Module 2 Exercises

Monday Morning Haskell
Practical Haskell Course

Lecture 1

Before you start the rest of the exercises for this module, make sure to checkout the code for this branch on the Github repository! You'll want to save any changes you made to the `module-1` branch, and then pull the `module-2` branch. Here's the series of commands you can follow to do that, running from the `PracticalHaskell` root directory:

```
git add .
git commit -m “Module 1 Changes”
git fetch origin module-2
git checkout module-2
```

We'll also be using a few different programs and tools for this module, so you'll want to make sure you've got a few things installed before we get too far. First, make sure you've installed the Postman program. This allows you to manually craft HTTP requests to send to web servers, including a server running locally. It'll be a useful tool for checking how our code works.

You'll also want to make sure you have free accounts set up for Heroku and Circle CI. We'll be using these tools to setup our servers to actually run on the internet. With Circle CI, you'll want to choose the option where you authenticate through your Github account. With Heroku, you'll also want to install the Heroku Command Line Interface.

Lecture 2

For this lecture, take a look at `BasicServer.hs`. It has a really simple HTTP server built using Servant. You can run the server by building the code on this branch and then using `stack exec run-basic-server`.

To start off, run the server, and then open up Postman. Test the server out by sending the following GET requests to the server. Observe the responses at the bottom.

```
http://localhost:8080/api/version
http://localhost:8080/api/name
http://localhost:8080/api/description
```

Take a look at the server code, and see that you can draw the connection between the different endpoints you called and their output. Then try to follow the patterns in the file and add your own new endpoints. Make it so that the endpoint `/api/libraries` returns the string `"This server uses Servant"`, and that the endpoint `/api/num_users` returns the number `1`. (You must add them in this order for the tests to work).

Next, observe in the code that we're hard-coding the port as 8080. Add an environment variable `BASIC_SERVER_PORT` containing this value. Use this environment variable instead of hard-coding it. (Note you'll probably have to open a new terminal to recognize the new environment variable before running the server again).

Verify that these endpoints work as expected in Postman. You can also run `test-2` as a quick check that you've made the endpoints work and that you've substituted the environment variable properly.

As a last note, you can also use Postman to make requests to any URL. Try sending a request to https://www.mmhaskell.com. Your output should be a lot of HTML. We'll see eventually how to return an HTML page from our servers.

Lecture 3

Go back to `BasicServer.hs`. To start, modify the description endpoint so that it adds the path component "`brief`". Use Postman queries to ensure that this change works.

Add a `Capture` parameter to the `name` endpoint. Have this capture a parameter `user_id` of type `Int64`. Look up the name in the `nameLookup` dictionary at the bottom of the file. If the ID doesn't exist in the map, throw a 404 error, stating "`That user doesn't exist!`". Verify this works as expected using Postman.

In `BasicServerTypes.hs`, you'll find a couple wrapper types. A `BasicServerUserId` wraps an `Int64`, and a `DescriptionResponse` wraps `Text`. Use `BasicServerUserId` as the capture parameter for the `name` endpoint, and use `DescriptionResponse` as the result type of the `description` endpoint. You'll need to define the typeclass instances listed in that file in order to make this work.

Remember to test your API on your own by using `run-basic-server` and sending requests through Postman. Observe that you shouldn't need to reference these new wrapper types at all when sending and receiving raw HTTP requests. When you're all done, you should be able to test your code with `test-3`.

Lecture 4

In the exercises for this lecture, we're going to start making a basic REST API for the types we made in our schema from the last module. You'll be working in `RestServer.hs`. Fill in the type `RestAPI` with the following endpoints:

1. `GET /users` – Retrieve all the users in `nameDictionary`
2. `GET /users/:id` – Retrieve a particular user by their `Int64` ID in `nameDictionary`. This should throw a 404 error if the ID doesn't exist in our dictionary.
3. `GET /users/filter?age&emails` – Take two query parameters: first, a possible age to filter by, and second, a list of emails. Return all users who are at least as old as the given age, and whose email is in the supplied list. If no parameter is supplied for a given parameter, do not use it as a filter. (i.e. if no emails are queried, return all users that fit the age filter). Include their IDs in the return value. That is, your return type should be `[(Int64, User)]`.
4. `POST /users/create` – Take a User as a JSON input in the request body. Return the `Int64` ID the user would have if we inserted it into our dictionary (i.e. one past the maximum ID in the dictionary). Since the dictionary is immutable, we can't actually create this new user, but print it out anyways. We'll get into combining our API with a real database in a couple lectures.
5. `DELETE /users/delete/:id` – Take an `Int64` ID as an input. Once again, we can't actually "delete" our user, but print the user that you would delete. Return `()` as the output of this endpoint. This should throw a 404 error if the ID doesn't exist.

Be sure to also fill in the proxy for your type, `restAPI`. Fill out `runRestAPI` so that it runs a server with your API on the port 8080. To run your API, you'll want to use `stack ghci` and then call your `runRestAPI` function.

Test your API by trying each endpoint in Postman. Then you can use the `test-4` command to run some automated tests on this API. Note you must implement these endpoints *in order* for the tests to work.

Lecture 5

You have three tasks for this lecture. First, in `RestAPIClient.hs`, define a set of client functions for the API you created in the last lecture. Then, fill in the function `runClientAction`. This should take a client action call and use it on your API! You'll need to generate the `ClientEnv` as part of this function. You should use the `Int` argument as the port number for the base URL for the environment.

When you're done, make two terminal windows with `stack ghci`. On one of them run `RestServer.runRestAPI` to start your server. On the other, use `runClientAction` to test your different client commands!

The final part of this lecture wraps together a lot of the concepts we've been working with. We're going to imagine we're calling someone else's API! You can run a server with this API using:

```
stack exec -- run-secret-server.
```

The API is set up with two endpoints:

```
POST /auth
GET /secret/:api_key
```

The first takes a request body containing a JSON object, with string fields `username` and `password`. This returns a `Text` API key if you provide “proper” login information (use `username “test”` and `password “pass”`).

The second endpoint captures the API key as a parameter and returns another `Text` item which is the API secret.

Your job is to fill in `retrieveSecret`. This function will make two calls to the API. Once to login and get the API Key, and then another to use the API key to retrieve the secret. Once again, you'll want to test this by running the secret server in one GHCI window and calling your function in another.

To do this, you should make a Servant API type for the server, even though it isn't your own! Then you can make client functions from this type. You'll also need to come up with a type to represent the initial authentication information, along with JSON instances for it. Your `runClientAction` function will also come in handy. The secret server uses port 8081.

This lecture comes with automated tests. When you're done, run them with `test-5`.

Lecture 6

Before you start this lecture, ensure you have a clean database, or the tests might flake out. Drop all the existing tables in the `mmh` database and run:

```
psql -U postgres mmh < db_dumps/database_3.sql
```

In the lecture slides, we saw a fleshed out example of a series of CRUD endpoints for the `User` type. You can find a version of this in `CRUDServer.hs`. Your first task for this lecture is to repeat this process for the other two types in our schema, `Article` and `Comment`. Fill in the `ArticlesCRUD` and `CommentsCRUD` types, the associated handler functions, and the `Server` expressions. Make sure to follow the guide from the User type for the order of your endpoints, or the tests will not work.

Next, make the full API complete by adding the new type definitions to `FullCRUD` and `fullServer`. Go into GHCI, kick off the `runServer` function from this module and do a little testing of your endpoints. The `test-6` command will also run some tests on your API.

Finally, see if you can make a generic API type `CrudAPI` with a type parameter `a`. Test it out by using this abbreviation of `UsersCRUD`, `ArticlesCRUD`, and `CommentsCRUD` instead of the long-form definitions. Make sure the tests still work!

Lecture 7

The code for this lecture is in `AuthServer.hs`. Throughout these exercises, you'll want to have two GHCI prompts available. One should run the server with `AuthServer.runServer`. You should use the other to call the server client helper functions such as `createUserHelper`, `retrieveUserHelper`, and so on. You can also use Postman instead.

To start off, observe, by using the `retrieveUserHelper` function, that you can use this endpoint to retrieve the data for *any* user. Try any ID from 2-101 and check that it works. The `String` arguments for this function are for the email and password of that user. But we don't use them yet, so you can enter anything! Our first goal is to change this so that it is an authenticated endpoint that only allows a user to access their own information. This is a multi-stage process. Here's an outline:

1. Modify the retrieval endpoint so that it uses `BasicAuth` on the `"user"` realm, returning an `Int64`.
2. Add this parameter to the `retrieveUserHandler`. If the authentication ID and the capture parameter ID do not match, throw a 403 error from the endpoint.
3. Fill out the `authCheck` function. A user will pass their email as the user name. Use this to go into the database and retrieve the ID and the `AuthData` related to that user. Then use the `verifyPassword` function from `Crypto.PasswordStore`. Return authentication results such as `Unauthorized` or `BadPassword` in the appropriate cases.
4. Modify the `authContext` to use this check.
5. Add `BasicAuthData` as a parameter to the client function type definition at the bottom of the file.
6. Update the helper function to make `BasicAuthData` and use it in the client call.

Now check the endpoint once again, and ensure that you can only retrieve the data with the proper authentication information. To do this, you can first try creating a user with the `createUserHelper` function so you can set your own password. Pay attention to the ID that gets printed when you run this, as you'll need it to retrieve it. In addition, you can use the following combinations to test the endpoint:

```
ID | Email                  | Password
50 | Santa@Washington.com | "christmas"
80 | Romaine@Trent.com    | "lettuce"
```

For the final task in this lecture, try adding a separate authentication realm for admin users, and apply this to the "all users" endpoint, so that only admins can retrieve all the users. When adding the combinator to your API, you can use a different string if you like (`"admin"`), but it will use the same auth check as above if you use the same `Int64` type! We don't want this!

To get around this, use the wrapper `newtype AdminUserId` as the result type of the `BasicAuth` combinator on this endpoint. Follow the same steps you did above. Your `adminAuthCheck` should be identical to the original `authCheck`, except that it will also check that the `authDataUserType` is equal to `"admin"`. Then it will need to wrap the resulting `Int64` id. Use "admin@test.com" as the admin email, and "adminpassword" as the password. You should get the user count of 101.

When you're all done, you can use `test-7` to verify your results!

As an added optional challenge, you can try adding another endpoint to create admin users. It should require being authenticated as another admin user. Then the user you create should have `"admin"` for their type instead of `"user"`. Note, doing this will cause the automated tests for this lecture to fail.

Lecture 8

For this lecture, we'll be working in `AdvancedAuthServer.hs`, which implements roughly the same API we had before except it also has `login` and `logout` endpoints. More of the initial boiler-plate is already implemented.

Take a second to observe the type family instances for `AuthProtect "user"`, `authenticateReq`, and `authHandler` near the top of the file. In particular, `authenticateReq` shows you how to add an item to a `Request` header, and `authHandler` shows you how to retrieve it.

The `authHandler` also calls a function `decodeJWTCookie`. This verifies that a given cookie corresponds to a given user. It lives in `JWTHelpers.hs`. Feel free to peruse this file and get a feel for how we can work with the `Web.JWT` library. But you don't need to change anything in it.

User Level Authentication

For your first task, incorporate `"user"` level authentication for the "retrieve user" endpoint in `UserAuthAPI` like you did for basic authentication. This time, use the `AuthProtect "user"` combinator. You'll need to add the extra parameters to the `Handler` and `ClientM` functions, and update the behavior of `fetchUserHandler` and `retrieveUserHelper` as needed. Then update the type of `authContext` and add the `authHandler` to it.

Logging In and Out

Next, implement the `loginHandler`. This should take a user's email and password, and provide them with their user ID and a JWT cookie, which you should also insert into the database as a `LoginToken`**.

You should incorporate the `makeJWTCookie` function from `JWTHelpers.hs`, as imported. You'll want to incorporate a lot of code from your `authCheck` in the last lecture so you can check that the password is correct! The difference between this and `BasicAuth` is that we only send the plain password once, rather than with every request. Throw errors (like `err403`) as appropriate.

Next, implement `logoutHandler`. This should log a user out by deleting all of their `LoginTokens`. You'll need to add `"user"` level authentication to this endpoint as well.

(Continues next page)

Admin Authentication

Finally, add admin authentication for the “all users” endpoint. This should use `AuthProtect “admin”` instead of `AuthProtect “user”`. As in the last lecture, use `AdminUserId` as the resulting type, rather than a plain `Int64`. This process involves a few steps, but most of the hard work is already done!

1. Modify the endpoint, handler function, and client function with the appropriate combinator and parameters.
2. Fill in the expressions `adminAuthHandler` and `adminAuthenticateReq`. The first will be a lot like `authHandler` except for incorporating an extra `userType` check and wrapping the result. The second will be identical to `authenticateReq` except for the type signatures!
3. Add `adminAuthHandler` to your `authContext` and modify the type signature appropriately. Then incorporate `adminAuthenticateReq` into `fetchAllUsersHelper`.

Testing

When you're all done, use `test-8` to run some automated tests. Note, these tests will delete any `LoginTokens` you've made. You're also highly encouraged to try out the various endpoints using Postman, as shown in the screencast. There are also appropriate functions for each endpoint if you'd rather test in a GHCI environment. It can be difficult to copy/paste cookies in GHCI though.

As mentioned in the screencast and lecture, **you should not store JWT cookies in a database for a production server. These should be stored on the server as part of its “session”, not on a persistent table. Remember, a JWT is almost as good as a password for most cases, since it allows an attacker to impersonate a user! So doing this isn't much better than storing their passwords in plain text. It's not *as bad* as long as you require a user's password to reset their password. An attacker with the cookie will still be able to search around and get information, but they won't be able to lock the original user out.

In module 4, we'll see a good way to store cookies as part of a Servant session.

Lecture 9

For these exercises we're going to follow the outline we've seen so far in the slides and screencast.

Creating the App

Starting by opening up a console and logging into Heroku with the `heroku login` command, entering your email and password.

Next come up with a name from your application that should be unique to you. For instance `practical-haskell-{GITHUB_NAME}` would be a good idea. We'll use `$APP_NAME` from now on to refer to whatever you named it. Create a new application with this name using the following command:

```
heroku create $APP_NAME -b https://github.com/mfine/heroku-buildpack-stack
```

Now, from the base directory of the project, add the automatically generated Heroku Github links for your app like so:

```
heroku git:remote -a $APP_NAME
```

Confirm you've added these with the command `git remote -v`. You should see the following two lines appear, in addition to the normal Git remote options:

```
heroku https://git.heroku.com/$APP_NAME.git (fetch)
heroku https://git.heroku.com/$APP_NAME.git (push)
```

Now push your current branch (`module-2`) to the master branch of the Heroku remote like so:

```
git push -f heroku module-2:refs/heads/master
```

This will kick off some long terminal output. Your remote Heroku container will start by downloading programs like Stack and GHC and installing them. Then it will build our project. Since the project has a lot of dependencies, this will take a little while (mine took 30 minutes)! It might even timeout! If this happens, don't worry! Heroku caches build artifacts. So if you push again, it won't need to re-build from scratch! Eventually it will complete!

The last step for our app really runs is that we need to make sure Heroku gives it a machine to run on. This is called “scaling” the app. Run the following command to ensure the app can run:

```
heroku ps:scale web=1
```

Now you should be able to go to `http://$APP_NAME.herokuapp.com/hello`, and your web browser should show you “`Hello Haskell Heroku`”, just as we've written for the `hello` endpoint in `FinalServer.hs`! Now we need to ensure we can connect to a database appropriately (see next page).

Connecting to a Database

We now need to provision a database for our application. This involves using an addon. First verify that your app has no addons with the following command:

```
heroku addons
```

Now, add a Postgresql database addon with the following command:

```
heroku addons:create heroku-postgresql:hobby-dev --version=10
```

It may take a few minutes for your database to become available. In the meantime, we need to make a couple changes to our application to ensure it works appropriately. First, we'll want to ensure our database is migrated appropriately. We want to make this part of our build process. Go to `Procfile` and observe that the only command we run right now for the `web` process is `run-final-server`. Change this so that it also runs our migration command:

```
migrate-db && run-final-server
```

Next, go to `FinalServer.hs`. Observe how we are using `lookupEnv` to first check for an environment variable called `PORT`. Heroku assigns this variable for us to use as our application's port. If we find this, we'll use it as the port, otherwise we'll use a default. Instead of using `localConnString` as the database connection, you should now do the same thing with `DATABASE_URL`. Commit your changes to this branch on your Github fork. Then push to the Heroku master branch again!

Once your application is ready, try hitting some of the endpoints. Create a user by hitting the `"create"` endpoint in Postman (remember to use your app's Heroku base URL!). Then try retrieving the user information from the `"retrieve"` endpoint by adding the email and password as basic auth data. You've now got a functioning application on the cloud!

Lecture 10

For these exercises, we'll get our feet wet with combining Circle CI with our Haskell code. Start off by making sure your Github fork is connected to Circle CI. Log on to the Circle CI website and connect it to your Github account. Then make sure this project is connected to Circle.

Then, take a look at our config at `.circleci/config.yml`. Here we have all the basic steps in place. We use a Docker image to get a base container that has Stack installed already. Then for the `steps`, we `setup` the machine, `build` our code, and then run the `final-server-tests`. (These are some simplistic tests that just check the basic `hello` endpoint).

For your first task, push the code changes you made for the last lecture, and push them to the normal origin, rather than the Heroku origin:

```
git push origin module-2
```

Then go to the Circle CI site for your repository, and observe the status of this build. As with Heroku, this will take a while, so you may want to come back later and see the results. Your build step may even time out!

When the build is finished, make a small change to your code, like adding a comment. Then make a separate commit, and push this to origin. If you watch the build process again, you'll see that it repeats building every dependency! We're going to fix that by adding two more steps to our workflow.

After the `test` step, add a `save_cache` step. This should use a key involving a `checksum` on your `stack.yaml` file and `PracticalHaskell.cabal`. For the paths, you should save `"/root/.stack"` and `".stack-work"`.

Then, after the `checkout` step, add a `restore_cache` step. This should have the same key as you saved it with. Commit and push this change. The next build will still take a long time, like the old changes.

But then, you should make one more small commit and push. Observe that the next time Circle CI runs, it doesn't need to re-build the same dependencies! Your build will complete a lot faster. If it was timing out before, it should now complete! Of course, if you change the `.cabal` file or `stack.yaml`, it will need to do another long build, because you might have changed the dependencies. So keep this in mind when updating these files!

At this point, we'll observe that if we want to deploy our code, we'll need two different pushes. We need to push to the origin to test our code, and then to Heroku to deploy it. However, Heroku makes it very easy to integrate these services! Go to the Heroku dashboard for this application, and select the `Deploy` panel. Then select the GitHub option instead of Heroku Git (you might need to connect your Github account to Heroku). Then you can select your fork of `PracticalHaskell` as the repository, and you can use the `module-2` branch. Check the box for "Wait for CI to pass before deploy", and then click the "Enable Automatic Deploys" button.

Now, anytime you push to the `module-2` branch (the Github origin, not the Heroku origin), Circle CI will run your tests and, if they pass, deploy your code. Awesome! For the last part of these exercises, try making a small change to the server, push it to Github, watch the tests pass in Circle CI, and ensure it deploys automatically. You might, for instance, change the `hello` endpoint string. But if you do that, make sure you also change the test string in `FinalServerTests.hs`!