Module 3 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-2` branch, and then pull the `module-3` 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 2 Changes"
git fetch origin module-3
git checkout module-3
```

You'll also want to make sure you have Elm installed! Take a look at this page for details. If you're on Windows or a Mac, the installer program should be sufficient. On Linux, you'll either want to use `npm` or directly download the binary. Make sure you get version 0.19!

Most of our work for this module goes in the `frontend` directory that should now exist on this branch. Go to this branch and run `elm install` to get all the proper packages loaded.

Then use the `elm repl` command to bring up a REPL similar to GHCI. Try a few different commands and see what the output is. (Note this command uses Node.js to execute Javascript outside the browser. If you don't have Node installed, you will need to do so for the REPL, but not for the rest of this module).

```
>> 3 + 4
>> 5 / 2
>> List.length [2, 4, 6, 8]
>> String.reverse "Hello World!"
>> :exit
```

Another command to familiarize yourself with is `elm reactor`. Try this command, and bring up a web browser to `localhost:8000`. Follow the links to open up `src/HelloWorld.elm`. You can also examine the `frontend/src/HelloWorld.elm` source file to see what goes into making this simple web page.

Lecture 2

For this lecture, you'll be working in `frontend/src/BasicPractice.elm`, where you should find a couple Haskell examples that are commented out. The first contains a couple Haskell types for points and shapes. There's also an accompanying `calculateArea` function.

Convert all this code to Elm. The `Point` type should be a type alias for one record with `x` and `y` fields. Then `Shape` should be a type with two different constructors, each of which has a single record field. For a `Triangle`, name these fields `p1`, `p2`, and `p3`. For a `Circle`, name these fields `radius` and `center`. Then, convert the `calculateArea` function into Elm.

Your next example concerns monads and lists. In Haskell, lists are a monad, so we can use do-syntax with them. In the commented out example, we put Haskell list do-syntax to use in the `sumsAndProducts` function. We take a list and two other integer arguments. For every element in the list, we take both the sum and product with the first argument `j`. Then we subtract all of these results from `k`. There are a couple examples listed.

In Elm, there is no built-in `List.andThen` function to treat lists as a monad. However, we can make our own definition with `listAndThen`! Fill in this function to capture the Haskell list monad behavior. Then use your function to convert `sumsAndProducts` to Elm!

Hint: While Elm hasn't defined it as `andThen`, there's still a List library function that's very useful here! Pay attention to the type signatures!

To determine if your code works, run `elm reactor` and open up `BasicPracticeTests.elm` in your brower. It should compile properly, and all your tests should be green.

Lecture 3

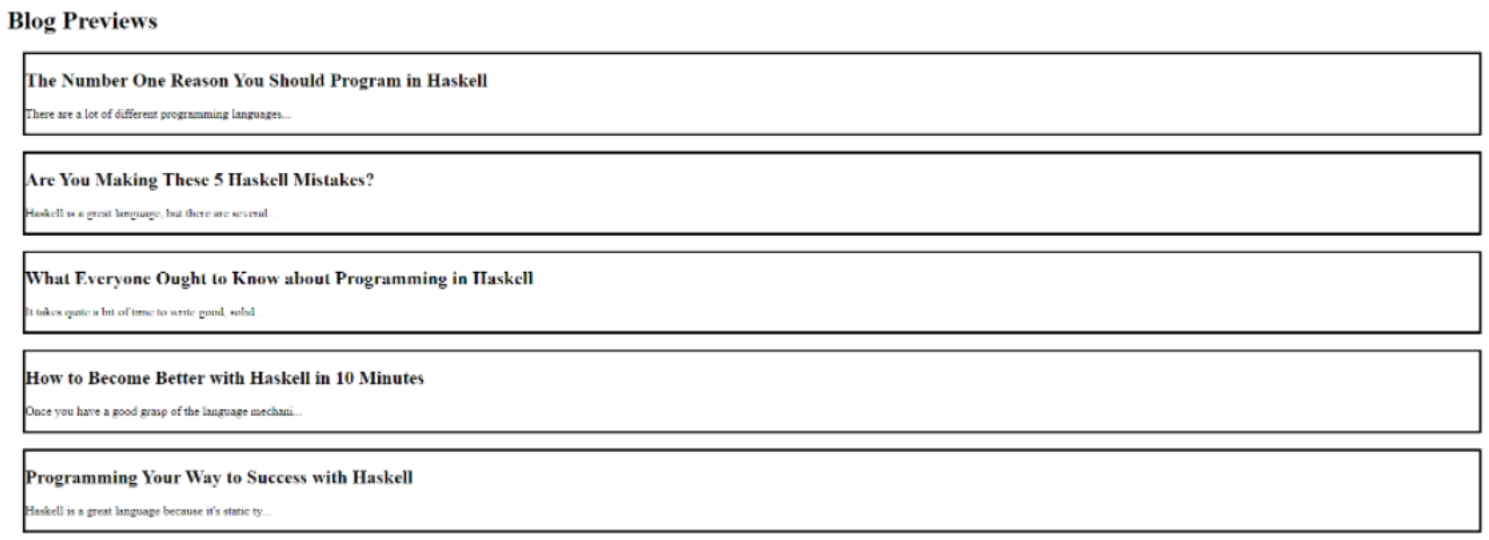
For these exercises, you'll start working on our overarching project for this module: building a blog page to display the articles we're storing in our database. You'll be working in `BlogLandingBasic.elm` for this lecture. Open up that file and notice we have a record type alias for `Article`, as well as a hard-coded list of Articles to display.

This list is the “model” for this page, which we capture under the `BlogLandingModel` type. We'll discuss the relationship between the model and the view more in the next lecture. But for now, your task is to fill in the `view` function. This should take our list of articles, and display a list of previews for them.

Each preview should have the title of the article in an `h2` element, with a preview on the body text in a `p` element (take the first 50 characters or so and then add an elipsis). Surround each separate preview with a black border using the `border-style: solid` CSS style. You can also use margins on each of the elements to keep them a bit more separated. Another good idea would be to add an h1 element in front of this section with the title `Blog Previews`.

On the whole, this page does not have to look pretty! Towards the end of the module, we'll show how to incorporate full CSS stylesheets into our Elm application. At the bottom of the page you'll see my basic rendering of these previews.

You might find it helps to read up a bit on the [doucmentation for Elm Strings](#).



Blog Previews

The Number One Reason You Should Program in Haskell

There are a lot of different programming languages...

Are You Making These 5 Haskell Mistakes?

Haskell is a great language, but there are several...

What Everyone Ought to Know about Programming in Haskell

It takes quite a lot of time to write good, solid ...

How to Become Better with Haskell in 10 Minutes

Once you have a good grasp of the language mechani...

Programming Your Way to Success with Haskell

Haskell is a great language because it's static ty...

Lecture 4

For these exercises you'll continue to work in `BlogLandingBasic.elm`. We'll want to add a few extra features to our landing page.

Your first task is to add a new section below the previews for adding a new blog post. This should be a form with a normal text `input` element for the title, and a `textarea` input for the body of the article. You'll want a `"Submit"` button at the bottom of the form to create this new article.

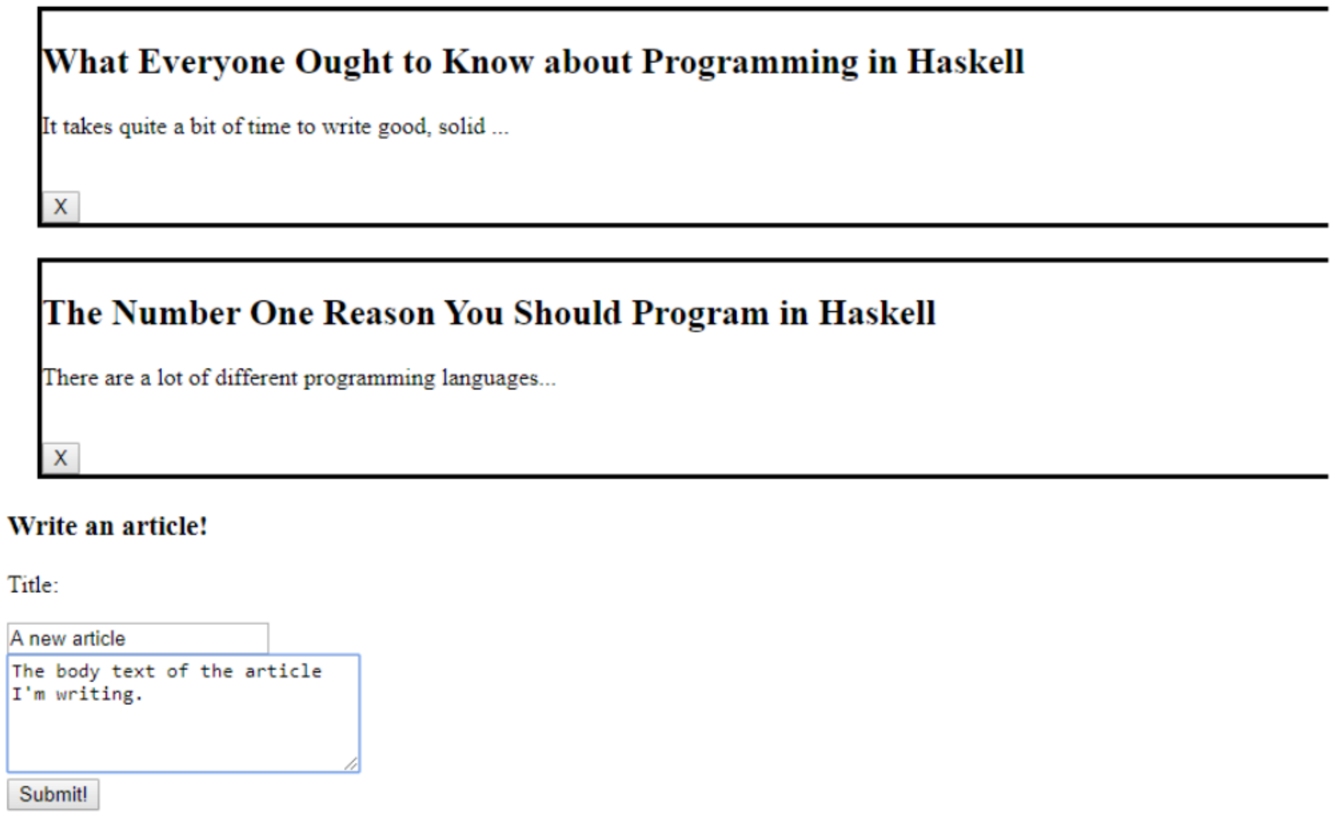
When you submit a new blog post, it should get added to the model's list of articles. This will require adding a new message type, as well as updating the model to keep track of the "in-progress" new article. Your article IDs should increment each time. So the first new article should get ID 6, then 7, and so on.

The previews section should only display the 5 articles with the highest ID values (in decreasing order). However, you should also add a small "x" button at the bottom of each preview that can "remove" that preview, and replace it with the next most recent. So it might help to add a model field for removed preview IDs.

As a final task, you should also disable the submit button unless both the title and the body field are non-empty.

You may wish to consult the Elm docs for help on some of these small issues. Here's a link to the List package, which provides a lot of useful helpers.

Here's an image with my rudimentary submission form. Note the text area is expandable:


What Everyone Ought to Know about Programming in Haskell

It takes quite a bit of time to write good, solid ...

The Number One Reason You Should Program in Haskell

There are a lot of different programming languages...

Write an article!

Title:

A new article

The body text of the article I'm writing.

Lecture 5

Once again, we'll be working in `BlogLandingBasic.elm`. First, modify your `main` function to use an `application` instead of a `sandbox`. To help you prepare this, here are some steps you'll want:

1. The `update` function should also return a command. For now let this be `Cmd.none`.
2. The `init` field should now be a function with three arguments that returns the model and a command. You can ignore the arguments.
2. The `view` function needs to produce a `Document`, instead of just `Html`. Add a record that contains a title and then your existing body `Html`, as shown in the slides.
3. Add a `subscriptions` function, which can give `Sub.none` as its result for now.
4. Add a dummy message constructor, which you can send for `onUrlRequest` and `onUrlChange`.

Your next task is to create a random title generator. At the bottom of the file are three lists. One of these is `titleTemplates`, which contains strings that allow you to substitute two more adjectives. Then `titleAdjectives` is a list of possible adjectives. Then `titleNumbers` is a list of number strings. Create a `Generator` that comes up with new article title by selecting three random indices, one from each list. Then select the string elements from the lists, and combine them with `makeTitle` to return this string. Add a button on the article submission form to generate a random title and substitute it into the title box. Add whatever other messages and commands you need in order to do this.

Take a look at the Random Documentation if you need any help.

Now, we want to store a `Posix` time for each of our articles for when it was published. Add a new field `articlePublishedAt` to the `Article` record with type `Posix`. You can use these values for the original 5 articles:

```
millisToPosix 1570318123000
millisToPosix 1570398123000
millisToPosix 1570478123000
millisToPosix 1570558123000
millisToPosix 1570638123000
```

Now, save the current time in some part of your model with the help of a subscription. Update the current time every minute. And then whenever you create a new `Article`, use the current time as its `Posix` value.

Lecture 6

To start off, generate our initial schema types by going to the root project directory (not the frontend root), building our Haskell code, and running the Elm bridge executable:

```
stack build
stack exec make-elm-types
```

This includes the alterations we made in the screencast, but the resulting types file still won't work because of the `Posix` JSON issues we pointed out. Demonstrate this by importing the `SchemaTypes` file into `BlogLandingBasic.elm` and trying to load it in your browser through `elm reactor`. (Note you should delete the existing Article definition in the landing page file, and then change each `articleId` field so that it says `articleAuthorId`). You should see that it can't find `jsonDecPosix` and `jsonEncPosix`.

To fix help fix this, we have another file `PosixJson.elm`. It contains a couple dummy implementations of `jsonDecPosix` and `jsonEncPosix`. You'll fill these in properly in the next lecture's exercises. But for now, observe that you can make your landing page compile again by adding the following import to `SchemaTypes.elm`:

```
import PosixJson exposing (..)
```

However, we would rather get this line automatically! Go the `PracticalHaskell/app/MakeElmTypes.hs` file. Modify the `moduleHeader` expression to include this import. Now regenerate the `SchemaTypes` file and observe that it compiles without any extra modification!

As we move forward, we'll want more supporting items from our Haskell types. Add extra definitions to make Elm types for the following Haskell types:

```
LoginInfo
LoginResponse
ArticleReaction
ReactionType
Metadata
```

You'll need to add one more type alteration. The `LoginResponse` type uses `Int64`, but we'll just want `Int` on the Elm side. Update the `alterKeyTypes` function to account for this change in a separate case. Your pattern match constructor should look a lot like the existing one, except you won't need a `TyApp` wrapper! Then re-build and re-generate your `SchemaTypes` file.

Now consider that some of our endpoints will return database entities, which match a key with the object's other fields. We want to generalize the idea of an `Entity` without putting Persistent's `Entity` wrapper through out Elm Bridge machinery. To experiment, add a `type alias` at the top of your `SchemaTypes` file for a parameterized `Entity` type. It should be a record with an `entityKey` field that is an `Int`, as well as an `entityValue` field that uses the parameterized type. Confirm this compiles, and then move the definition over to `MakeElmTypes.hs` in the `moduleHeader`, so it gets automatically generated!

To test your code for this lecture, load up the `ElmBridgeTests` module in your browser. If this compiles without any errors, you're good!

Lecture 7

To start off, let's make some proper implementations for parsing `Posix` time. Go to the `PosixJson` module and implement `jsonDecPosix` and `jsonEncPosix`. Our Postgres database stores timestamps in ISO8601 format, for example `2019-06-01T12:00:00-00`. Elm's time package generally takes the opinion that we shouldn't use this format for time. So it's up to us to write our own instances to go between ISO8601 and `Posix`, which is essentially the number of milliseconds since the Unix epoch (January 1st, 1970).

There are a few helpers already in this file. For encoding, you'll mainly want to rely on `getIso8601CalendarElems`. Don't forget to use `zeroPad`, which will ensure your 1-digit numbers get a 0 in front. For decoding, you'll eventually want the reverse helper function, `millisFromCalendarElems`. Those two functions should keep you from needing to mess around with anything too intricate in terms of calendar functions. In the decoding process, you'll generally want to use the `String.split` function to break the parsed string into reasonable pieces that will give you the number for each calendar component.

We don't really care about timezone information, but you should be able to handle inputs that contain it. Your hint here is that we only care about the first 19 characters of the total string! When you encode your times, it's very important that they contain the extra `-00` like in the example above. Otherwise our Haskell server won't be able to parse them!

For the next part, remember the Entity wrapper you implemented in the last lecture's exercises? In `SchemaTypes`, write two functions that will generalize the process of encoding and decoding an Entity in JSON. Their signatures should look like this:

```
encodeEntity: (a -> Value) -> Entity a -> Value
decodeEntity: Decoder a -> Decoder (Entity a)
```

The entity JSON structure should be multi-layered. The top level should have two fields, `key` and `value`. The `key` should refer to the integer, and the `value` should contain the JSON for the embedded element.

Test your code by loading up the `JsonTests` module in your browser!

When the tests pass, you should move the `Entity` conversion code to get automatically generated in the header as well! It shouldn't be too many lines of code!

Lecture 8

In these exercises, we're going to start working on a proper implementation of a couple of our blog pages. We'll do one example of a get request, and one example of a post request.

To start, go into `BlogLandingV2.elm`. We're going to fetch the five most recent articles and render previews for them on this page. Bring over the code you wrote in `BlogLandingBasic.elm` for rendering a preview of an article, except don't worry about the "remove preview" functionality. (Also don't worry about article submission, we'll do another version of that later).

Now modify the `init` portion of our main application to send a command requesting the 5 most recent articles. The URL should use the path `/api/articles/newest?limit=5`. You should parse the JSON response into a list of articles to place into your model. Note that the endpoint returns `Entity Articles`! So use the entity decoder you made last lecture.

In the Haskell part of the project, we have a server* in `src/BlogServer.hs` that will serve as a backend for the rest of this module. Rebuild the Haskell with `stack build`, and then run the executable with `stack exec run-blog-server` in the background, which will live at `localhost:8080.`

Add any necessary messages to your Elm code, and change your `update` function as necessary to update your page! You'll want at least one message for a successful query and one for a failed query.

For your next task, take a look at `LoginPage.elm`. It contains a small login form, where a user will enter their email and password. The existing model contains the `LoginInfo` from the form, as well as an indicator on the last login attempt. When they press the `Submit` button, you should send a POST request to `/api/users/login` with the `LoginInfo` as the body. Parse out the `LoginResponse` as JSON. Change the model's indicator based on the success or failure of the login attempt. Remember the following correct email/password combinations.

```
"Santa@Washington.com" "christmas"
"Romaine@Trent.com" "lettuce"
```

For now, continue to verify your results by running `elm reactor` and opening different pages. As we get towards the end of the module, we'll link things together better with `elm make`!

* This server has a few more modifications from the servers we built in module 2 to make it compatible with Elm. We'll talk about a couple of these in the coming lectures. But for the more esoteric updates, there are some comments at the bottom of that file explaining what they are. Read them if you're curious!

Lecture 9

After your work from the last module, there should now be three separate pages, `LoginPage.elm`, `BlogLandingV2.elm`, and then `ArticlePage.elm` which has a text entry box to load a single article. Your task for this exercise is combine these three into `Multipage.elm`.

When accessing this application, the root path / should take a user to the landing page. You should also use this as the starting page no matter what. The path `/login` should take them to the login page. And the path `/blog/{article-id}` should take them to a page for viewing a single article with that ID. Any other path (including an invalid article ID) should show the 404 page (there's a simple view for this in the file already). To start, the file contains a simple model with the navigation `Key` and a `Route` type of our pages.

You'll need to fill in `routeParser` and `parseUrl`, though you won't really need anything beyond what's in the slides. You should then display the proper view for each of these pages, adding information to the model as needed.

You should build the following links into your application:

1. Clicking on a preview from the landing page should go to that article's page (load the newest articles as a command at the start).
2. The article page's should have a link at the bottom back to the landing.
3. Add a login link at the top of landing page. It should bring the user to the login page.
4. When the user successfully logs in, they should advance to the landing. Save the login response in the model as a `Maybe` field! Remember that a successful login should return the user's ID and a cookie! So when the user is logged in, add a portion near the login link showing their current logged in ID!

You'll need to add a lot of different messages and model fields from your different pages! Don't worry if everything gets a little messy with cramming all this information into one file. In the next lecture, we'll talk about how we can organize it in a better, systematic way.

Lecture 10

In the `src/final` directory, you'll find the starter version of our blog application from the screencast. It has four different pages, the `Landing` page, the `Article` page, the `Login` page, and the `NotFound404` page. Most of the groundwork is already laid out in `final/App.elm`. However, it remains for you to take your code from `Multipage.elm` and break it up into the different model and view files throughout the `final` subdirectories. Your application should work the same as it did before!

Here are a few considerations. Remember that it's valid for the `App` level functions to intercept certain messages that might otherwise go to the individual page update functions. For example, you might want to store the `LoginResponse` at the global app level. But you could also store it in the model for each individual page that needs it. In the former case, you might need to provide it as an additional parameter to the `view` and `update` functions, or else supply the `AppModel` instead of the individual view model.

When you're done, you should add a new page, `SubmitArticle`. This page should allow a user to create a new article, but only if they are logged in. If they aren't logged in, the page should display a message telling them to do so and giving a link back to the `Login` page. The files SubmitArticleModel.elm and SubmitArticleView.elm exist, but they are empty!

There should be a normal input for the title of the article, and a text area for the body. Write a subscription to get the current time every minute. Bring this field into the model for the submit page and use it to populate the `publishedAt` field for the new article. When the user presses a `Submit` button, you should send a post request to `/api/articles/create` creating the article in your database, and then go back to the landing page. The previews on the landing page should now refresh and show your new article!

Following our organizational pattern, this page should get its own files `SubmitArticleModel.elm` and `SubmitArticleView.elm`. Add its model and message types to those of the app as a whole. Add a new route for it at `/blog/submit`. Finally, update the view and update functions in `src/final/App.elm` to include it.

For all of your URLs, you should use `hostString` and `portNum` from `Model.elm` for your URLs! You'll need to change them all in the next set of exercises!

For a bonus challenge, you can implement JWT based authentication on the article creation endpoint, so that you need to include the logged in user's cookie when making the post request. You'll need to learn how to use Elm's `Http.request` function to add the cookie to the header. This function is a bit lower level than `Http.get` and `Http.post` but doesn't really force you to use any new concepts.

Lecture 11

For your first task, you should make a couple changes to the way we initialize our final application. We have a couple small hacks so that we always re-route to the landing page of the application. We should now un-do these. In `App.elm`, instead of pushing to `"/"` in the `init` function, push to the URL argument (convert it to a string first). Then in `Model.elm`, in `initModel`, parse the URL to get the `currentPage` instead of using `Landing` automatically.

Next, you should go into `src/BlogServer.hs` and ensure that our server can serve the `SubmitArticle` page you made in the last lecture. You'll need to modify `StaticContentApi` and `staticServer` to serve the index at the proper path. Verify this works by re-compiling Elm and running the blog server. The first command should be run from the `frontend` directory, and then you should back out into the root project directory and building the Haskell code.

```
elm make src/final/App.elm --output static/app.js
cd ..
stack build
stack exec run-blog-server
```

For your next task, tweak the CSS for your application. Go into `static/main.css`, and observe the `.article-preview` class, which contains some of the effects you might currently be using for an article preview. You can add more effects if you like. Now add this class to your preview elements on the landing page using the `class` function from `Html.Attributes`. You should then be able to remove the other style elements you put there. Then when running your server locally, observe that the styles work!

Finally, try out deploying the site on Heroku! There are a couple things you'll need to do. First, the URLs for making any kind of API call will need to change their host name to use the URL for your app on Heroku instead of `localhost`. So in Model.elm change the `hostString` expression use that string, and change the port value to `Nothing`. Second, go into your Heroku dashboard, and change it so that your Heroku app uses `module-3` as the deployment branch instead of `module-2`. Finally, you'll need to push your `module-3` branch to Github*. Ensure you add the generated `app.js` file!

When your build finishes and deploys, you should be able to go to `http://$APP_NAME.herokuapp.com/`, see your landing page, and navigate around the app!

*Assuming your app is still configured to do Github deploys. You can also deploy it manually using the same techniques we saw in module 2.

Lecture 12

There's no more content for this module, but there are still tons of ways you can improve on your blog project and learn more about connecting Elm and Haskell in the process! Here are just a couple ideas that use some of the existing database elements we have:

1. Add an archive page allowing a user to see links to all the articles in the database
2. Display information about the author with every article.
3. Allow users to comment on articles. This requires adding a submission UI, a display UI for the comments and hooking them up with the backend! It's a really good exercise for going through all the steps of our process.
4. Allow users to react to articles using the `ArticleReaction` type.
5. Allow advanced searching and filtering of articles. For instances, see all the articles by a particular user. Or for something more difficult, find all the articles mentioning particular keywords. These features could require adding more features to the backend as well!
6. Add more CSS styling in `static/main.css`!

And here's another challenge for the deployment side, if you're brave enough! We might observe that it's undesirable to commit a generated file (`app.js`) to our Github repository. We'd like to compile this on our Heroku machine. If you want, you can go onto your heroku machine terminal and install Elm using the Linux instructions. Then you can add a few lines to the `Procfile` to compile your Elm code before running the Haskell server!

For a super challenge, you can fork the Haskell Heroku buildpack, and update the `bin/compile` file to also install Elm. Try following the code in that file which installs Stack, copy it, and change the URLs to use Elm instead! Then if you change your Haskell application to use your own forked buildpack, it will automatically have Elm installed!