTable

getTenants :: IO [Tenant]
getTenants = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runQuery conn $ queryTable tenantTable

insertTenant :: IO ()
insertTenant = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  runInsertManyReturning conn tenantTable [constant getTestTenant] (\x -> x) :: IO [Tenant]
  return ()

insertProduct :: IO ()
insertProduct = do
  conn <- connect defaultConnectInfo { connectDatabase = "scratch"}
  product <- getTestProduct
  runInsertManyReturning conn productTable [constant product] (\x -> x) :: IO [Product]
  return ()

getTestTenant :: TenantIncoming
getTestTenant = Tenant {
  tenant_id = (),
  tenant_name = "Tenant Bob",
  tenant_firstname = "Bobby",
  tenant_lastname = "Bob",
  tenant_email = "bob@gmail.com",
  tenant_phone = "2255",
  tenant_status = TenantStatusInActive,
  tenant_backofficedomain = "bob.com"
}

getTestProduct :: IO Product
getTestProduct = do
  time <- getCurrentTime
  let properties =  ProductProperties { product_color = "red", product_weight = "200gm"}
  return $ Product {
    product_id = (ProductId 5),
    product_created_at = time,
    product_updated_at = time,
    product_tenant_id = (TenantId 5),
    product_name = "snacks",
    product_description = Just "",
    product_url_slug = "",
    product_tags = ["tag1", "tag2"],
```

```
    product_currency = "INR",
    product_advertised_price = 30,
    product_comparison_price = 45,
    product_cost_price = Nothing,
    product_product_type = ProductPhysical,
    product_is_published = False,
    product_properties = properties
  }

main :: IO ()
main = do
  insertTenant
  insertProduct
  tenants <- getTenants
  products <- getProducts
  putStrLn $ show tenants
  putStrLn $ show products

-- Output
--
--
-- [Tenant {tenant_id = TenantId 1, tenant_name = "Tenant John", tenant_firstname =
↪"John", tenant_lastname = "Honai", te
-- nant_email = "john@mail.com", tenant_phone = "2255", tenant_status =␣
↪TenantStatusInActive, tenant_backofficedomain = "
-- jhonhonai.com"},Tenant {tenant_id = TenantId 5, tenant_name = "Tenant Bob", tenant_
↪firstname = "Bobby", tenant_lastnam
-- e = "Bob", tenant_email = "bob@mail.com", tenant_phone = "2255", tenant_status =␣
↪TenantStatusInActive, tenant_backoffi
-- cedomain = "bob.com"}]
-- [Product {product_id = ProductId 5, product_created_at = 2016-11-28 12:31:40.
↪085634 UTC, product_updated_at = 2016-11-
-- 28 12:31:40.085634 UTC, product_tenant_id = TenantId 5, product_name = "snacks",␣
↪product_description = Just "", produc
-- t_url_slug = "", product_tags = ["tag1","tag2"], product_currency = "INR", product_
↪advertised_price = 30.0, product_co
-- mparison_price = 45.0, product_cost_price = Nothing, product_product_type =␣
↪ProductPhysical, product_is_published = Fa
-- lse, product_properties = ProductProperties {product_color = "red", product_weight␣
↪= "200gm"}}]
```

The type *Conversion* is a functor, so you can define instances for custom types in terms of existing *FromField* instances. For example, if you have a type that wraps an Int, like

> data ProductId = ProductId Int

You can make a field parser instance for *ProductId* as follows

```
instance FromField ProductId where
  fromField field mb_bytestring = ProductId <$> fromField field mb_bytestring
```

While doing the above method, you have to make sure that the *FromField* instance that you are depending on can actually accept data from the underlying database column. This is relavant if you want to do this for enum types.

If you depend on the *FromField* instance of a String to read the data coming from an Enum field, it will error out because the *FromField* instance of String checks if the data is coming from a Varchar or Char field (using the first argument to the *fromField* function), and errors out if it is not.

Since the second argument to the fromField functon is a *Maybe Bytestring*, for a data type *TenantStatus* defined as

```
data TenantStatus = TenantStatusActive | TenantStatusInActive | TenantStatusNew
```

we could do the following

```
instance FromField TenantStatus where
  fromField field mb_bytestring = makeTenantStatus mb_bytestring
    where
    makeTenantStatus :: Maybe ByteString -> Conversion TenantStatus
    makeTenantStatus (Just "active") = return TenantStatusActive
    makeTenantStatus (Just "inactive") = return TenantStatusInActive
    makeTenantStatus (Just "new") = return TenantStatusNew
    makeTenantStatus (Just _) = returnError ConversionFailed field "Unrecognized␣
↪tenant status"
    makeTenantStatus Nothing = returnError UnexpectedNull field "Empty tenant status"
```

With OverloadedStrings extension enabled, we could pattern match on Bystrings using normal String literals, and return the proper value. You can also see how we are handling unexpected values or a null coming from the column.

Selecting rows

TODO

Inserting rows

SQL for table creation

We'll stick with the same `tenants` table as the previous chapter:

```
--
-- Tenants
--

create type tenant_status as enum('active', 'inactive', 'new');
create table tenants(
       id serial primary key
       ,created_at timestamp with time zone not null default current_
↪timestamp
       ,updated_at timestamp with time zone not null default current_
↪timestamp
       ,name text not null
       ,first_name text not null
       ,last_name text not null
       ,email text not null
       ,phone text not null
       ,status tenant_status not null default 'inactive'
       ,owner_id integer
       ,backoffice_domain text not null
       constraint ensure_not_null_owner_id check (status!='active' or owner_
↪id is not null)
);
create unique index idx_index_owner_id on tenants(owner_id);
create index idx_status on tenants(status);
```

```
create index idx_tenants_created_at on tenants(created_at);
create index idx_tenants_updated_at on tenants(updated_at);
create unique index idx_unique_tenants_backoffice_domain on␣
↪tenants(lower(backoffice_domain));
```

Inserting rows

TODO

- Quick example of inserting a new row into the `tenants` table using `runInsertMany`
- Explanation of the code and how it corresponds to the type-signature of `runInsertMany`

Getting the ID of a newly inserted row

TODO

- Quick example of inserting a new row into the `tenants` table and getting back the ID
- Explanation of the type-signature of `runInsertManyReturning` API call
- Showing the actual SQL queries being executed in the background

Three functions missing from the Opaleye API

TODO: Recommended functions for the following two common operations:

- Inserting a row using Haskell types as input (as against the PG type as input)
- Inserting a single row and getting back the newly inserted ID
- Inserting a single row and getting back the newly inserted row

Dealing with errors

TODO:

- What happens when an insert fails at the DB level, eg. a `CHECK CONSTRAINT` prevents insertion?
- Take the example of `idx_unique_tenants_backoffice_domain`

Using a different record-type for INSERTs

TODO

- Example of defining and using a `NewTenant` type for row creation
- Commentary on why this could be useful
- Link-off to a later section which discusses these design decisions in detail - "Designing a domain API using Opaleye"

Updating rows

SQL for table creation

We'll stick with the same `tenants` table as the previous chapter:

```
--
-- Tenants
--

create type tenant_status as enum('active', 'inactive', 'new');
create table tenants(
       id serial primary key
       ,created_at timestamp with time zone not null default current_
↪timestamp
       ,updated_at timestamp with time zone not null default current_
↪timestamp
       ,name text not null
       ,first_name text not null
       ,last_name text not null
       ,email text not null
       ,phone text not null
       ,status tenant_status not null default 'inactive'
       ,owner_id integer
       ,backoffice_domain text not null
       constraint ensure_not_null_owner_id check (status!='active' or owner_
↪id is not null)
);
create unique index idx_index_owner_id on tenants(owner_id);
create index idx_status on tenants(status);
create index idx_tenants_created_at on tenants(created_at);
create index idx_tenants_updated_at on tenants(updated_at);
create unique index idx_unique_tenants_backoffice_domain on␣
↪tenants(lower(backoffice_domain));

---
--- Products
---

create type product_type as enum('physical', 'digital');
create table products(
       id serial primary key
       ,created_at timestamp with time zone not null default current_
↪timestamp
       ,updated_at timestamp with time zone not null default current_
↪timestamp
       ,tenant_id integer not null references tenants(id)
       ,name text not null
       ,description text
       ,url_slug text not null
       ,tags text[] not null default '{}'
       ,currency char(3) not null
       ,advertised_price numeric not null
       ,comparison_price numeric not null
       ,cost_price numeric
       ,type product_type not null
       ,is_published boolean not null default false
```

```
       ,properties jsonb
);
create unique index idx_products_name on products(tenant_id, lower(name));
create unique index idx_products_url_sluf on products(tenant_id, lower(url_
↪slug));
create index idx_products_created_at on products(created_at);
create index idx_products_updated_at on products(updated_at);
create index idx_products_comparison_price on products(comparison_price);
create index idx_products_tags on products using gin(tags);
create index idx_product_type on products(type);
create index idx_product_is_published on products(is_published);
```

Updating rows

TODO

- Quick example of selecting a single row by PK, changing a field, and updating it back, using `runUpdate`
- Explanation of the code and how it corresponds to the type-signature of `runUpdate`

Getting the updated rows back from the DB

TODO

- Quick example of updating multiple rows in the `products` table and getting back the updated rows
- Explanation of the type-signature of `runUpdateReturning` API call
- Show the actual SQL queries being executed in the background

Commentary on Opaleye's update APIs

TODO:

- Opaleye forces you to update every single column in the row being updated. Why is this?

Multi-table updates (updates with JOINs)

TODO: Does Opaleye even support them? If not, what's the escape hatch?

CHAPTER 2

Reflex Tutorials

Contents:

An outline of the tutorials

This tutorial will be a progressive installment on how to write more and more complex reflex apps; Each major section will have a companion repo that you can install and use to learn the concepts we're presenting.

First Part: How to get started

Here we'll cover how to build, and minify an example app (commands, cabal flags, etc). From the code perspective, the code is slightly more complex than the one in the author's reflex tutorial, offering a first example of a more complex interaction of signals.

Companion repo: starterApp

Second Part: Client-Server structure and validations

Here we'll see how to write an application with a server and a client part, doing a simple authentication of a form.

- How to organize a project with a common part shared between backend and frontend.
- A simple server, handling the requests for authentication and using wai to gzip the js he's sending.
- Servant integration: how to treat communication with server in the reflex network (and calculate the reflex functions directly from the API specification).
- A general take on validation, showing how to mix validations on the client and on the server side.

Companion repo: mockLoginPage, corresponding to the mockup here.

Third Part: Large scale structure of the app, JSX templating

Here we'll show how to write a multi-page app complete with routing, jsx templating, hiding of signals with EventWriter, and we'll share a simple case of ffi binding.

- Descriving the problem we're solving with reflex-jsx and the solution
- Global app structuring
- Routing with servant-router and reflex-contrib-router
- An example of advanced widget creation
- EventWriter and the related advantages in the link structure
- The global interceptor-like feature
- FFI bindings
- Comments on Reflex Ecosystem

Companion repo: mockUsersRoles, corresponding to the mockup here and related.

Getting Started with Reflex

In this first installment of the reflex tour, we'll set up a stack-based infrastructure for compiling reflex programs, see some basic code, and see how we can compile and minify our app.

Quick Start

Contrary to the standard way of installing reflex, which is based on the `nix` package manager, we'll focus on a `stack` based installation. The repo for this tutorial is here.

Clone the entire repo, move to that folder and launch these installation steps:

- `stack build gtk2hs-buildtools`
- Be sure to have the required system libraries (like `webkitgtk`). If you miss some of the libraries, they will pop up as error in the next step, and you can install the missing ones
- Build with ghc: `stack build`
- Execute the desktop app: `stack exec userValidation`
- Build with ghcjs: `./deploy.sh`
- Execute the web app: `firefox js/index.html`
- TODO: check that this works on macOS

Update: Instruction for macOS, on yosemite 10.10.5

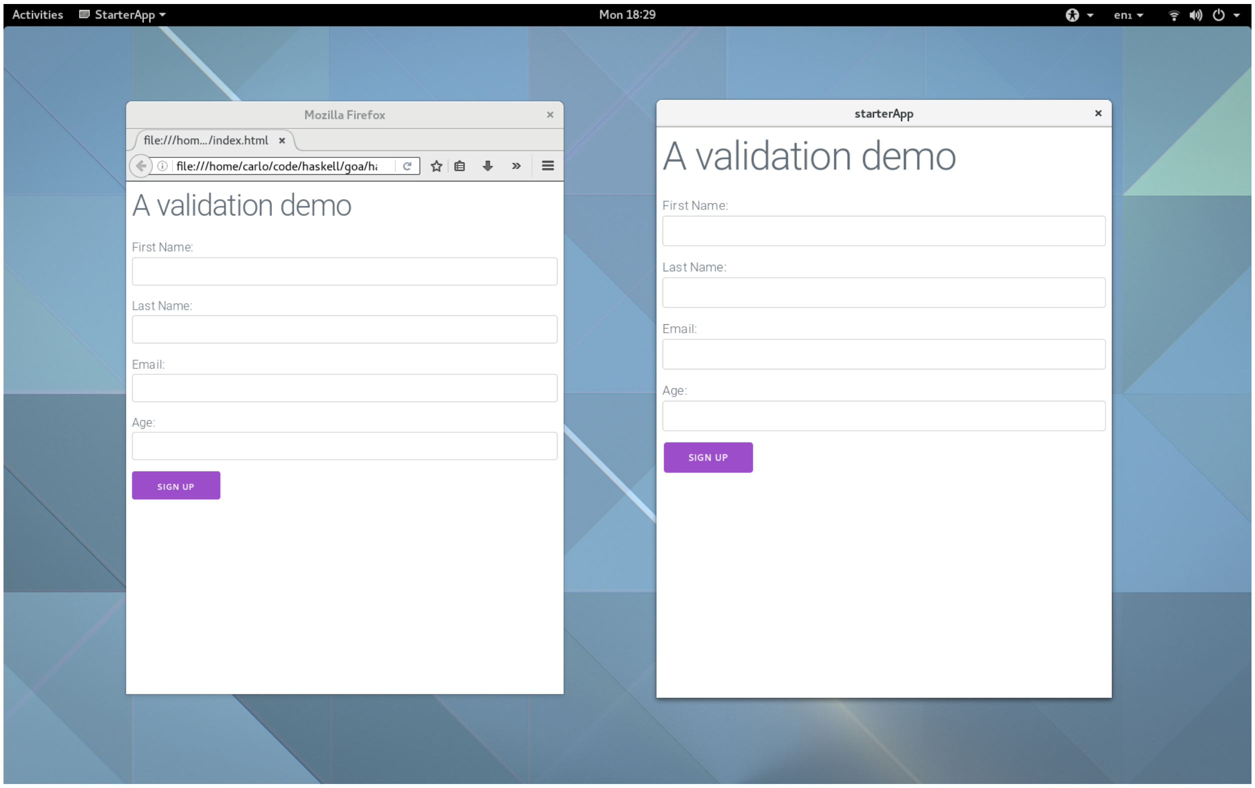
```
git clone https://github.com/vacationlabs/haskell-webapps.git
cd haskell-webapps/
cd UI/ReflexFRP/starterApp/
stack build gtk2hs-buildtools
stack setup --stack-yaml=stack-ghcjs.yaml
stack install happy
stack build --stack-yaml=stack-ghcjs.yaml
/Applications/Firefox.app/Contents/MacOS/firefox $(stack path --local-install-root --
↪stack-yaml=stack-ghcjs.yaml)/bin/starterApp.jsexe/index.html
```

While all this builds (it will be a fairly lengthy process the first time), if you are a new reflex user, be sure to check the beginners tutorial (if you want an installation process based on stack for the same code, check out here.

and the two quick-start references that will constitute most of the function we'll use in this series (for both reflex and reflex-dom).

You can see that there are two files: a stack.yaml and a stack-ghcjs.yaml. Both contain the same version of the libraries we're using, but with this setup we get a desktop app for free (built using webkit), and we're able to use tools for checking the code (like `intero` or `ghc-mod`) that don't yet directly support ghcjs.

Here below you can see the two versions of the app:



A look at the code

The first objective that this file has is to show how to deal with the fact that sometimes we don't want our values to be updated continuously: for example when designing a form, we want the feedback from the program to happen only when something happens (like, the login button is clicked, or the user navigates away from the textbox

Let's begin commenting the main function:

```
main :: IO ()
main = run 8081 $ mainWidgetWithHead htmlHead $ do
  el "h1" (text "A validation demo")
  rec firstName <- validateInput "First Name:" nameValidation  signUpButton
      lastName  <- validateInput "Last Name:"  nameValidation  signUpButton
      mail      <- validateInput "Email:"      emailValidation signUpButton
      age       <- validateInput "Age:"        ageValidation   signUpButton
      signUpButton <- button "Sign up"
  let user = (liftM4 . liftM4) User firstName lastName mail age
```

The first function we'll going to see is:

```
mainWidgetWithHead :: (forall x. Widget x ()) -> (forall x. Widget x ()) -> IO ()
```

This is the type of a `Widget`:

```
type Widget x = PostBuildT Spider
                  (ImmediateDomBuilderT Spider
                    (WithWebView x
                      (PerformEventT Spider
                        (SpiderHost Global))))
```

(it's a bit scary, but I want to introduce it here because there is an error that happens sometimes when not constraing the monad enough, and this is the key to understand that. TODO, flesh out this section)

You don't need to concern yourself with the exact meaning of this, it's just a convenient way to talk about a monadic transformer which hold the semantics together. Usually we just pass to that function an argument of type `MonadWidget t m => m ()`, as you can see from:

```
htmlHead :: MonadWidget t m => m ()
htmlHead = do
  styleSheet "https://fonts.googleapis.com/css?family=Roboto:300,300italic,700,
↪700italic"
  styleSheet "https://cdnjs.cloudflare.com/ajax/libs/milligram/1.1.0/milligram.min.css
↪"
  where
```

In which we import the css files we need from a cdn.

As you can see, the structure of the main function denotates the components of this simple app, giving a name to the return values.

Note that the `RecursiveDo` pragma lets us use the return value of the button before of his definition. It's useful to think at the main as having the following meaning: in the first pass, the widgets are constructed, and subsequently the reactive network continues the elaboration (TODO: I'm not sure to include this visualization).

The most important functions are `validateInput` and `notifyLogin`, defined below:

```
validateInput :: MonadWidget t m
              => Prompt                     -- ^ The text on the label
              -> (Text -> Either Text a)    -- ^ A pure validation function
              -> Event t b                  -- ^ An event so syncronize the update with
```

The `validateInput` function is directly responsable for the rendering of the label, using the pure function to validate the data, and change the value reported back to the caller as soon as the button is pressed.

On the other hand, the function:

```
notifyLogin :: MonadWidget t m
```

is responsible for drawing the notification for the successful login as it happens.

With these suggestions in mind, you can read directly the source code which is thoroughly commented.

Simple deployment

The ghcjs compiler by default generates some extra code dealing with `node` bindings: as we want only the webapp here, the first pass in the optimization is using the `-DGHCJS_BROWSER` option to strip the node code from the generated executable. We also use the new `-dedupe` flags that optimizes for generated size. All this is accomplished in this section of the cabal file:

```
  if impl(ghcjs)
    ghc-options:     -dedupe
    cpp-options:     -DGHCJS_BROWSER
  else
```

The next step will be using google's `closure compiler` to minify the compiles javascript, and then google's `zopfli` to gzip it; go ahead and install those tools (I just did `sudo dnf install ccjs zopfli` on fedora, but you can find the relevant instructions on their github pages).

I included a simple deployment script to show how you could compile and minify your app (I'm purposefully creating a simple bash script, there are much more things you can do, check them at ghcjs deployment page).

```
#!/usr/bin/env bash

# Compiling with ghcjs:
stack build --stack-yaml=stack-ghcjs.yaml

# Moving the generated files to the js folder:
mkdir -p js
cp -r $(stack path --local-install-root --stack-yaml=stack-ghcjs.yaml)/bin/starterApp.
↪jsexe/all.js js/

# Minifying all.js file using the closure compiler:
cd js
ccjs all.js --compilation_level=ADVANCED_OPTIMIZATIONS > all.min.js

# OPTIONAL: zipping, to see the actual transferred size of the app:
zopfli all.min.js
```

Here's the relevant output of `ls -alh js`, to show the size of the generated files:

```
-rw-r--r--. 1 carlo carlo 3.0M Dec 12 17:16 all.js
-rw-rw-r--. 1 carlo carlo 803K Dec 12 17:17 all.min.js
-rw-rw-r--. 1 carlo carlo 204K Dec 12 17:17 all.min.js.gz
```

So, the final minified and zipped app is about 204 Kb, not bad since we have to bundle the entire ghc runtime (and that's a cost that we only pay once, regardless of the size of our application).

We could also wonder if we have a size penalty from the fact that I used *classy-prelude* instead of manually importing all the required libraries. So I did an alternative benchmark, and it turns out that that's not the case:

```
-rw-r--r--. 1 carlo carlo 3.1M Dec 12 17:35 all.js
-rw-rw-r--. 1 carlo carlo 822K Dec 12 17:35 all.min.js
-rw-rw-r--. 1 carlo carlo 206K Dec 12 17:35 all.min.js.gz
```

As you can see, the difference is really minimal. In fact, all the size is probably taken up by the encoding of the ghc runtime.

A server-client architecture

In this installment of the series, we'll see:

- how to implement a client-server architecture, with a common package to share code and abstractions between the two parts.
- how to use the package `servant-reflex` to seamlessy embed server requests in the frp network.

- how to use a library to talk about data validation, of the kind done in html forms.

The code for today's repo is in: TODO

Let's begin with the simplest matter: how to share data definitions and abstractions between the backend and the frontend. It seems a very widespread practice to create three packages: one, let's say `common`, will contain the shared abstractions, and will be included by the other two, `client` (with the code for the webapp, to be compiled with ghcjs), and `server` (with the code for the server, to be compiled with ghc). That's all.

Let's also briefly describe here what this application does and the structure of the server: TODO

Validation

The requisites for validation

When designing a web app there are two kinds of validations that can be run: the first is the one done on the client, to provide validation against crude error (think of inputing a well-formed email address); the other one, usually done on the server, is about validating the data against our knowledge (think of checking if an email address is in the user database).

Sometimes, for security reasons, the server might want to do again the validations which happened in the client, and so we need way of easily composing validations, sharing the common infrastructure, so that code duplication is reduced.

Another problem that we encouter is that the format in which we report back the error to the client must be convenient enough to report errors near the UI element which caused them; for example, when validating a user with a combination of mail and password, an error message for a wrong password should be displayed near the password itself.

This brings us to discussing common solution for validation: there is the `Data.Validation` approach, in the `validation` package, which is essentially `Either` with another applicative instance. Unfortunately this approach fails us because we have no obvious way of reporting back errors to their use site.

On the other hand we have the `digestive-functors` approach, which unfortunately is geared towards a server-centric approach, and makes validations on the client difficult to write (TODO: Check the correctness of this information with Jasper).

A possible solution

So let's think about another solution: let's say I'm implementing a Mail/Password validation, so the type of my user could be

```
data User = User Mail Text
```

Now, if we expand slightly our definition to

```
data UserShape f = UserShape (f Mail) (f Text)
```

we gain the possibility of talking about a structure whose fields talk about operations or data parametrized by `Mail` and `Text`.

For example, some functor that we might want to use are `Identity` (and in fact `User` is obiously isomorphic to `UserShape Identity`), `Maybe` or `Either Text` to model the presence of errors, or for example

```
newtype Validation a = Validation { unValidationF :: Compose ((->) a) Maybe a }
```

so that:

```
UserShape Validation ~ UserShape (Mail -> Maybe Mail) (Text -> Maybe Text)
```

Now that we can talk about this "user shaped" objects, we might want to combine them, for example with something like:

```
validateUser :: User -> UserShape Validation -> UserShape Maybe
```

the `shaped` library has a generic mechanism of doing this kind of manipulations (check out the `validateRecord` function). The library uses internally `generics-sop` to construct and match the generic representations, and some Template Haskell to shield the user from the boilerplate instance declarations.

Now, we can send to the server a tentative `User` to check, and get back a `UserShape Maybe` that we can easily map back to our input text-boxes.

You can check how that's done in the client for today's installment (TODO link the correct lines).

How to query the API endpoint

The common code in this simple case contains only the definition of the user type and the type for our servant API

The server code is a simple server that serves a mock authentication. I'm not entering in an in depth discussion on the `servant` approach here (if you're interested check the wonderful servant documentation, but the gist is that you can create from a description of the api, in this project:

```
type MockApi = "auth" :> ReqBody '[JSON] User :> Post '[JSON] Text
          :<|> Raw
```

A server satisfying that api, here:

```
server :: Server MockApi
server = authenticate :<|> serveAssets :<|> serveJS
```

The package `servant-reflex` transforms a Servant API in Reflex functions for querying it, in the same way `servant-server` transforms it in a server. The invocation is very easy:

```
let url = BaseFullUrl Http "localhost" 8081 ""
(invokeAPI :<|> _ :<|> _) = client (Proxy @MockApi) (Proxy @m) (constDyn url)
```

```
client :: HasClient t m layout => Proxy layout -> Proxy m -> Dynamic t BaseUrl ->
↪Client t m layout
```

As you can see, `client` is the most important function: it takes proxies for the API and the monad in which the computation is executed (as it's customary to run a reflex computation in a (constrained) universally quantified monad, like our own `body :: MonadWidget t m => m ()` (the syntax with `@` is due to the ghc 8's `TypeApplications` extension, without it you should have written `Proxy :: Proxy MockApi` etc.)

That gives us a mean to call the relevant API endpoint (TODO: detail the type of the transformed function, detailing how the API call is translated in events. Also talk about Xhr).

For example in our code we use this feature to like this:

CHAPTER 3

Webapp Framework

Contents:

Migrations: Creating and editing DB models

Setting up a fresh database

```
poi migrate prepare
```

This command will generate the following tables and triggers in your DB, **if they don't already exist:**

1. `schema_migrations` table to track which migrations have already been run. This is directly influenced from Rails migrations.
2. `trg_update_modified_column` - a trigger to automatically set `updated_at` column to `current_timestamp` whenever any row is updated in a table which contains this column.

Creating a new model

```
poi migrate new createUsers
```

This will create a file called `<projectRoot>/migrations/MYYYYMMDDHHmmSS-createUsers.hs` (where `YYYYMMDDHHmmSS` is the actual timestamp on which you run the command). The file will look like the following:

```
module M20170828164533_createUsers where

import Control.Monad
import Database.Rivet.V0
import Text.InterpolatedString.Perl6 (qc)

migrate :: Migration IO ()
migrate = sql up down
```

```
up = ([qc|
-- INSERT YOUR MIGRATION SQL HERE
|])

down = ([qc|
-- INSERT YOUR ROLLBACK SQL HERE
|])
```

Now edit this file to create your tables, indexes, constraints, triggers, etc. using raw SQL:

```
module M20170828164533_createUsers where

import Control.Monad
import Database.Rivet.V0
import Text.InterpolatedString.Perl6 (qc)

migrate :: Migration IO ()
migrate = sql up down


up = ([qc|
CREATE TABLE users
             (
               id serial primary key
              ,created_at timestamp with time zone not null default current_timestamp
              ,updated_at timestamp with time zone not null default current_timestamp
              ,username text not null
              ,password text not null
              ,first_name text
              ,last_name text
              ,status user_status not null default 'inactive'
              CONSTRAINT chk_status CHECK ((status IN ('active', 'inactive', 'deleted
↪', 'blocked')))
             );
CREATE INDEX idx_users_created_at on users(created_at);
CREATE INDEX idx_users_updated_at on users(updated_at);
CREATE INDEX idx_users_status on users(status);
CREATE UNIQUE INDEX idx_users_username on users(username);

CREATE TRIGGER trg_modify_updated_at
       BEFORE UPDATE ON users
       FOR EACH ROW EXECUTE PROCEDURE update_modified_column();
|])

down = ([qc|
DROP TABLE users;
|])
```

Tip: We should probably have our own quasi-quoter called `sql` or something, which allows mixing of raw SQL along with custom helper functions. We can write helper functions to generated indexes, triggers for audit logs, triggers for updating `updated_at`, triggers for pushing to DB based `event_log`, etc.

Now, run the migration, with the following command:

```
poi migrate up
```

Here is what this will do, under the hood:

1. This will connect to the development database (by default) and execute all pending migrations. The timestamp/version of all migrations in the `<projectRoot>/migrations/` directory will be looked-up in the `schema_migrations` table. Any migration which is not there in the table will be executed in ascending order of the timestamp/version.
2. Each individual migration will be wrapped within a **single BEGIN/COMMIT** block - which means that if any migration throws an error:
   (a) that particular migration will be rolled back,
   (b) all previous migrations (which have already executed successful) will persist,
   (c) and all migrations which are yet to be executed, will be aborted.
3. Once the migration runs successfully, it will run the model code-generator under the hood, to create/modify/delete any model files that need to be updated as a result of this migration.

Editing existing models

The worlflow remains pretty much the same as "Creating a new model":

1. Create a migration file
2. Write a bunch of `ALTER` statements in the migration
3. Run `poi migrate up`

Other useful command-line arguments

```
poi migrate [ up | down | redo | prepare | new ]

--env environmentName

       Explicitly pass an environment to the script. Default value is
      `development` or the value of the `APP_ENV` environment variable (in
      that order)

--version regex

       Pass a specific migration version to the script. A fuzzy (or regex)
       match will be attempted with the given argument. If exactly one
       migration matches, it will be targeted, else all matching migrations
       will be printed out STDOUT.
```

Basic CRUD Operations with models

Model code-generator

Once you've generated your models using *the migration tool* you'll notice a lot of files getting auto-generated in the `<projectRoot>/autogen` & `<projectRoot>/src/Models` directories:

1. For every table that your DB contains you'll have an auto-generated DB interface called `AutoGenerated.Models.<SingularizedTableNameInCamelCase>`.
2. For every table that has a primary key called `id` (which is a recommended convention), you'll have an auto-generated module called `AutoGenerated.PrimaryKeys.<SingularizedTableNameInCamelCase>Id`
3. For every **unique** column-name, across all your tables, you'll have an auto-generated lens-class called `AutoGenerated.Classes.Has<FieldNameInCamelCase>`
4. For every model that is **newly generated**, you'll have a file called `Models.<SingularizedTableNameInCamelCase>` and a file called `Models.<SingularizedTableNameInCamelCase>.Types`

For example, if you have the following two tables in your DB schema...

users	contacts
id	id
created_at	created_at
updated_at	updated_at
email	email
password	first_name
first_name	last_name
last_name	street_address
	state
	country
	zip
	user_id references users(id)

...you'll end up with the following files:

Filename	Purpose	Overwritten?
autogen/AutoGenerated/Models/User.hs	Auto-generated DB interface	Yes
autogen/AutoGenerated/Models/Contact.hs	Auto-generated DB interface	Yes
autogen/AutoGenerated/PrimaryKeys/UserId.hs	newtype for PK	Yes
autogen/AutoGenerated/PrimaryKeys/ContactId.hs	newtype for PK	Yes
autogen/AutoGenerated/Classes/Id.hs	Lens class	Yes
autogen/AutoGenerated/Classes/Id.hs	Lens class	Yes
autogen/AutoGenerated/Classes/CreatedAt.hs	Lens class	Yes
autogen/AutoGenerated/Classes/UpdatedAt.hs	Lens class	Yes
autogen/AutoGenerated/Classes/Email.hs	Lens class	Yes
autogen/AutoGenerated/Classes/Password.hs	Lens class	Yes
autogen/AutoGenerated/Classes/FirstName.hs	Lens class	Yes
autogen/AutoGenerated/Classes/LastName.hs	Lens class	Yes
autogen/AutoGenerated/Classes/StreetAddress.hs	Lens class	Yes
autogen/AutoGenerated/Classes/State.hs	Lens class	Yes
autogen/AutoGenerated/Classes/Country.hs	Lens class	Yes
autogen/AutoGenerated/Classes/Zip.hs	Lens class	Yes
autogen/AutoGenerated/Classes/UserId.hs	Lens class	Yes
src/Models/User.hs	Domain-level model	No
src/Models/User/Types.hs	supporting types for Models.User	No
src/Models/Contact.hs	Domain-level model	No
src/Models/Contact/Types.hs	supporting types for Models.Contact	No

Points to note

1. All files in the `<projectRoot>/autogen` directory are marked as read-only and **will be over-written** if the underlying DB schema changes. You **should not** touch these files. Simply commit them into your version control.
2. All files in `<projectRoot>/src/Models` will be **generated only once** by the code-generation tool. Once generated, they will **never be touched** by the tool. You should put all your domain logic, custom types, enumeration types, etc. in these files.

(C)reate operations on models

Try the following in your REPL:

```
createModel UserPoly
{
  _userId = Nothing
, _userCreatedAt = Nothing
, _userUpdatedAt = Nothing
, _userEmail = "saurabh@vacationlabs.com"
, _userPassword = "blahblah"
, _userFirstName = "Saurabh"
, _userLastName = "Nanda"
}
```

(R)ead operations on models

Try the following in your REPL:

```
-- finding by a primary key
findByPk (PK 1 :: UserId)

-- find a single row by matching over two columns. Will throw an error if
-- this results in multiple rows being returned.

findSingle2 tableForUser
  (
    (email, pgEq, "saurabh@vacationlabs.com")
  , (password, pgEq, "blahblah")
  )

-- find a single row by matching over three columns. Will throw an error if
-- this results in multiple rows being returned.
findSingle3 tableForUser
  (
    (email, pgEq, "saurabh@vacationlabs.com")
  , (firstName, pgEq, "Saurabh")
  , (lastName, pgEq, "Nanda")
  )

-- find the first row by matching over four columns. Will not throw an error
-- if this results in multiple rows being returned. Will silently return the
-- first row.
findFirst4 tableForUser
  (
```

```
    (email, pgEq, "saurabh@vacationlabs.com")
  , (country, pgIn, ["IN", "US"])
  , (state, pgIn, ["UP", "MH"])
  , (userId, pgEq, PK 10)
  )


-- return all matching rows
filter1 tableForUser
  (
    (email, pgEq, "saurabh@vacationlabs.com")
  )

filter2 tableForUser
  (
    (email, pgEq, "saurabh@vacationlabs.com")
  , (country, pgIn, ["IN", "US"])
  )

-- and so on, up to filter6. If you need more than 6 columns, you should
-- probably use the underlying Opaleye querying infrastructure.
```

(U)pdate operations on models

Try the following in your REPL:

```
u <- findByPk (PK 1 :: UserId)
saveModel (u & firstName .~ "new name")

-- OR

updateModel
  (PK 1 :: UserId) -- which row to update
  (\u -> (u & firstName .~ (pgStrictText "new name"))) -- updater function
```

(D)elete operations on models

Try the following in your REPL:

```
u <- findByPk (PK 1 :: UserId)
deleteModel u

-- OR

deleteModelByPk (PK 1 :: UserId)
```

General validation helpers

```
--
validateLength :: (Foldable t, Monoid e, MonadIO m) => Text -> (Int, Int) -> Getting␣
↪(t a) s (t a) -> s -> m e
```

```
-- NOTE: The type signature is probably incomplete. Please refer to the usage
-- sample to figure out what the actual type signature needs to be.
validateFormat :: (MonadIO m, Monoid e) => m RE -> Lens' s a -> s -> m e

-- Strips the field of all leading and trailing whitespace and then ensures
-- that is not a blank string. TODO: Should the whitespace-stripped string be
-- stored in the DB? How do we ensure that?
validatePresence :: (Monoid e, MonadIO m) => Text -> Getting Text s Text -> s -> m e

-- Ensures that a field is either Nothing OR a blank string (ignoring all
-- leading and trailing whitespace). TODO: How do we ensure that a blank-string
-- is actually treated as a Nothing when storing into the DB? Also, is there a
-- use-case for having a non-Maybe (i.e. NOT NULL) field, which is validated to
-- be a blank string?
validateAbsence :: (Monoid e, MonadIO m) => Text -> Getting (Maybe Text) s (Maybe␣
↪Text) -> s -> m e

-- This will end up making a DB call, because of which, more class -
-- constraints will get added. Like `Default Constant a1 (Column a1)`. Also,
-- please NOTE - you have to be careful while querying the DB for rows with the
-- same fields to NOT match the record which is being validated. This can be
-- ensured by passing another condition to `filterN` -
-- (id, pgNotEq, record ^.id)
validateUnique1 :: (Monoid e, HasDatabase m) => Text -> (Getting a1 s a1) -> s -> m e
validateUnique2 :: (Monoid e, HasDatabase m) => Text -> (Getting a1 s a1, Getting a2␣
↪s a2) -> s -> m e
validateUnique3 :: (Monoid e, HasDatabase m) => Text -> (Getting a1 s a1, Getting a2␣
↪s a2, Getting a3 s a3) -> s -> m e
-- and so on... til validateUnique5

--
validateIn :: (Monoid e, MonadIO m) => Text -> [a] -> Getting [a] s [a] -> s -> m e
```

Strict model validations

```
module Models.User
  (
    module Models.User
  , module Models.User.Types
  , module Autogenerated.Models.User
  ) where

instance DbModel User where
  strictValidations :: (MonadIO m) => User -> m [Error]
  strictValidations user =
    (validateUnique "Email must be unique" email)
    <> (validateLength "Name must be between 5 and 100 chars" (5, 100) name)
    <> (validateFormat "Doesn't seem like a valid email." (compiledRegex "(.*)@(.*)\.
↪(.*)") email)
    <> (validatePresence "Name should be present" name) -- strips the field of␣
↪whitespace
    <> (validateIn "Should be one of black or gray" ["black", "gray"] colourCode)
    <> (if (present $ user ^. firstName)
        then (validatePresence "Last name should be present if first name is given"␣
↪lastName)
```

```
        else [])
```

Deploying

Using stack with Docker

NOTE: If you are using Windows operating system, this is not yet working for Windows. Watch this issue https://github.com/commercialhaskell/stack/issues/2421

The Stack tool has built in support for executing builds inside a docker container. But first you have to set up some stuff on your machine. First of which is installing docker on your system.

https://docs.docker.com/engine/installation/

Download and install the CE (Community Edition) version. After the installation you should have a `docker` command available in your terminal.

Try the docker command `docker images` and see if works without errors. If you are getting a permission denied error, try running the following command,

> sudo usermod -a -G docker $USER

NOTE: After the above command, you should completly log out and log in to see the affect. Or if you cannot do that, just relogin as the same user, for ex, if you are loggied in as user `vl` just do a `su vl` and that should be enough.

Next you have to build the docker image that we will use for our builds. You have two options here.

1. You can either build one from using the docker file
2. You can pull a prebuilt image from the docker hub.

Building from docker file

Open up a terminal and go to the root of the app. There should be a `docker` folder there. Go to that folder, and do `docker build .` there.

```
cd docker
docker build -t vacationlabs-ubuntu .
```

When this is done, you will have a new docker image with name "vl-ubuntu-image".

Configuring Stack

Your stack.yaml will contain the following lines.

```
docker:
  env:
    - "APP_ENV=development"
  enabled: false
  image: vacationlabs-ubuntu
  run-args: ["--ulimit=nofile=60000", "--memory=4g"]
```

1. The `env` key contains a list and is used to set environment variables inside the container before the build.yaml

2. The `enabled` flag set to false to NOT use docker by default. Docker will be involved only upon specifing the command line flag `--docker`.
2. The `image` key is used to specify the docker image from which the container for the build will be made. This should already exist.
3. The `run-args` key us used to pass arguments to the docker command that created the container. Here we have used it to increase the maximum number of open files that will be allowed inside the container and the maximum amount of host memory the container is allowed to use.

Now you can build the app using the `stack build --docker`

When you do this for the first time, stack will complain there is no compiler installed in the container. Just use `--install-ghc` flag like `stack build --docker --install-ghc`. And it will install the compiler inside the container.

Stack will mount the ~/.stack folder inside the container, so installing compiler and dependencies only need to be done once. That is unless you change the image for the container.

If you find that stack gets stalled after downloading the compiler at around 90mb, you can just download the required tar archive from https://github.com/commercialhaskell/ghc/releases to the `~/.stack/programs/x86_64-linux-*` folder and name it using format `ghc-8.0.2.tar.xz` and run the build command again. That stack will use downloaded archive instead of downloading it again.

After the build, the binary file will be in the usual location.

Further reference : https://docs.haskellstack.org/en/stable/docker_integration/

Outline

1. Overall project layout - partial design:

```
projectRoot
|
|-- src
|   |
|   |-- Models
|   |   |
|   |   |-- User
|   |   |    \-- Types
|   |   |
|   |   |-- Customer
|   |   |    \-- Types
|   |   |
|   |   |-- Order
|   |   |    \-- Types
|   |   |
|   |   \-- (and so on)
|   |
|   |-- Endpoints
|   |   |
|   |   |-- User
|   |   |    \-- Types
|   |   |
|   |   |-- Customer
|   |   |    \-- Types
|   |   |
|   |   |-- Order
```

```
|   |   |    \-- Types
|   |   |
|   |   \-- (and so on)
|   |
|   \-- Foundation
|       | Import
|       | DBImport
|       \-- Types
|         |-- Currency
|         |-- PrimaryKey
|         |-- Config
|         \-- (and so on)
|
|-- app
|   \-- Main
|
|
|-- autogen
|   \-- AutoGenarated
|       |
|       |-- Models
|       |   |-- User
|       |   |-- Customer
|       |   |-- Order
|       |   \-- (and so on)
|       |
|       |-- PrimaryKeys
|       |   |-- UserId
|       |   |-- CustomerId
|       |   |-- OrderId
|       |   \-- (and so on)
|       |
|       \-- Classes (used for lenses)
|           |-- Id
|           |-- Name
|           |-- Phone
|           \-- (and so on)
|
|-- autogen-config.yml
|
\-- scripts
```

2. Models / Database
   (a) Naming conventions - almost final design
   (b) Migrations: Creating and editing models - almost final
   (c) Strict validations - WIP
   (d) Query helpers - partial design
   (e) DB transactions & savepoints - partial design
3. Creating JSON APIs - WIP
   (a) Basic JSON API - almost final
   (b) API-specific validations - WIP
   (c) File-uploads - WIP

4. Frontend/UI code
    (a) Communicating with JSON APIs - WIP
    (b) Validations - WIP
    (c) Static assets - WIP
5. Logging
    (a) File based logging - almost final
    (b) Exception/error notifications - WIP
    (c) Performance metrics in production - WIP
6. Sending emails - almost final
7. Job queues - partial design
8. Testing - WIP
9. Deployment - WIP
10. Authentication & authorization - WIP
11. Audit logs - partial design