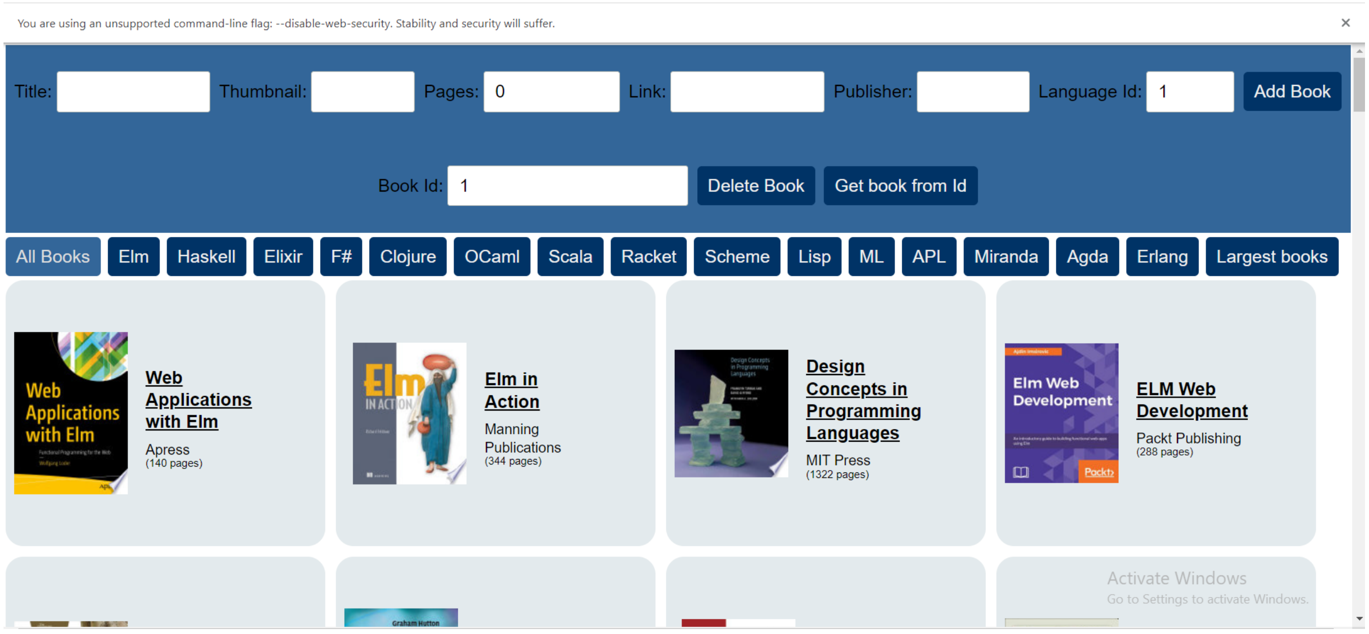
FUNKCIONALNO-PROJEKAT

Ovo je prikaz i opis web-aplikacije koju smo radili za projekat iz Funkcionalnog programiranja. Zadatak nam je bio realizovati web-aplikaciju u čisto funkcionalnim jezicima, a prvenstveno u Haskelu, koji smo radili i na nastavi. Ja sam se odlučila da pravim aplikaciju iz dva dijela, klijent i server; da klijent i server komuniciraju preko http-zahtjeva i odgovora, a da pri tom podaci uglavnom budu prenošeni u obliku JSON-a, u body elementu http-zahtjeva ili odgovora. Klijent sam realizovala u jeziku za frontend pod nazivom Elm, o kojem ce biti više riječi kasnije. Server sam realizovala u Haskellu, koristeći PostgreSQL bazu podataka i Servant biblioteku za izradu servera i API-ja. Za konekciju sa bazom koristi se biblioteka Persistent, koja omogućava ORM (object relational maping), tj. ekvivalentan koncept u funkcionalnim jezicima. Ta biblioteka takođe omogućava jednostavnije upite u obliku funkcija kao što su get, insert, delete, čijom se upotrebom smanjuje potreba za pisanjem čistog SQL koda od strane programera, pa prema tome se smanjuje mogućnost napada kao što su SQL injection. Druga biblioteka koričćena za pisanje upita je Esqueleto. Ona omogućava pisanje složenijih upita kao što su JOIN-i dvije ili više tabela, WHERE upiti i slično.

Ideja za ovakav izbor tehnologija i strukture aplikacije mi je prirodno došla jer sam već izrađivala web-aplikacije kod kojih je React korišćen za frontend, a Node.js za backend; frontend je prema tome SPA (single-page application), a backend je RESTful API. Izradom ovog projekta sam dobila priliku da vidim prednosti i mane korišćenja funkcionalnih jezika, dok mi je sama arhitektura aplikacije ostala ista kako sam i navikla.

Aplikacija se zove `Katalog knjiga o funkcionalnim jezicima` i pruža mogućnost prikaza velikog broja knjiga koje se bave funkcionalnim programskim jzicima. Klijent nakon pokretanja izgleda kao na narednoj slici, gdje se na stranici ispod forme prikaže red dugmadi za filtriranje knjiga po kategorijama. Prvo dugme u redu postiže isti efekat kao pokretanje aplikacije, a to je prikaz svih knjiga u nizu kartica, dok se na svaku karticu može kliknuti, što nas vodi na Google Books stranicu namijenjenu toj knjizi. Ostala dugmad filtriraju kartice, tako da se prikažu knjie po jezicima, dok posljednje dugme vraća prikaz 10 najobimnijih knjiga u bazi i postoji prvenstveno da bi se demonstrirala upotreba složenijih upita, kao što su JOIN dvije tabele i sotiranje. Forma omogućuje unos podataka o novoj knjizi koja se klikom na dugme Add Book dodaje u bazu, kao i brisanje knjige sa unijetim indeksom, ako ona postoji u bazi, ili pak selektovanje i prikaz jedne knjige koja odgovara unijetom indeksu.


You are using an unsupported command-line flag: --disable-web-security. Stability and security will suffer.

Title: Thumbnail: Pages: 0 Link: Publisher: Language Id: 1 Add Book

Book Id: 1 Delete Book Get book from Id

All Books | Elm | Haskell | Elixir | F# | Clojure | OCaml | Scala | Racket | Scheme | Lisp | ML | APL | Miranda | Agda | Erlang | Largest books

Web Applications with Elm
Apress
(140 pages)

Elm in Action
Manning Publications
(344 pages)

Design Concepts in Programming Languages
MIT Press
(1322 pages)

ELM Web Development
Packt Publishing
(288 pages)

Naredni dio ovog teksta, za server koji je rađen u Haskelu, je preuzet i prilagodjen sa: https://github.com/MondayMorningHaskell/RealWorldHaskell#readme. Ostavljeni su dijelovi koje sam koristila kao primjer i osnovu, a to su prvi, drugi i peti dio, a proširen je tekst u nekim dijelovima, jer je i moj kod opširniji od ovog koji mi je služio kao polazna tačka i referenca. Zahvaljujem se prema tome autorima i toplo preporučujem njihov blog, gdje je sve još detaljnije objašnjeno Real World Haskell

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.

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 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
    }
  ]
]
```