Practical Haskell

Module 1 Lecture Slides

Class Overview

- Module 1: Databases and Persistent

Class Overview

- Module 1: Databases and Persistent
- Module 2: Web APIs, Servant

Class Overview

- Module 1: Databases and Persistent
- Module 2: Web APIs, Servant
- Module 3: Frontend Web Development with Elm

Class Overview

- Module 1: Databases and Persistent
- Module 2: Web APIs, Servant
- Module 3: Frontend Web Development with Elm
- Module 4: Monad Transformers and Free Monads

Class Overview

- Module 1: Databases and Persistent
- Module 2: Web APIs, Servant
- Module 3: Frontend Web Development with Elm
- Module 4: Monad Transformers and Free Monads
- Module 5: Testing in Haskell

Course Mechanics

- Lectures
  - Videos and Slides explaining core concepts

Course Mechanics

- Lectures
  - Videos and Slides explaining core concepts
- Screencasts
  - Video Walkthroughs of code concepts or program usage

Course Mechanics

- Lectures
  - Videos and Slides explaining core concepts
- Screencasts
  - Video Walkthroughs of code concepts or program usage
- Exercises
  - See PDF for Instructions
  - Get code from Github

Module 1 Overview

- Use Haskell to Connect to Databases

Module 1 Overview

- Use Haskell to Connect to Databases
- Persistent Library

Module 1 Overview

- Use Haskell to Connect to Databases
- Persistent Library
- Create Schema

Module 1 Overview

- Use Haskell to Connect to Databases
- Persistent Library
- Create Schema
- Write Typesafe Queries

Module 1 Overview

- Use Haskell to Connect to Databases
- Persistent Library
- Create Schema
- Write Typesafe Queries
- Serialize Types

Relational Databases

- Structured system for persisting data
- Tables with schemas
- Each table stores specific type of information
- Tables can reference each other

SQL

- Standard Query Language
- Used by many database systems
- Allows queries for certain information

Postgres

- Common RDBMS
- Uses SQL
- Must run a local server

SQLite

- Lightweight Alternative
- Stores all information in a single file
- Quick and Easy to Use

SQL Basics

- Creating a Database
- Creating a Table
- Inserting into a Table
- Selecting from a Table

Creating a Database

```
CREATE DATABASE my-database;
```

Creating a Table

```
CREATE TABLE table_name (
  column_name_1 datatype,
  column_name_2 datatype,
  ...
);

datatype <- [int, float, text, timestamptz]
```

Creating a Table

```
CREATE TABLE users (
  id int,
  name text,
  email text,
  age int
);
```

Creating a Table

```
CREATE TABLE users (
  id int NOT NULL PRIMARY KEY,
  name text,
  email text,
  age int
);
```

Inserting into a Table

```
INSERT INTO table_name (column_1, column_2, ...)
VALUES (value_1, value_2, ...);
```

Inserting into a Table

```
INSERT INTO users (id, name, email, age)
VALUES (1, 'James', 'james@test.com', 25);
```

Inserting into a Table

```
INSERT INTO users
VALUES (1, 'James', 'james@test.com', 25);
```

Selecting from a Table

```
SELECT * FROM table_name;
```

Selecting from a Table

```
SELECT (column_1, column_2) FROM table_name;
```

Selecting from a Table

```
SELECT * FROM table_name WHERE (conditions);
```

Selecting from a Table

```
SELECT * FROM users WHERE name = 'James';
```

Selecting from a Table

```
SELECT * FROM users WHERE age > 20;
```

Selecting from a Table

```
SELECT * FROM users WHERE age > 20
ORDER BY name ASC;
```

Selecting from a Table

```
SELECT * FROM users WHERE age > 20
ORDER BY name DESC;
```

SQL Schemas

- Schema gave us a structure
- Column names and types
- Can create a Haskell type to match
- But how do we keep these in sync?

Persistent Library

- Create a schema definition
- Template Haskell
- Translates into SQL
- Translates into Haskell type

Template Haskell

- Define a language within Haskell
- Translates into Haskell Code
- Sectioned off with quasi-quoter `[| ... |]`

Template Haskell Setup

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

PTH.share
  [PTH.mkPersist PTH.sqlSettings, PTH.mkMigrate “migrateAll”]
    [PTH.persistLowerCase|
    ...
    |]
```

Template Haskell Setup

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

PTH.share
  [PTH.mkPersist PTH.sqlSettings, PTH.mkMigrate “migrateAll”]
    [PTH.persistLowerCase|
    ...
    |]
```

Schema Definition

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    email Text
    age Int
    deriving Show Read Eq
|]
```

Schema Definition

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    email Text
    age Int
    deriving Show Read Eq
|]
```

Schema Definition

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    email Text
    age Int
    deriving Show Read Eq
|]
```

Schema Definition

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    email Text
    age Int
    deriving Show Read Eq
|]
```

Migrating Our Database

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

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

Migrating Our Database

```
import Database.Persist.Postgresql (runMigration)
import Schema (migrateAll)

migrateDB :: ConnectionString -> IO ()
migrateDB conn = runAction conn (runMigration migrateAll)

runAction = ...
```

Schema Definition

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    email Text
    age Int
    deriving Show Read Eq
|]
```

Generated Types

```
data User = User
  { userName :: Text
  , userEmail :: Text
  , userAge :: Int
  } deriving (Show, Read, Eq)
```

Entities

```
data Entity a = Entity (Key a) a
```

Entities

```
data Entity User = Entity (Key User) User
```

Entities

```
data Entity a = Entity (Key a) a

entityAge :: Entity User -> Int
entityAge (Entity key user) = userAge user
```

Entities

```
data Entity a = Entity (Key a) a

entityKey :: Entity a -> Key a
```

Entities

```
data Entity a = Entity (Key a) a

entityVal :: Entity a -> a
```

Keys

```
data Entity User = Entity (Int64) User
```

Keys

```
toSqlKey :: Int64 -> Key a

fromSqlKey :: Key a -> Int64
```

SQL Queries

- Insertions
- Deletions
- Retrieval
- Select

Insertions

```
INSERT INTO users (id, name, email, age)
VALUES (1, 'James', 'james@test.com', 25);
```

Insertions

```
insert :: PersistEntity val => val -> m (Key val)
```

Insertions

```
insert :: PersistEntity val => val -> m (Key val)
```

Insertions

```
insert :: val -> m (Key val)
```

Insertions

```
insert :: User -> m (Key User)
```

Insertions

```
myUser :: User
myUser = User “James” “james@test.com” 25

insertUser :: IO ()
insertUser = do
  userKey <- runSql $ insert myUser
  print $ fromSqlKey userKey
```

Deletions

```
delete :: PersistEntity val => (Key val) -> m ()
```

Deletions

```
delete :: (Key User) -> m ()
```

Deletions

```
myUser :: User
myUser = User “James” “james@test.com” 25

insertAndDeleteUser :: IO ()
insertAndDeleteUser = do
  userKey <- runSql $ insert myUser
  runSql $ delete userKey
```

Retrieval

```
get :: (Key val) -> m (Maybe val)
```

Retrieval

```
insertAndRetrieveUser :: IO ()
insertAndRetrieveUser = do
  userKey <- runSql $ insert myUser
  user <- runSql $ get userKey
  assert (user == (Just myUser))
```

Select Statements

- Much more complicated!
- Different ways to filter, order, limit

Select Statements

```
SELECT * FROM users WHERE age > 20
ORDER BY name ASC;
```

Select Statements

```
selectList :: (... val) =>
  [Filter val] ->
  [SelectOpt val] ->
  m [Entity val]
```

Select Statements

```
selectList :: (... val) =>
  [Filter val] ->
  [SelectOpt val] ->
  m [Entity val]
```

Select Statements

```
selectList :: (... val) =>
  [Filter val] ->
  [SelectOpt val] ->
  m [Entity val]
```

Select Statements

```
SELECT * FROM users;

getAllUsers :: m [Entity User]
getAllUsers = selectList [] []
```

Filters

```
SELECT * FROM users WHERE age < 25;

getYoungUsers :: m [Entity User]
getYoungUsers = selectList
  [UserAge <. 25]
  []
```

Filters

```
SELECT * FROM users WHERE age < 25;

getYoungUsers :: m [Entity User]
getYoungUsers = selectList
  [UserAge <. 25]
  []
```

Filters

```
SELECT * FROM users WHERE age < 25;

getYoungUsers :: m [Entity User]
getYoungUsers = selectList
  [UserAge <. 25]
  []
```

Filters

```
SELECT * FROM users WHERE age < 25 AND name = “James”;

getUsers :: m [Entity User]
getUsers = selectList
  [UserAge <. 25, UserName ==. “James”]
  []
```

Filters

```
SELECT * FROM users WHERE age < 25 OR name = “James”;

getUsers :: m [Entity User]
getUsers = selectList
  [OrFilter [UserAge <. 25, UserName ==. “James”]]
  []
```

Orders, Limits etc.

```
SELECT * FROM users WHERE age < 25 ORDER name ASC;
```

```
getUsers :: m [Entity User]
getUsers = selectList
  [UserAge <. 25]
  [Asc UserName]
```

Orders, Limits etc.

```
SELECT * FROM users WHERE age < 25 LIMIT 10 OFFSET 5;
```

```
getUsers :: m [Entity User]
getUsers = selectList
  [UserAge <. 25]
  [LimitTo 10, OffsetBy 5]
```

Compile Driven Development

- Learning new libraries and functions

Compile Driven Development

- Learning new libraries and functions
- Build code frequently
- Use `undefined` as a placeholder

Compile Driven Development

- Learning new libraries and functions
- Build code frequently
- Use `undefined` as a placeholder
- Learn type signatures
- Learn errors

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
????
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = undefined
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = undefined

selectList ::
  [Filter User] ->
  [SelectOpt User] ->
  SqlPersistT (LoggingT IO) [Entity User]
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = undefined

selectList :: ... -> SqlPersistT (LoggingT IO) [Entity User]

runAction
  :: ConnectionString
  -> SqlPersistT (LoggingT IO) a
  -> IO a
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = undefined
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = selectList undefined undefined
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = selectList [] []
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = selectList [userAge <. 25] []
```

Compile Driven Development

```
Couldn't match expected type `EntityField User typ0'
                  with actual type `User -> Int'
    * In the first argument of `(<.)', namely `userAge'
      In the expression: userAge <. 25
      In the first argument of `selectList', namely
`[userAge <. 25]'
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = selectList [UserAge <. 25] []
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = selectList [UserAge <. 25] [Ascending UserAge]
```

Compile Driven Development

```
Data constructor not in scope:
Ascending :: EntityField User Int -> SelectOpt User
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = selectList [UserAge <. 25] [Asc UserAge]
```

Compile Driven Development

```
fetchYoungUsers :: IO [Entity User]
fetchYoungUsers = runAction localConnString query
  where
    query :: SqlPersistT (LoggingT IO) [Entity User]
    query = selectList [UserAge <. 25] [Asc UserAge]
```

Foreign Keys

- Database types can relate to each other!
- Use **foreign keys** to relate different types
- Every `Article` will have a `User` author

Existing Schema

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    ...

  Article sql=articles
    title Text
    ...
|]
```

Hacky/Simple Approach

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    ...

  Article sql=articles
    title Text
    authorId Int64
    ...
|]
```

Hacky/Simple Approach

- Keeps SQL concepts out of types

Hacky/Simple Approach

- Keeps SQL concepts out of types
- But we lose out on generated SQL!

Better Approach

```
[PTH.persistLowerCase|
  User sql=users
    name Text
    ...

  Article sql=articles
    title Text
    authorId UserId
    ...
|]
```

Foreign ID Field

```
type UserId = Key User
```

Better Approach

- Our database knows about relation **automatically**
- Persistent generates SQL linking tables
- Ensures we cannot delete a user with articles!

Code

```
myUser :: User
myArticle :: UserId -> Article

createArticle SqlPersistT (LoggingT IO) (Key Article)
createArticle = do
  userKey <- insert myUser
  let newArticle = myArticle userKey
  insert newArticle
```

Esqueleto

- Extra SQL library we can use on top of Persistent
- Different, monadic syntax for specifying SQL queries
- Can use type-safe **joins**!

Esqueleto Syntax

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = ...
```

Esqueleto Syntax

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  ...
```

Esqueleto Syntax

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  ...
```

Esqueleto Syntax

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  return users
```

Esqueleto Syntax

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [(Text, Int)]
fetchYoungUsers = select . from $ \users -> do
  return (users ^. UserName, users ^. UserAge)
```

Esqueleto Syntax

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  return users
```

Filtering

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ ...
  return users
```

Filtering

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ (users ^. UserAge <. ...)
  return users
```

Filtering

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ (users ^. UserAge <. 25) -- WRONG!
  return users
```

Filtering

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ (users ^. userAge <. val 25)
  return users
```

Filtering

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ (users ^. userAge <. val 25)
  where_ (users ^. userName == val “James”)
  return users
```

Filtering

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ ((users ^. userAge <. val 25)
      &&. (users ^. userName == val “James”))
  return users
```

Filtering

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ ((users ^. userAge <. val 25)
      ||. (users ^. userName == val “James”))
  return users
```

Select Operations

```
fetchYoungUsers :: SqlPersistT (LoggingT IO) [Entity User]
fetchYoungUsers = select . from $ \users -> do
  where_ (users ^. userAge <. val 25)
  orderBy [asc (users ^. UserName)]
  limit 10
  return users
```

Joins

```
usersAndArticles ::
  SqlPersistT (LoggingT IO) ???
usersAndArticles = ...
```

Joins

```
usersAndArticles ::
  SqlPersistT (LoggingT IO) [(Entity User, Entity Article)]
usersAndArticles = ...
```

Joins

```
usersAndArticles ::
  SqlPersistT (LoggingT IO) [(Entity User, Entity Article)]
usersAndArticles = select . from $ ...
```

Joins

```
usersAndArticles = select . from $
  \(InnerJoin users articles) -> do
    ...
```

Joins

```
usersAndArticles = select . from $
  \(users `InnerJoin` articles) -> do
    ...
```

Joins

```
usersAndArticles = select . from $
  \(users `InnerJoin` articles) -> do
    on (...)
```

Joins

```
usersAndArticles = select . from $
  \(users `InnerJoin` articles) -> do
    on (users ^. UserId ==. articles ^. articleAuthorId)
    ...
```

Joins

```
usersAndArticles = select . from $
  \(users `InnerJoin` articles) -> do
    on (users ^. UserId ==. articles ^. articleAuthorId)
    where_ (users ^. UserName <. val “J”)
    where_ (articles ^. ArticleTitle <. val “J”)
    orderBy
      [asc (users ^. UserAge), asc (articles ^. ArticleTitle)]
    ...
```

Joins

```
usersAndArticles = select . from $
  \(users `InnerJoin` articles) -> do
    on (users ^. UserId ==. articles ^. articleAuthorId)
    where_ (users ^. UserName <. val “J”)
    where_ (articles ^. ArticleTitle <. val “J”)
    orderBy
      [asc (users ^. UserAge), asc (articles ^. ArticleTitle)]
    return (users, articles)
```

JSON Instances

- Often need to serialize/deserialize our data
- JSON is a common format
- `FromJSON` and `ToJSON` typeclasses (`Data.Aeson`)

JSON `Value` Type

```
data Value = ...
```

JSON `Value` Type

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

JSON `Value` Type

```
data Value =
  Object Object |      -- { 'name' : 'James', 'age': 25 }
  Array Array |        -- [ 'j@test.com', 25, {'name': 'james'}]
  String Text |        -- 'James'
  Number Scientific |  --  25
  Bool Bool |          --  True
  Null                 -- (null)
```

JSON `Value` Type

```
data Value =
  Object (HashMap Text Value) |
  Array (Vector Value) |
  String Text |
  Number Scientific |
  Bool Bool |
  Null
```

Converting to JSON

```
class ToJSON a where
  toJSON :: a -> Value
```

Converting to JSON

```
instance ToJSON User where
  toJSON user = object …
```

Converting to JSON

```
instance ToJSON User where
  toJSON user = object
    [ “name” .= userName user
    , “email” .= userEmail user
    , “age” .= userAge user
    ]
```

Converting to JSON

```
instance ToJSON User where
  toJSON user = Array $ Vector.fromList
    [ toJSON $ userName user
    , toJSON $ userEmail user
    , toJSON $ userAge user
    ]
```

Parsing from JSON

```
instance FromJSON User where
  -- No argument because Eta reduction!
  parseJSON = withObject ...
```

Parsing from JSON

```
instance FromJSON User where
  parseJSON = withObject “User” $ \o -> do
    ...
```

Parsing from JSON

```
instance FromJSON User where
  parseJSON = withObject “User” $ \o -> do
    name <- o .: “name”
    ...
```

Parsing from JSON

```
instance FromJSON User where
  parseJSON = withObject “User” $ \o -> do
    name <- o .: “name”
    email <- o .: “email”
    age <- o .: “age”
    return $ User name email age
```

Deriving JSON Instances

```
-- Super simple!
import Data.Aeson.TH (deriveJSON, defaultOptions)

deriveJSON defaultOptions ''User
```

Deriving JSON Instances

- Less boilerplate code
- But trickier to control the instances
- Can help to have definition be explicit
- Versioning issues

Encoding Bytestrings

```
encode :: (ToJSON a) => a -> ByteString
decode :: (FromJSON a) => ByteString -> Maybe a
```

Encoding Bytestrings

```
encode :: (ToJSON a) => a -> ByteString
decode :: (FromJSON a) => ByteString -> Maybe a

>> let bs = encode (User "James" "james@test.com" 25)
>> bs
"{"name":"James", "email":"james@test.com", "age":25}"
>> decode bs
Just (User {name="James", email="james@test.com", age=25})
```

Other Kinds of Types

- Nullable Types and Defaults
- Uniqueness
- Full Custom Types

Nullable Types

```
[PTH.persistLowerCase|

  Article sql=articles
    title Text
    body Text Maybe
    publishedAt UTCTime
    authorId UserId

|]
```

Defaults

```
[PTH.persistLowerCase|

  Article sql=articles
    title Text
    body Text Maybe default=NULL
    publishedAt UTCTime
    authorId UserId
|]
```

Defaults

```
[PTH.persistLowerCase|

  Article sql=articles
    title Text
    body Text Maybe default=NULL
    publishedAt UTCTime default=now()
    authorId UserId
|]
```

Uniqueness

```
[PTH.persistLowerCase|

  Article sql=articles
    title Text
    body Text Maybe default=NULL
    publishedAt UTCTime default=now()
    authorId UserId
    UniqueTitle title
|]
```

Uniqueness

```
[PTH.persistLowerCase|

  Article sql=articles
    title Text
    body Text Maybe default=NULL
    publishedAt UTCTime default=now()
    authorId UserId
    UniqueTitleAuthor title authorId
|]
```

Other Types

```
data Metadata = ...
```

Other Types

```
[PTH.persistLowerCase|

  Article sql=articles
    title Text
    body Text Maybe default=NULL
    publishedAt UTCTime default=now()
    authorId UserId
    metadata Metadata
|]
```

Persist Field

```
class PersistField a where
  toPersistValue :: a -> PersistValue
  fromPersistValue :: PersistValue -> Either Text a
```

Persist Field

```
data Metadata = ...

instance PersistField Metadata where
  toPersistValue metadata = ...
  fromPersistValue persistValue = ...
```

Deriving Persist Field!

```
data Metadata = ...

derivePersistField "Metadata"
```

Deriving Persist Field!

```
data Metadata = ...

derivePersistField "Metadata"
```

Deriving Persist Field!

```
data Metadata = …
  deriving (Show, Read)

derivePersistField "Metadata"
```

Deriving Persist Field!

```
data Metadata = …
  deriving (Show, Read)

derivePersistFieldJSON "Metadata"
```

Migrations

- Persistent gives automatic migrations
- But not full history
- *Destructive* migrations can't be run
- Persistent migration library
  - Allows custom, manually run migrations

Simple Operations

- Create, Drop, Rename Tables
- Add, Drop, Rename Columns
- Add Constraints to tables

Create Table

```
createUser :: Operation
createUser = CreateTable
  ...
```

Create Table

```
createUser :: Operation
createUser = CreateTable
  { name = "users"
  , schema =
    [ Column "id" SqlInt32 [NotNull, AutoIncrement]
    , Column "name" SqlString [NotNull]
    , Column "age" SqlInt32 [NotNull]
    ]
  , ...
  }
```

Create Table

```
createUser :: Operation
createUser = CreateTable
  { name = "users"
  , schema =
    [ Column "id" SqlInt32 [NotNull, AutoIncrement]
    , Column "name" SqlString [NotNull]
    , Column "age" SqlInt32 [NotNull]
    ]
  , constraints =
    [ PrimaryKey ["id"] ]
  }
```

Modifying Columns

```
addEmail :: Operation
addEmail = AddColumn
  "users" (Column "email" SqlString []) Nothing
```

Modifying Columns

```
addEmail :: Operation
addEmail = AddColumn
  "users" (Column "email" SqlString [NotNull]) (Just "")
```

Modifying Columns

```
addEmail :: Operation
addEmail = AddColumn
  "users" (Column "email" SqlString [NotNull]) (Just "")

renameColumn :: Operation
renameColumn = RenameColumn
  "users" "name" "full_name"
```

Modifying Columns

```
addEmail :: Operation
addEmail = AddColumn
  "users" (Column "email" SqlString [NotNull]) (Just "")

renameColumn :: Operation
renameColumn = RenameColumn
  "users" "name" "full_name"

dropColumn :: Operation
dropColumn = DropColumn ("users", "email")
```

Raw SQL Operations

- Perform More Advanced Logic
- Parse existing data into Haskell types
- Plug back into new columns

Raw SQL Operations

```
splitNames :: Operation
splitNames = RawOperation "Split name into first and last" $
  ...
```

Raw SQL Operations

```
splitNames :: Operation
splitNames = RawOperation "Split name into first and last" $
  map migrateNames <$> rawSql "SELECT id, name FROM users" []
  where
    migrateNames :: (Single Int64, Single Text) -> MigrateSql
```

Raw SQL Operations

```
splitNames :: Operation
splitNames = RawOperation "Split name into first and last" $
  map migrateNames <$> rawSql "SELECT id, name FROM users" []
  where
    migrateNames :: (Single Int64, Single Text) -> MigrateSql
    migrateNames (Single id', Single fullName) = ...
```

Raw SQL Operations

```
splitNames :: Operation
splitNames = RawOperation "Split name into first and last" $
  map migrateNames <$> rawSql "SELECT id, name FROM users" []
  where
    migrateNames :: (Single Int64, Single Text) -> MigrateSql
    migrateNames (Single id', Single fullName) =
      let (fullName : rest) = splitOn " " fullName
      in ...
```

Raw SQL Operations

```
splitNames :: Operation
splitNames = RawOperation "Split name into first and last" $
  map migrateNames <$> rawSql "SELECT id, name FROM users" []
  where
    migrateNames :: (Single Int64, Single Text) -> MigrateSql
    migrateNames (Single id', Single fullName) =
      let (fullName : rest) = splitOn " " fullName
      in  MigrateSql
        "UPDATE users SET name = ?, last_name = ? where id = ?"
        ...
```

Raw SQL Operations

```
splitNames :: Operation
splitNames = RawOperation "Split name into first and last" $
  map migrateNames <$> rawSql "SELECT id, name FROM users" []
  where
    migrateNames (Single id', Single fullName) =
      let (fullName : rest) = splitOn " " fullName
      in  MigrateSql
        "UPDATE users SET name = ?, last_name = ? where id = ?"
        [ PersistText firstName, PersistText (unwords rest)
        , PersistInt64 id' ]
```

Migrations

- Combine Operations
- Provide versioning

Migrations

```
lastNameMigration :: Migration
lastNameMigration =
  [ 0 ~> 1 := …
  , 1 ~> 2 := …
  ]
```

Migrations

```
lastNameMigration :: Migration
lastNameMigration =
  [ 0 ~> 1 :=
    [ AddColumn "users" (Column "last_name" …)
    ]
  ]
```

Migrations

```
lastNameMigration :: Migration
lastNameMigration =
  [ 0 ~> 1 :=
    [ AddColumn "users" (Column "last_name" …)
    , splitNames
    ]
  ]
```

Migrations

```
lastNameMigration :: Migration
lastNameMigration =
  [ 0 ~> 1 :=
    [ AddColumn "users" (Column "last_name" …)
    , splitNames
    , RenameColumn "users" "name" "first_name"
    ]
  ]
```

Running Migrations

```
lastNameMigration :: Migration

main :: IO ()
main = runAction connString $
  runMigration
    (MigrateSettings (const $ Just "name_migration"))
    lastNameMigration
```

Class Overview

- **Module 1: Databases and Persistent**
- Module 2: Web APIs, Servant
- Module 3: Monad Transformers and Free Monads
- Module 4: Testing in Haskell
- Module 5: Frontend Web Development with Elm

Module 1 Review

- Use Haskell to Connect to Databases

Module 1 Review

- Use Haskell to Connect to Databases
- Setup Postgres and SQLite

Module 1 Review

- Use Haskell to Connect to Databases
- Setup Postgres and SQLite
- Make a Schema using Persistent

Module 1 Review

- Use Haskell to Connect to Databases
- Setup Postgres and SQLite
- Make a Schema using Persistent
- Write Queries Using these Types

Module 1 Review

- Use Haskell to Connect to Databases
- Setup Postgres and SQLite
- Make a Schema using Persistent
- Write Queries Using these Types
- Serialize Our Data

Module 1 Review

- Use Haskell to Connect to Databases
- Setup Postgres and SQLite
- Make a Schema using Persistent
- Write Queries Using these Types
- Serialize Our Data
- Compile Driven Development

Class Overview

- Module 1: Databases and Persistent
- **Module 2: Web APIs, Servant**
- Module 3: Frontend Web Development with Elm
- Module 4: Monad Transformers and Free Monads
- Module 5: Testing in Haskell