RealWorldHaskell

Requirements

Postgresql

Building this code requires that you have Postgres installed on your system. If you run `stack build` and see the following error message, this indicates that you do not have Postgres:

```
>> stack build
setup: The program 'pg_config' is required but it could not be found
```

On Linux, you'll want at least the following packages:

```
>> sudo apt install postgresql postgresql-contrib libpq-dev
```

On Windows and MacOS, you should be able to use the downloads here.

Redis

Starting with part 3, our server starts incorporating caching using Redis. You'll want to follow the instructions for installing it based on your operating system.

The two main functions you should be concerned with are running the server and the client CLI. With the server, you should be able to run the `redis-server` command and leave it running in the background. Then you can bring up the CLI with `redis-cli` and run some basic commands:

```
>> redis-cli
> EXISTS "1"
1
> EXISTS "45"
0
```

Docker

In part 4 we write some tests, using Docker so that the tests are agnostic to the programmer's environment. So head to the Docker homepage and follow the instructions to get it working on your system.

Running the Code

Part 1: Persistent

The code for this part can be run pretty easily through GHCI. The main thing is you need your Postgres server to be up and running. You can modify the `localConnString` variable in the code matches up with the settings you used. The default we use is that the username, DB name, and password are all "postgres":

```
localConnString :: PGInfo
localConnString = "host=127.0.0.1 port=5432 user=postgres dbname=postgres
password=postgres"
```

Then once you load the code in GHCI, you should use the `migrateDB` expression. This will migrate your Postgres database so that it contains the `users` table specified in our schema! Note how you can also set your connection string here as well if it's different from our built-in.

```
>> stack ghci
>> :l
-- (Removes all modules so there are no name conflicts)
>> import Database
>> let localConnString' = "host=127.0.0.1 port=5432 user=postgres dbname=postgres
password=postgres" :: PGInfo
>> migrateDB localConnString'
```

Then you'll be able to start running queries using the other functions in the `Database` module

```
>> let u = User "Kristina" "kristina@gmail.com" 45 "Software Engineer"
>> createUserPG localConnString u
1
>> fetchUserPG localConnString 1
Just (User {userName = "Kristina", userEmail = "kristina@gmail.com", userAge = 45,
userOccupation = "Software Engineer"})
>> deleteUserPG localConnString 1
```

After each step, you can also check your Postgres database to verify that the queries went through! You can bring up a Postgres query terminal with the `psql` command. You can use the `-U` argument to pass your username. Then enter your password. And finally, you can connect to a different database with `\c`.

```
>> psql -U postgres
(enter password)
>> \c postgres
>> select * from users;
(See the users you've created!)
```

Part 2: Servant

In this second part, we make a very basic server to expose the information in our database. Take a look at the source in this module.

To run this server, first make your database is migrated, if you didn't do that in part 1:

```
>> stack exec migrate-db
```

Then you can run the server with this executable:

```
>> stack exec run-server
```

Now you can make HTTP requests to your server from any client program. My favorite is Postman. Then you can follow the same pattern you did in the first part. Try creating a user:

```
POST /users
{
  "name": "Kristina",
  "email": "kristina@gmail.com",
  "age": 45,
  "occupation": "Software Engineer"
}

...

2
```

Then try fetching it:

```
GET /users/2

...

{
  "name": "Kristina",
  "email": "kristina@gmail.com",
  "age": 45,
  "occupation": "Software Engineer"
}
```

You can also try fetching invalid users!

```
GET /users/45

...

Could not find user with that ID
```

Part 3: Redis

The third part of the series discusses a library for using Redis in Haskell. The code examples in the article can be seen in the Cache Module and the CacheServer module, which has updates to the original server from part 2 with our caching functionality.

You can run the updated server with the `run-server` executable and the `cache` argument, though you should have the Redis server running in the background first:

```
>> redis-server &
>> stack exec run-server -- cache
```

The external API is the same, so you can make the same kinds of HTTP requests. For example, following the example above in the Part 2 section, we might do `GET /users/2`. Then you should be able to bring up the `redis-cli` and find that the user is stored in our cache:

```
>> redis-cli
> GET "2"
"User {userName = \"Kristina\", userEmail = \"kristina@gmail.com\", userAge = 45,
userOccupation = \"Software Engineer\"}"
> GET "45"
(nil)
```

Part 4: Docker Tests

In this part, we write some tests for our Server. You can see some setup code in this module, but the main assertions are written in Hspec in this file. You can run the tests by running:

```
stack test
```

These tests depend on having a postgres database and a Redis cache running on your machine. There are two options for them. First, you can rely on your local system, and run these services as you have been so far. Be aware that the tests have "side effects" this way. If they run to completion, there shouldn't be any extra rows in the table. However, because the tests insert and then delete a user, the primary key index will increment.

The second option is to use Docker. To do this, you can start the Docker container specified in the repository by going to the root project directory and running the following command:

```
>> docker-compose up
```

You can observe the configurations in the Docker Compose file.

Then, you need to go to the stack.yaml file and change the project settings to use Docker. You'll change the `docker.enable` flag to `true`:

```
docker:
  enabled: true
```

Then you can run `stack test` again. The first time you do this, the Docker container will need extra time to setup, downloading Stack, GHC, and all the utilities it needs. But subsequent runs will be faster.

Part 5: Esqueleto

In part 5, we add a new type to our schema, this time incorporating a foreign key relation. This part has its own distinct set of modules to avoid conflicts with the code from the first 4 parts:

1. New Schema Module
2. New Database Library
3. Updated Server (this server does not have any Redis caching)
4. Sample Objects (for database insertion)

To try out this code, you should start by running a new migration on your database:

```
>> stack exec migrate-db -- esq
```

This will add the `articles` table, but it should leave the `users` table unaffected, so it shouldn't cause any problems with your original code.

Next you can update the database in a couple different ways. First, as with the first part, you can open up GHCI and try running the insertions for yourself. In the `SampleObjects` module, we've provided a set of objects you can use as sample database items. The `User` objects are fine on their own, but the `Article` objects require you to pass in the integer ID of the User after they've been created. For example:

```
>> stack ghci
>> :l
>> :load DatabaseEsq SampleObjects
>> import SampleObjects
>> createUserPG localConnString testUser1
5
>> createArticlePG localConnString (testArticle1 5)
1
>> fetchRecentArticles
[(Entity {entityKey = SqlBackendKey 5, entityVal = User {...}}, Entity {entityKey
= SqlBackendKey 1, entityVal = Article {...}})]
```

The other way to do this is to use the API via the server. Start by running the updated server:

```
>> stack exec run-server -- esq
```

And then you can make your requests, via Postman or whatever service you use:

```
POST /users
{
  "name": "Kristina",
  "email": "kristina@gmail.com",
  "age": 45,
  "occupation": "Software Engineer"
}

5

POST /articles

{
  "title": "First Post",
  "body": "A great description of our first blog post body.",
  "publishedTime": 1498914000,
  "authorId": 5
}

1

GET /articles/recent
[
  [
    {
      "name": "Kristina",
      "email": "kristina@gmail.com",
      "age": 45,
      "occupation": "Software Engineer",
      "id": 5
    },
    {
      "title": "First Post",
      "body": "A great description of our first blog post body.",
      "publishedTime": 1498914000,
      "authorId": 5,
      "id": 1
    }
  ]
]
```