Snap for Beginners

Chris Biscardi

Contents

Chapter 1

Getting Started

This chapter will be about getting started with Snap. We will begin by giving some backround and progress through a working scaffolding app. A passing familiarity with Haskell development and the command line is assumed, but a brief rundown is given for those that do not have the background.

1.1 How to Read this Book

This book strives to place example code in your hands rather than lead you through a project. I suggest that you pick an idea such as a simple blog, where you input some data in a form and can render posts at different urls. Then you can use the code examples in the book to build up your project little by little.

The chapters may depend on each other (The Authorization Snaplet references the PostgreSQL chapter for example) and you are encouraged to bounce around while reading the book.

1.2 What Snap is

Snap is a web framework for the Haskell language that is most similar to Express or Sinatra. As contrast, Yesod (Another Haskell framework), can be viewed as a more Rails-like framework.

From SnapFramework.com:

Snap is a simple web development framework for unix systems, written in the Haskell programming language. Snap has a high level of test coverage and is well-documented. Features include:

- A fast HTTP server library
- A sensible and clean monad for web programming

- An HTML-based templating system for generating pages

Snap also contains `snaplets`, which are modular units of stateful code that are usable between applications. For example, there are snaplets for heist, authentication and Postgres. Snaplets also come with some nice extras such as a unified configuration format.

One of the nice things about the Haskell web framework landscape is that many of the components are replaceable with components from other projects. We will not be going into that in this book but you can read more here

1.3 Where to Get Help

There are a few places to get help or submit issues. This book is availible in source form on github for issues. Check out the discussions of the book and topics. Also useful are the official snap website and irc channel (#snapframework).

1.4 Install Haskell

By far the easiest way to install Haskell is the Haskell Platform, which is availible from http://www.haskell.org/platform/. Some package managers also include haskell-platform as well, including homebrew and apt-get. This book is written using the 2013.2.0.0 Haskell Platform, which includes GHC 7.6.3.

1.5 A Note on Sandboxes

It is a good idea to separate Haskell projects from each other if there are multiple Haskell projects on a machine. This makes it easier to manage dependencies and enables a simpler workflow.

There are a few options for creating environments, we will go over `cabal sandbox`.

After installing Haskell Platform, we need to upgrade cabal to ~v1.18. Run:

```
cabal update
cabal install cabal-install
```

Then you will see something like this at the end:

```
Installing executable(s) in
/Library/Haskell/ghc-7.6.3/lib/cabal-install-1.18.0.2/bin
Installed cabal-install-1.18.0.2
Updating documentation index /Library/Haskell/doc/index.html
```

According to the code we just ran, the updated `cabal` is in `/Library/Haskell/ghc-7.6.3/lib/cabal-install-1.18.0.2/bin` so we need to add that to our path:

```
export PATH="/Library/Haskell/ghc-7.6.3/lib/cabal-install-1.18.0.2/bin:$PATH"
```

Note that this path will be different based on your system.

1.5.1 cabal sandbox

Now that we have a recent cabal version, we can use `cabal sandbox`. For a deeper understanding of `cabal sandbox`, refer to this post.

```
cabal sandbox init
```

will create a directory and a file in the current directory that will hold packages and other settings for the sandbox.

```
.cabal-sandbox
cabal.sandbox.config
```

Now, when we run `cabal install` for this project, the executable will be in `.cabal-sandbox/bin/`. For example, if we are in `code/heist-app` (one of the sample projects that comes with the book) and we created a sandbox in `heist-app`, we can install and run the executable without affecting our other projects as such:

```
cd code/heist-app
cabal sandbox init
cabal install
.cabal-sandbox/bin/heist-app
```

which will result in

```
Initializing app @ /
Initializing heist @ /
...loaded 7 templates from /heist-app/snaplets/heist/templates
Initializing CookieSession @ /sess
Initializing JsonFileAuthManager @ /auth

Listening on http://0.0.0.0:8000/
```

1.6 Install Snap

Installing Snap is easy because of cabal, Haskell's package manager. Just run:

```
cabal install snap
```

Chapter 2

Scaffolding a New Project

– note: The code for a scaffolding app is already present in code/scaffold-app

Now that we have Snap installed we can use the CLI to scaffold a new project.

Create a new directory:

```
mkdir scaffold-app
```

Then enter the directory and initialize a “default” project:

```
cd scaffold-app
cabal sandbox init
snap init default
```

We now have a default Snap app with a basic user authentication scheme. Install the app by running:

```
cabal install
```

This uses the `scaffold-app.cabal` file to install dependencies. We can run it the app by running:

```
.cabal-sandbox/bin/scaffold-app
```

The server defaults to port 8000, so by navigating to `localhost:8000` we should see a running instance of the app. From the homepage, we can create a user and then log in to see the demo website.

2.1 code/scaffold-app

The Scaffolding code distributed with this book (in `code/scaffold-app`) is modified in that it contains additional comments. The two files we are concerned with are `src/Application.hs` and `src/Site.hs`. `src/Application.hs` includes some basic setup code for the Authorization, Session and Heist Snaplets. We will go into more detail with Snaplets in later chapters.

`src/Site.hs` is where most of our development will happen. It includes the routing, initialization and some route handlers. The handlers can be split out into other files, but we will keep them in a single file for now.

2.2 src/Application.hs

`src/Application.hs` starts off with something that tells our compiler that we are using an extension to the haskell language[1]. In this case, it is the `TemplateHaskell` extension. This won't actually affect us much, as the only place we use Template Haskell is in the call to `makeLenses` later in this file.

```haskell
{-# LANGUAGE TemplateHaskell #-}
```

The next bit of code defines the module for this file. We will use this in our `src/Site.hs` to import this file. In this case, `import Application` is what we will write.

```haskell
module Application where
```

The imports list is next and defines some of the modules we'll be using in our code in this file. `Control.Lens` will be used as part of our call to `makeLenses` and the rest are Snaplet modules, since we are defining some of our Snaplet code in this file.

```haskell
import Control.Lens
import Snap.Snaplet
import Snap.Snaplet.Heist
import Snap.Snaplet.Auth
import Snap.Snaplet.Session
```

Next is the most important part of this file, our `App` datatype[2]. This defines the Snaplets we will be using as part of a data structure so that we can initialize and access them later on in `src/Site.hs`.

We are using the Heist (`_heist`), Session (`_sess`) and Authentication (`_auth`) Snaplets. Each comes with it's own type declaration so that we can be assured that we are putting the right Snaplets in the right places when we initialize our app.

[1] This is a Language Pragma. There is plenty of information on them online if you search for "haskell language pragmas".

[2] The way we are writing this datatype is called "Record Syntax".

```haskell
data App = App
    { _heist :: Snaplet (Heist App)
    , _sess :: Snaplet SessionManager
    , _auth :: Snaplet (AuthManager App)
    }
```

`makeLenses` is next. Basically, this automatically creates getters/setters and some other things for us so we don't have to write a bunch of boilerplate. We are calling it on our `App` datatype, so when we use our Snaplets in `src/Site.hs` we can call them without the underscores in front (ie: `_heist` becomes `heist`).

```haskell
makeLenses ''App
```

Writing an instance for our Heist Snaplet allows us to write less boilerplate code. If we didn't write this instance, we would have to write `with heist dosomething` whenever we wanted to render a template. The instance basically tells the compiler how to access the Heist Snaplet when we are in a route, so it can figure things out for us.

```haskell
instance HasHeist App where
    heistLens = subSnaplet heist
```

This is a simple alias. `AppHandler` and `Handler App App` mean exactly the same thing. If we were writing a handler for a Snap route, either one of these would be acceptable as the type signature.

```haskell
type AppHandler = Handler App App
```

2.3 src/Site.hs

2.3.1 Language Pragma

`Site.hs` starts off with an extension to the Haskell language[3]. This one makes it easier to work with string literals in our source code files. Typically, a String literal is of type `String`. Using `OverloadedStrings` allows us to write string literals (a string literal is `"like this"`) of type `Text`.

```haskell
{-# LANGUAGE OverloadedStrings #-}
```

[3] This is a Language Pragma. There is plenty of information on them online if you search for "haskell language pragmas".

2.3.2 Module Declaration and Imports

Then we declare our module (`Site`) and a few imports. This includes the `src/Application.hs` module, which is imported as `import Application`.

```
module Site
  ( app
  ) where

------------------------------------------------------------------------------
import           Control.Applicative
import           Data.ByteString (ByteString)
import qualified Data.Text as T
import           Snap.Core
import           Snap.Snaplet
import           Snap.Snaplet.Auth
import           Snap.Snaplet.Auth.Backends.JsonFile
import           Snap.Snaplet.Heist
import           Snap.Snaplet.Session.Backends.CookieSession
import           Snap.Util.FileServe
import           Heist
import qualified Heist.Interpreted as I
------------------------------------------------------------------------------
import           Application
```

2.3.3 handleLogin

Next, we set up the rendering of the login form template (with errors).

```
handleLogin :: Maybe T.Text -> Handler App (AuthManager App) ()
handleLogin authError = heistLocal (I.bindSplices errs) $ render "login"
  where
    errs = maybe noSplices splice authError
    splice err = "loginError" ## I.textSplice err
```

The type signature breaks down into two pieces split by ->. The first:

```
Maybe T.Text
```

is the type of the argument to this function. It says that we might get some text or we might get nothing. The second type:

```
Handler App (AuthManager App) ()
```

is what the function returns. In this case it returns a Snap handler that uses the Authentication Snaplet. A basic handler (without Authentication) has the type `Handler App App ()`.[4]

The next part starts the function definition.

```
handleLogin authError = heistLocal (I.bindSplices errs) $ render "login"
```

`handleLogin` takes one argument, which we've named `authError`. `heistLocal` is a function that lets us bind custom splices[5] to be used in the `"login"` template and then use them.

`errs` defines our custom splice:

```
errs = maybe noSplices splice authError
```

`maybe` takes a default values (`noSplices` in this case), our custom splice (defined as `splice` on the line below) and the `authError`. If the `authError` is `Nothing` (no errors) we use `noSplices`, otherwise we use our custom splice.

```
splice err = "loginError" ## I.textSplice err
```

Here we define our splice. If the `authError` exists it gets passed to this function as `err`. We then bind the name `"loginError"` to our `textSplice`, which we created from the `err` text. The splice we just created displays the error using the tag `<loginError/>` in our heist templates (specifically `snaplets/heist/templates/_login.tpl`).

2.3.4 handleLoginSubmit

`handleLoginSubmit` handles retrieving values from a login form submission using the Authentication Snaplet's `loginUser` function.

```
handleLoginSubmit :: Handler App (AuthManager App) ()
handleLoginSubmit =
    loginUser "login" "password" Nothing
              (\_ -> handleLogin err) (redirect "/")
  where
    err = Just "Unknown user or password"
```

`loginUser` takes the names of the username and password form fields (`"login"` and `"password"` in our case), the "Remember Me" field (In our case, `Nothing` since we aren't using one), a failure function and a success function.

Our failure function is

[4] More on this in the Authentication and Routing chapters.

[5] More on splices in the Heist chapter

```
(\_ -> handleLogin err)
```

Which is an anonymous function that takes anything (the _ is Haskell for “we don’t care what this argument is”, in this case because we aren’t using any arguments) and returns `handleLogin` with the error value `err`.

`err` is `Just "Unknown user or password"`. We put `Just` in front of the value because as we saw before, `handleLogin` takes `Maybe T.Text` as an argument. The two possible values being `Nothing` and `Just "some text"`.

The success function, `(redirect "/")` simply redirects a successful login to the homepage.

2.3.5 handleLogout

`handleLogout` uses the Authentication Snaplet’s `logout` function and then redirects the user to the homepage.

```
handleLogout :: Handler App (AuthManager App) ()
handleLogout = logout >> redirect "/"
```

The >> operator sequences the two functions, discarding any values produced by `logout`.

2.3.6 handleNewUser

`handleNewUser` splits a request into two different functions for `GET` and `POST`.

```
handleNewUser :: Handler App (AuthManager App) ()
handleNewUser = method GET handleForm <|> method POST handleFormSubmit
  where
    handleForm = render "new_user"
    handleFormSubmit = registerUser "login" "password" >> redirect "/"
```

For a `GET` request, we use `handleForm`, which just renders the `"new_user"` template.

For a `POST` request, we use the Authentication Snaplet’s `registerUser`. `registerUser` takes the username and password fields (In our case `"login"` and `"password"`).

2.3.7 Routing

Our routes are defined next. `with auth` is how we say “this route is going to be using the Authentication Snaplet’s functions”.

```
routes :: [(ByteString, Handler App App ())]
routes = [ ("/login",    with auth handleLoginSubmit)
         , ("/logout",   with auth handleLogout)
         , ("/new_user", with auth handleNewUser)
         , ("",          serveDirectory "static")
         ]
```

We also serve static files from the `static` folder.

2.3.8 Initialization

The most involved code is the `app` initialization code.

```
app :: SnapletInit App App
app = makeSnaplet "app" "An snaplet example application." Nothing $ do
    h <- nestSnaplet "" heist $ heistInit "templates"
    s <- nestSnaplet "sess" sess $
           initCookieSessionManager "site_key.txt" "sess" (Just 3600)
    a <- nestSnaplet "auth" auth $
           initJsonFileAuthManager defAuthSettings sess "users.json"
    addRoutes routes
    addAuthSplices h auth
    return $ App h s a
```

First we say that `app` will hold our initialized `App` (from `src/Application.hs`). `makeSnaplet` takes an id (`"app"` in this case), a description, a `Maybe (IO FilePath)` (which we'll just set to `Nothing` since this isn't a packaged Snaplet) and an Initializer.

In this case our Initializer is our `do` statement.

Common to all of the Snaplets we are about to initialize is `nestSnaplet`. `nestSnaplet` takes a root url for any routes defined in the Snaplet, the name of the Snaplet as defined in `src/Application.hs` without the underscore (also known as a Lens because we ran `makeLenses` on it), and the Snaplet specific initializer function.

The first thing we do is initialize our Heist Snaplet.

```
h <- nestSnaplet "" heist $ heistInit "templates"
```

Using a call to `nestSnaplet` we pass in: The root path for the routes (`""`), `heist` (which is the Lens value we made from _heist) and the result of `heistInit "templates"`, which is our Heist initializer. `heistInit`'s argument is the folder that we are storing our templates in (in this case the Heist Snaplet is located in `snaplets/heist` and our templates are in `snaplets/heist/templates` so we pass in `"templates"`).

The next Snaplet to be initialized is the Session Snaplet. This will be used with the Authentication Snaplet to give us sessions.

```
s <- nestSnaplet "sess" sess $
       initCookieSessionManager "site_key.txt" "sess" (Just 3600)
```

Once again we call `nestSnaplet` with the base route and Lens value (`sess` because we used `_sess` in `src/Application.hs`). We then initialize a Cookie-based backend with `initCookieSessionManager`.

`initCookieSessionManager` takes an encryption key (generated for us in `site_key.txt`), a name (`"sess"`) and a session timeout for replay attack protection (`Just 3600`).

The Authorization Snaplet is initialized next.

```
a <- nestSnaplet "auth" auth $
       initJsonFileAuthManager defAuthSettings sess "users.json"
```

Again a call to `nestSnaplet`. The Authentication Snaplet has support for multiple backends, such as a flat json file or PostgreSQL. In this case, we initialize a JSON file with the default authentication settings (`defAuthSettings`), the Session Snaplet we just initialized (`sess`) and a filename to store the data in (`"users.json"`).

`defAuthSettings` contains a few fields:

```
asMinPasswdLen = 8
asRememberCookieName = "_remember"
asRememberPeriod = Just (2*7*24*60*60) = 2 weeks
asLockout = Nothing
asSiteKey = "site_key.txt"
```

Currently, `asMinPasswdLen` is not used by the Auth Snaplet. More information about these fields is availible in the Snap docs on Hackage.

Finally:

```
addRoutes routes
addAuthSplices h auth
return $ App h s a
```

We add our routes, add some splices from the Auth Snaplet and return an instance of the `App` definition from `src/Application.hs` that includes the heist (h), session (s) and auth (a) instances.

2.4 snaplets/heist/templates/

This folder holds our Heist templates. `snaplets/heist` is the base directory for the Heist Snaplet and templates is a directory that has been created so that Heist has access to our templates.

2.4.1 _login.tpl

The _login template is rendered as a sub-piece of the login.tpl template.

```
<h1>Snap Example App Login</h1>

<p><loginError/></p>

<bind tag="postAction">/login</bind>
<bind tag="submitText">Login</bind>
<apply template="userform"/>

<p>Don't have a login yet? <a href="/new_user">Create a new user</a></p>
```

<loginError/> is a splice we created in handleLogin in our src/Site.hs file. The splice, as we defined it, shows the error message if it exists.

We have two <bind> tags next. These function a bit like defining variables and are used later on in our template. Specifically in the userform section specified by the apply tag below.

The next line is an <apply> tag. It is used to render userform.tpl as part of this template.

2.4.2 _new_user.tpl

```
<h1>Register a new user</h1>

<bind tag="postAction">/new_user</bind>
<bind tag="submitText">Add User</bind>
<apply template="userform"/>
```

_new_user.tpl is similar to _login.tpl. The only difference is that the values of the <bind> tags are different. This shows how a template can be modified by the context in which it is rendered.

2.4.3 base.tpl

base.tpl is the base outline of our templates. It includes all the scaffolding such as <html>, <head> and <body>.

```
<html>
  <head>
    <title>Snap web server</title>
    <link rel="stylesheet" type="text/css" href="/screen.css"/>
```

```
  </head>
  <body>
    <div id="content">

      <apply-content/>

    </div>
  </body>
</html>
```

Inside of the `<div id="content">` is `<apply-content>`. This allows us to use `base.tpl` as a wrapper for whatever content we want, as we will see in `index.tpl`.

2.4.4 index.tpl

The `index.tpl` template is a little more interesting. The first tag applies the base template. Anything inside the `<apply template="base">` tag will go where we wrote `<apply-content>` in `base.tpl`.

```
<apply template="base">

  <ifLoggedIn>
    <p>
      This is a simple demo page served using
      <a href="http://snapframework.com/docs/tutorials/heist">Heist</a>
      and the <a href="http://snapframework.com/">Snap</a> web framework.
    </p>

    <p>Congrats!  You're logged in as '<loggedInUser/>'</p>

    <p><a href="/logout">Logout</a></p>
  </ifLoggedIn>

  <ifLoggedOut>
    <apply template="_login"/>
  </ifLoggedOut>

</apply>
```

`<ifLoggedIn>` is one of the Auth Splices we added in `src/Site.hs` when we initialized our app. The content inside this tag only renders if the user is logged in.

`<loggedInUser/>` is similar, but it displays the username of the logged in user.

`<ifLoggedOut>` is also an Auth Splice. It renders it's content if the user is not logged in. In this case, it renders the `_login.tpl` template.

2.4.5 login.tpl

The `login.tpl` template is super simple. It applies the base template and uses `_login.tpl` as the content.

```
<apply template="base">
  <apply template="_login"/>
</apply>
```

2.4.6 new_user.tpl

The `new_user.tpl` template is very similar to `login.tpl`. It applies the base template and uses `_new_user.tpl` as the content.

```
<apply template="base">
  <apply template="_new_user" />
</apply>
```

2.4.7 userform.tpl

`userform.tpl` uses the content of the `<bind>` tags from the other templates. To access the value of the bind tag, we use `${tag}`. In the case of `postAction` it looks like `${postAction}`.

```
<form method="post" action="${postAction}">
  <table id="info">
    <tr>
      <td>Login:</td><td><input type="text" name="login" size="20" /></td>
    </tr>
    <tr>
      <td>Password:</td><td><input type="password"
      name="password" size="20" /></td>
    </tr>
    <tr>
      <td></td>
      <td><input type="submit" value="${submitText}" /></td>
    </tr>
  </table>
</form>
```

2.5 Fin

That's it for the default template. From here use the other chapters to learn more about various pieces of Snap. Later in the book we will go over Digestive Functors, which can be used to render and process forms with validation, and Heist, which has more splices (such as Markdown) an Interpreted and a Compiled library.

Chapter 3

Heist Snaplet

The Heist Snaplet handles the rendering of templates. We will be going over Interpreted Heist in this chapter. There is another usage of Heist called Compiled Heist that is higher performance but also slightly more difficult to work with.

3.1 Initializing the Snaplet

As we can see in `code/scaffolding`, and in the scaffolding chapter, we have to first add Heist to our app definition:

```
data App = App { _heist :: Snaplet (Heist App) }
```

Then we write a simple instance so we don't have to write `with heist` in front of all our routes that render templates:

```
instance HasHeist App where
    heistLens = subSnaplet heist
```

Here we declare an instance of `HasHeist App`, which is to say that we are telling the compiler that our `App` does indeed have an instance of Heist accessible. We then define `heistLens`, which is the function that will be called to access heist from our `App`, to be `subSnaplet heist`. This is because in our initialization code (as seen in the scaffolding chapter) we define `heist` to be a `subSnaplet` of `App`.

This instance is totally optional, but if we don't write it we will have to prefix the routes that use `heist` with `with heist`.

To finish off the initialization, we will go over how we set up `heist` as a `subSnaplet` to `App`:

```
appInit = makeSnaplet "app" "" Nothing $ do
     h <- nestSnaplet "heist" $ heistInit "templates"
     return $ App h
```

We can see that our app is initialized as `makeSnaplet "app" "" Nothing`, making it a Snaplet. Then, when we return our initialized `App` structure with the initialized `heist` Snaplet.

3.2 Handler Functions

3.2.1 render

`render` renders a template. It is a specialized version of `renderAs`.

3.2.1.1 render usage

```
myhandler :: Handler App App ()
myhandler = render "mytemplate.tpl"
```

3.2.2 heistLocal

`heistLocal` can be used, as seen in `code/scaffold-app/Site.hs`, to use cutomized splices for specific routes.

```
handleLogin authError = heistLocal (I.bindSplices errs) $ render "login"
  where
    errs = maybe noSplices splice authError
    splice err = "loginError" ## I.textSplice err
```

`heistLocal` takes a function that modifies the Heist state (`bindSplices` does this), and a Handler to run. In this case our handler is rendering the `login.tpl` template.

3.2.3 renderWithSplices

We can simplify the above code using `renderWithSplices`. `renderWithSplices` is sugar for the combination of `heistLocal`, `bindSplices` and `render` that we just used. The simplified version:

```
handleLogin authError = renderWithSplices "login" errs
  where
    errs = maybe noSplices splice authError
    splice err = "loginError" ## I.textSplice err
```

`renderWithSplices` takes a template name and some splices and returns a Handler for us that will render the template with the included splices.

3.3 Splices

Bind and Apply are the main splices that come with Heist.

3.3.1 Bind

The Bind tag is used to bind a value to a tag, such as:

```
<bind tag="postAction">/login</bind>
<bind tag="submitText">Login</bind>
```

After using the `<bind>` tag we can use the values later on in our template by using `${tag}` syntax.

```
<form method="post" action="${postAction}">
  <input type="submit" value="${submitText}" />
</form>
```

This form with those bound tags would render as:

```
<form method="post" action="/login">
  <input type="submit" value="Login" />
</form>
```

3.3.2 Apply

The Apply tag is used to apply templates as to the current template.

Given `something.tpl` as such:

```
<h1>In Something Template</h1>
<apply template="_something"/>
```

and a template called `_something.tpl` (the underscore is purely convention for a template we won't use when calling `render`, but is used as a sub-template) as such:

```
<p>Content from _something template</p>
```

When we `render "something"` the output will look like:

```
<h1>In Something Template</h1>
<p>Content from _something template</p>
```

3.3.3 apply-content

`apply-content` is used to allow an `apply` tag to wrap content. For example if we have this `base.tpl`:

```
<html>
  <head>
    <title>Snap web server</title>
    <link rel="stylesheet" type="text/css" href="/screen.css"/>
  </head>
  <body>
    <div id="content">

      <apply-content/>

    </div>
  </body>
</html>
```

We can use it as the "container" like this:

```
<apply template="base">
  <h1>All Your Base</h1>
</apply>
```

The rendered template will look like this:

```
<html>
  <head>
    <title>Snap web server</title>
    <link rel="stylesheet" type="text/css" href="/screen.css"/>
  </head>
  <body>
```

```
    <div id="content">
      <h1>All Your Base</h1>
    </div>
  </body>
</html>
```

Chapter 4

Routing

Snap allows writing routes in fairly familiar way. It then takes these routes in the `addRoutes` function and turns them into a trie that gives us O(log n) dispatching time.

4.1 Route Definitions

The route definitions from `code/routing-app/src/Site.hs`:

```
routes :: [(ByteString, Handler App App ())]
routes = [ ("/logins",    with auth handleLoginSubmit)
         , ("/logout",   with auth handleLogout)
         , ("/new_user", with auth handleNewUser)
         , ("",          serveDirectory "static")
         ]
```

which we then add in our app initialization:

```
addRoutes routes
```

We can see that we define our routes with a list of tuples. Each tuple consists of a URL fragment and a function.

Similar routes are combined using Control.Applicative's Alternative class (<|>). To get some basic intuition for how <|> works, we can run some experiments in `ghci`.

Enter ghci:

```
ghci
```

Import `Control.Applicative`

```
:m Control.Applicative
```

Now we can use the <|> operator to test. In this example > is used to represent the prompt, everything after > is typed into ghci and content without a > at the beginning is the return value of the previous line.

```
>Nothing <|> Just 4
Just 4
>Nothing <|> Just 4 <|> Just 5
Just 4
> Nothing <|> Nothing <|> Just 5
Just 5
>Nothing <|> Nothing <|> Nothing
Nothing
```

We use <|> later in this chapter to match routes based on method.

4.1.1 Parameters

Parameters can be in three places: `rqQueryParams` for the query string, `rqPostParams` for `POST` bodies and `rqParams` for a union of the two previous maps.

4.1.2 URL Parameters

We can also use the `:paramname` form in the route to get parameters from the URI. We'll use a sample handler to echo back the parameter in the url:

```
echoHandler :: Handler App App ()
echoHandler = do
  param <- getParam "echoparam"
  maybe (writeBS "must specify echo/param in URL")
         writeBS param
```

`getParam` will get the parameter from either a GET or POST request and then we respond with either "`must specify echo/param in URL`" if there is no `param` or the value of the `param`. Here is the route we use:

```
routes = [("/echo/:echoparam", echoHandler)]
```

It will be used for `/echo/something/` and for `/echo/something/many/things/` but not for `/echo/`. Both times it will respond with "`something`".

4.1.3 ifTop

We can solve this issue with `ifTop`. We can create a second route that will only respond to the base route we define, in this case `/echotwo/parameter`. `/echotwo/` and `/echotwo/something/anythinghere/` will fail.

```
("/echotwo/:echoparam", ifTop echoHandler)
```

4.1.4 method VERB

For additional restriction we can use `method`. `method` allows us to restrict route handlers to specific verbs, such as GET or POST.

We can define two handlers for GET and POST that simply respond with “getHandler” and “postHandler” respectively.

```
getHandler :: Handler App App ()
getHandler = writeBS "getHandler"

postHandler :: Handler App App ()
postHandler = writeBS "postHandler"
```

We can then set up `method GET getHandler`, which will only run GET requests to the `getHandler` we can then chain it with `method POST postHandler` using <|>. Note that this will behave very similar to the example at the beginning of the chapter. Behind the scenes `method` uses `unless` from `Control.Monad` to determine whether or not to “pass” to the next handler.

```
("/getorpost", method GET getHandler <|> method POST postHandler)
```

We could then run curl to test the routes:

```
curl localhost:8000/getorpost
```

should return "`getHandler`" while:

```
curl -XPOST localhost:8000/getorpost -d "stuff"
```

will return "`postHandler`".

4.2 Sending Data Back

There are many ways to send data in the response. A few of them are here. If we use the OverloadedStrings language pragma we can write string literals as below. If we don’t we would have to write the respective pack functions for each data type.

It is important to note that `writeBS` doesn't actually write to the socket, but rather adds to the closure in the Response that *will* be called. This allows us to use multiple calls to `writeBS` in the same handler. The 1.0 release of Snap will be based on streams (using io-streams). A future version of this book will cover that.

4.2.1 writeBS

Writes a ByteString back to the client.

```
writeBS "data here"
```

4.2.2 writeText

Writes Text back to the client.

```
writeText "data here"
```

4.2.3 writeJSON

`writeJSON` is from the Snap.Extras.JSON package and can be used in conjunction with Data.Aeson to more easily write JSON responses. It will set the MIME to `'application/json'` and write the given object into the response body.

If we have a custom datatype and a `ToJSON` instance from `Data.Aeson` we can use writeJ-SON`to send it as a JSON representation. From`code/routing-app/src/Site.hs`:

```
data Person  = Person {
  name :: String
  } deriving (Show)
instance ToJSON Person where
  toJSON (Person s) = object ["name" .= s]
```

and in our route/handler we create a new `Person` and pass it to `writeJSON`:

```
("/json", writeJSON $ Person "me")
```

When we hit `http://localhost:8000/json` we should get:

```
{"name":"me"}
```

Chapter 5

Digestive Functors

Digestive functors are one way to do form processing in Haskell. In this chapter we will build up a sample application to see how to accept and validate form input and render forms and errors with the Digestive Functors package.

Then, we'll move into a deeper exploration and examine all of the possible options digestive functors gives us.

5.1 Building a Digestive Functors Flow

To start off, we need a scaffold. We'll create a new folder named `df-one` and create a new scaffolding app inside it:

```
mkdir df-one
cd df-one
snap init
```

Alternatively, check out the code in `code/digestive-functors/df-one`, which has the completed code.

Our `src/Site.hs` currently looks as such:

```
{-# LANGUAGE OverloadedStrings #-}

------------------------------------------------------------------------
-- | This module is where all the routes and handlers are defined for your
-- site. The 'app' function is the initializer that combines everything
-- together and is exported by this module.
module Site
```

```haskell
  ( app
  ) where

------------------------------------------------------------------------
import           Control.Applicative
import           Data.ByteString (ByteString)
import           Data.Maybe
import qualified Data.Text as T
import           Snap.Core
import           Snap.Snaplet
import           Snap.Snaplet.Auth
import           Snap.Snaplet.Auth.Backends.JsonFile
import           Snap.Snaplet.Heist
import           Snap.Snaplet.Session.Backends.CookieSession
import           Snap.Util.FileServe
import           Heist
import qualified Heist.Interpreted as I
------------------------------------------------------------------------
import           Application


------------------------------------------------------------------------
-- | Render login form
handleLogin :: Maybe T.Text -> Handler App (AuthManager App) ()
handleLogin authError = heistLocal (I.bindSplices errs) $ render "login"
  where
    errs = [("loginError", I.textSplice c) | c <- maybeToList authError]


------------------------------------------------------------------------
-- | Handle login submit
handleLoginSubmit :: Handler App (AuthManager App) ()
handleLoginSubmit =
    loginUser "login" "password" Nothing
              (\_ -> handleLogin err) (redirect "/")
  where
    err = Just "Unknown user or password"


------------------------------------------------------------------------
-- | Logs out and redirects the user to the site index.
handleLogout :: Handler App (AuthManager App) ()
handleLogout = logout >> redirect "/"
```

```haskell
------------------------------------------------------------------------
-- | Handle new user form submit
handleNewUser :: Handler App (AuthManager App) ()
handleNewUser = method GET handleForm <|> method POST handleFormSubmit
  where
    handleForm = render "new_user"
    handleFormSubmit = registerUser "login" "password" >> redirect "/"


------------------------------------------------------------------------
-- | The application's routes.
routes :: [(ByteString, Handler App App ())]
routes = [ ("/login",    with auth handleLoginSubmit)
         , ("/logout",   with auth handleLogout)
         , ("/new_user", with auth handleNewUser)
         , ("",          serveDirectory "static")
         ]


------------------------------------------------------------------------
-- | The application initializer.
app :: SnapletInit App App
app = makeSnaplet "app" "An snaplet example application." Nothing $ do
    h <- nestSnaplet "" heist $ heistInit "templates"
    s <- nestSnaplet "sess" sess $
           initCookieSessionManager "site_key.txt" "sess" (Just 3600)

    -- NOTE: We're using initJsonFileAuthManager here because it's easy and
    -- doesn't require any kind of database server to run.  In practice,
    -- you'll probably want to change this to a more robust auth backend.
    a <- nestSnaplet "auth" auth $
           initJsonFileAuthManager defAuthSettings sess "users.json"
    addRoutes routes
    addAuthSplices h auth
    return $ App h s a
```

We are going to keep the handlers around so that later we can use the `auth` snaplet, which is already set up for us, to secure our form.

5.1.1 Define a new Datatype: Tweet

The very first thing we need is a new datatype to define the data we will be capturing in our form. In our case, Twitter is down and the world is in a panic, so we will create a `Tweet` in a new file `src/Twitter.hs` to build a new Twitter.

We will also include some module boilerplate so we can export our awesome new datatypes and functions.

```
{-# LANGUAGE OverloadedStrings #-}
module Twitter
(Tweet
  ) where

import qualified Data.Text as T

data Tweet = Tweet {
  username :: T.Text,
  timestamp :: Int,
  content :: T.Text
} deriving (Show)
```

In this example[1], we have created a new `Tweet` datatype using record syntax. Our `Tweet` has a field for `username` and `content`, which are both of type `T.Text`[2], and a field for a `timestamp` which we are going to deal with as an Integer (for example, the current epoch time, for me writing this, is 1391490698).

The beginning of the file says that our module name is `Twitter` (so if we were going to import it, we would write `import Twitter`). We also export the `Tweet` data type so that after importing it (in some other file), we can use `Tweet` to create new Tweets.

Also, since we are using `Data.Text` we import it `as T`.

We can check out our fancy new datatype using `ghci src/Twitter.hs`. This starts up Haskell's interpreter for the compiler we are using (ghc). Once inside the prompt, we can run `Tweet "MyAwesomeUsername" 1234567 "42 is the answer!"` to see how a `Tweet` is constructed. It looks somthing like this:

```
*Twitter> Tweet "MyAwesomeUsername" 1234567 "42 is the answer!"
Tweet {username = "MyAwesomeUsername",
       timestamp = 1234567,
       content = "42 is the answer!"}
```

`:q` will get us out of the prompt.

5.1.2 Creating the Form

Now we can create our forms. Since we are using Digestive Functors with Snap and Heist, we will need a couple imports. Digestive Functors can be used in other contexts, including against

[1] OverloadedStrings is a commonly used language extension that makes it easier to write string literals and use them in our application.

[2] If you don't know what this is, it would be a good idea to look up the difference between `ByteString`, `Text` and `String` at some point.

JSON data.

```
import          Text.Digestive
import          Text.Digestive.Heist
import          Text.Digestive.Snap
```

Along with these imports, we need to tell cabal what packages to include when installing. Add these digestive-functor imports to `df-one.cabal`.

```
  Build-depends:
    bytestring                >= 0.9.1   && < 0.11,
    heist                     >= 0.13    && < 0.14,
    MonadCatchIO-transformers >= 0.2.1   && < 0.4,
    mtl                       >= 2       && < 3,
    snap                      >= 0.11    && < 0.13,
    snap-core                 >= 0.9     && < 0.11,
    snap-server               >= 0.9     && < 0.11,
    snap-loader-static        >= 0.9     && < 0.10,
    text                      >= 0.11    && < 0.12,
    time                      >= 1.1     && < 1.5,
    xmlhtml                   >= 0.1,
    digestive-functors        >=0.6.1    && <0.7,
    digestive-functors-snap   >= 0.6.0.0 && < 0.7.0.0,
    digestive-functors-heist  == 0.7.0.0
```

Now we are going to construct the actual form. We'll use a helper function `isNotEmpty` to check the inputs and make sure they aren't empty. `isNotEmpty` will take a `T.Text` and return a `Bool` to us.

```
isNotEmpty :: T.Text -> Bool
isNotEmpty = not . T.null
```

We will also define some error strings to display if the input isn't quite right.

```
userErrMsg :: T.Text
userErrMsg = "Username can not be empty"
tsErrMsg :: T.Text
tsErrMsg = "timestamp must be an Int"
contentErrMsg :: T.Text
contentErrMsg = "Tweet can not be empty"
```

We can then use these in our form:

```
tweetForm :: (Monad m) => Form T.Text m Tweet
tweetForm = Tweet
  <$> "username" .: check userErrMsg isNotEmpty (text Nothing)
  <*> "timestamp" .: stringRead tsErrMsg Nothing
  <*> "content"   .: check contentErrMsg isNotEmpty (text Nothing)
```

We are using a simple `check` function which takes an error string, a test function and a form to validate. This may seem a little confusing, until we examine that `text`[3] returns a `Formlet` for us.

In the case of `text`, we can choose to specify a default value (for use as the username or content). Currently we have `Nothing`, or no default value. The alternative is `Just "sometext"`, which is a default value of `sometext`.

`stringRead` is similar to `text`, but for parseable and serializable values, such as our `Int`. `stringRead` takes an error string. After giving it an error string, the combination of `stringRead "error string"` acts exactly the same as `text`, which means we get to specify a default value. Again either `Nothing` or `Just 5`[4].

The <$> and <*> operators come from Control.Applicative and we have to import it to use them:

```
import Control.Applicative
```

Our `Twitter.hs` now looks like this:

```
{-# LANGUAGE OverloadedStrings #-}
module Twitter
(Tweet
  ) where

import qualified Data.Text as T
import           Text.Digestive
import           Text.Digestive.Heist
import           Text.Digestive.Snap
import           Control.Applicative

data Tweet = Tweet {
  username :: T.Text,
  timestamp :: Int,
  content :: T.Text
} deriving (Show)

isNotEmpty :: T.Text -> Bool
isNotEmpty = not . T.null
```

[3] and a bunch of other functions we will examine later such as `bool`, `optionalText` and `utcTimeFormlet`

[4] More on Maybe here: http://learnyouahaskell.com/a-fistful-of-monads#getting-our-feet-wet-with-maybe

```
userErrMsg :: T.Text
userErrMsg = "Username can not be empty"
tsErrMsg :: T.Text
tsErrMsg = "timestamp must be an Int"
contentErrMsg :: T.Text
contentErrMsg = "Tweet can not be empty"

tweetForm :: (Monad m) => Form T.Text m Tweet
tweetForm = Tweet
  <$> "username" .: check userErrMsg isNotEmpty (text Nothing)
  <*> "timestamp" .: stringRead tsErrMsg Nothing
  <*> "content" .: check contentErrMsg isNotEmpty (text Nothing)
```

In ghci, we can play around with the form a bit and see what happens.

```
ghci src/Twitter.hs
```

gets us into the prompt and `getForm` will give us the resulting view:

```
*Twitter> getForm "MyTweetForm" tweetForm

View "thing" [] App
  App
    Map _
      Ref "username"
        Map _
          Pure (Text "")
    Ref "timestamp"
      Map _
        Pure (Text "")
  Ref "content"
    Map _
      Pure (Text "")
 [] [] Get
```

5.1.3 Building The Heist Templates

Before we can render out our form we should write a template. `Text.Digestive.Heist` gives us some splices (bits of Heist templates) that we can use to render out our form.

In a new template file at `snaplets/heist/templates/tweetform.tpl`

```
<apply template="base">
  <dfForm action="/tweet">
    <dfChildErrorList ref="" />

    <dfLabel ref="username">Username: </dfLabel>
    <dfInputText ref="username" />
    <br>

    <dfLabel ref="timestamp">Timestamp: </dfLabel>
    <dfInput ref="timestamp" type="number" min="0" step="1" pattern="\d+" />
    <br>

    <dfLabel ref="content">Content: </dfLabel>
    <dfInputTextArea ref="content" />
    <br>

    <dfInputSubmit value="Submit" />
  </dfForm>
</apply>
```

The tags that start with `df` are processed by Digestive Functors before displaying. We are using a couple different tags: `dfForm`, `dfChildErrorList`, `dfLabel`, `dfInputText`, `dfInput`, `dfInputTextArea` and `dfInputSubmit`. We will go into these a bit more at the end of this chapter, but for now the important part is `ref`, which Digestive Functors uses to identify form elements.

When rendered without errors (and a form name of “tweet”, more on that in a sec), this template will look like this:

```
<form action="/tweet" method="POST"
 enctype="application/x-www-form-urlencoded">

    <label for="tweet.username">Username: </label>
    <input type="text" id="tweet.username"
     name="tweet.username" value="">
    <br>

    <label for="tweet.timestamp">Timestamp: </label>
    <input type="number" min="0" step="1" pattern="\d+"
     id="tweet.timestamp" name="tweet.timestamp" value="">
    <br>

    <label for="tweet.content">Content: </label>
    <textarea id="tweet.content" name="tweet.content"></textarea>
    <br>
```

```
    <input value="Submit" type="submit">
</form>
```

You'll notice that the field ids and names are all namespaced by the form name (tweet). We also get a couple things for free, including encodingtype, method, some types and values.

These values will keep the values that are input if there are errors in other fields and the errors will be displayed at the top of the form.

5.1.4 The FormHandler Routing Function

We can now write our Snap Form Handler:

```
tweetFormHandler :: Handler App App ()
tweetFormHandler = do
  (view, result) <- runForm "tweet" tweetForm
  case result of
    Just x  -> writeText $ T.pack $ show x
    Nothing -> heistLocal (bindDigestiveSplices view) $ render "tweetform"
```

We now need to export `tweetFormHandler` so we can use it later.

```
module Twitter
(Tweet
,tweetFormHandler  ) where
```

Remember when we used `getForm` to test our form in ghci? Well it so happens that Digestive Functors Snap has a function that will automatically choose between `getForm` and `postForm` for us based on the type of request. It is called `runForm`. In addition, runForm takes the form name that we saw in the html before ("tweet"). This name can be any string we want and our forms will automatically be namespaced by it.

Essentially what's going on here is that if we can parse a `Tweet` datatype, `result` is where it will be stored and `Just x` will match in our case statement. If we can't parse a `Tweet`, `result` will be `Nothing` and the `view` will be rendered out with our `tweetform` template. `bindDigestiveSplices` is what allows us to use the `dfInput` and other `df` tags in our html.

We also need some more imports. I've imported only what we need from Snap.Core and Snap.Snaplet.Heist to make it more obvious where these functions are coming from. In the future, to import the whole modules, you can delete the parentheses and the text inside them.

```
import           Snap.Core (writeText)
import           Snap.Snaplet
import           Snap.Snaplet.Heist (heistLocal, render)
import           Application
```

5.1.5 Final Steps

Now that we have everything set up in `src/Twitter.hs` and our template written, let's go into `src/Site.hs` and create a route. First we'll add Twitter to our list of imports.

```
import           Control.Applicative
import           Data.ByteString (ByteString)
import           Data.Maybe
import qualified Data.Text as T
import           Snap.Core
import           Snap.Snaplet
import           Snap.Snaplet.Auth
import           Snap.Snaplet.Auth.Backends.JsonFile
import           Snap.Snaplet.Heist
import           Snap.Snaplet.Session.Backends.CookieSession
import           Snap.Util.FileServe
import           Heist
import qualified Heist.Interpreted as I
import           Twitter
```

Now we can add our `tweetFormHandler` to our routes. Since our form is already set up to POST to /tweet, we'll use that as our route:

```
------------------------------------------------------------------------
-- | The application's routes.
routes :: [(ByteString, Handler App App ())]
routes = [ ("/logins",    with auth handleLoginSubmit)
         , ("/logout",   with auth handleLogout)
         , ("/new_user", with auth handleNewUser)
         , ("/tweet", tweetFormHandler)
         , ("",          serveDirectory "static")
         ]
```

That's it. Run `cabal install` and then we can run `df-one` to run our app.

Visit `localhost:8000/tweet` in a browser to see our form and error handling in action!

Here is our finalized `src/Twitter.hs`

```
{-# LANGUAGE OverloadedStrings #-}
module Twitter
(Tweet
,tweetFormHandler  ) where

import qualified Data.Text as T
import           Text.Digestive
```

```haskell
import           Text.Digestive.Heist
import           Text.Digestive.Snap
import           Control.Applicative
import           Snap.Core (writeText)
import           Snap.Snaplet
import           Snap.Snaplet.Heist (heistLocal, render)
import           Application

data Tweet = Tweet {
  username :: T.Text,
  timestamp :: Int,
  content :: T.Text
} deriving (Show)

isNotEmpty :: T.Text -> Bool
isNotEmpty = not . T.null

userErrMsg :: T.Text
userErrMsg = "Username can not be empty"
tsErrMsg :: T.Text
tsErrMsg = "timestamp must be an Int"
contentErrMsg :: T.Text
contentErrMsg = "Tweet can not be empty"

tweetForm :: (Monad m) => Form T.Text m Tweet
tweetForm = Tweet
  <$> "username" .: check userErrMsg isNotEmpty (text Nothing)
  <*> "timestamp" .: stringRead tsErrMsg Nothing
  <*> "content" .: check contentErrMsg isNotEmpty (text Nothing)

tweetFormHandler :: Handler App App ()
tweetFormHandler = do
  (view, result) <- runForm "tweet" tweetForm
  case result of
    Just x  -> writeText $ T.pack $ show x
    Nothing -> heistLocal (bindDigestiveSplices view) $ render "tweetform"
```

5.2 More Samples

The code for these samples is in `code/df-samples`. The app in df-samples is a copy of the Twitter app, with various forms, routes and templates added.

5.3 Adding Splices

5.3.1 digestiveSplices

`digestiveSplices` are the splices that comes from `digestive-functors-heist`. It takes a View T.Text (for example the one in `tweetFormHandler` in `code/df-one/src/Twitter.hs`)

5.3.2 bindDigestiveSplices

`bindDigestiveSplices` is used very similar to heist's `bindSplices`. As seen in `code/df-one/src/Twitter.hs` from earlier in this chapter:

```
tweetFormHandler :: Handler App App ()
tweetFormHandler = do
  (view, result) <- runForm "tweet" tweetForm
  case result of
    Just x  -> writeText $ T.pack $ show x
    Nothing -> heistLocal (bindDigestiveSplices view) $ render "tweetform"
```

5.3.3 digestiveSplices and custom splices

`bindDigestiveSplices` returns a `heistState`, just like `bindSplices` does. This means that if we need to add custom splices *and* `digestiveSplices` to render a template, we can do it like this:

Provided we have these imports:

```
import qualified Heist.Interpreted as I
import Snap.Snaplets.Heist
```

we can change this from `code/df-one/src/Twitter.hs`

```
heistLocal (bindDigestiveSplices view) $ render "tweetform"
```

to this

```
heistLocal bSplices $ render "tweetform"
  where mysplices = [("thing", I.textSplice "what")
                    ,("otherthing", I.textSplice "other")]
        bSplices = (I.bindSplices mysplices . bindDigestiveSplices view)
```

The key part is replacing `bindDigestiveSplices view` with:

```
I.bindSplices mysplices . bindDigestiveSplices view
```

where `mysplices` is a list of simple `textSplices`.

5.4 Forms

All of these return `Formlets` that can be used when creating forms.

5.4.1 text

`text` defines a text form and takes an optional default value (`Just "some text"` or `Nothing`)

```
"username" .: text Nothing
```

and as seen in `code/df-one/src/Twitter.hs`:

```
"username" .: check userErrMsg isNotEmpty (text Nothing)
```

5.4.2 string

Same as `text`, but works on the `String` type.

5.4.3 stringRead

`stringRead` is used for parseable and serializable values. In our Tweet example, we used `Int`. We can check the documentation for Int to see that in the `Instances` list there are instances for `Read Int` and `Show Int`. `Read` and `Show` are the instances that make a value “parseable and serializable”.

It takes an error message and possibly a default value (either `Just 5` or `Nothing` in our case).

```
"timestamp" .: stringRead "My Error Message" Nothing
```

5.4.4 choice

A `choice` form can be used to restrict options, such as with a select tag. It takes a list of value, identifier pairs and an optional default value. From `code/df-samples/Twitter.hs`:

```
-- Data Type with two possible values.
data Thing = ThingOne | ThingTwo
           deriving (Show, Eq)

-- Form for data type using choice. "t1" and "t2"
--  will render in the dfInputSelect box
thingForm :: (Monad m) => Form T.Text m Thing
thingForm = "thething"  .: choice [(ThingOne, "t1"),
                                   (ThingTwo, "t2")] Nothing
```

Showing the flexibility of Digestive Functors, we have two template choices set up to render our choices. The first is at `localhost:8000/thing`, which renders a select box, the second is at `localhost:8000/thingradio`, which renders a set of radio buttons. Here are the relevant parts of both templates:

```
<dfLabel ref="thething">The Thing: </dfLabel>
<dfInputSelect ref="thething" />
```

and

```
<dfLabel ref="thething">The Thing: </dfLabel>
<dfInputRadio ref="thething" />
```

5.4.5 bool

`bool` is for Boolean values, like true and false and of course, takes an optional default value (Such as `Just True` or `Nothing`).

```
data Runner = Runner {
  isRunner :: Bool,
  name :: T.Text
  } deriving (Show)

runnerForm :: (Monad m) => Form T.Text m Runner
runnerForm = Runner
  <$> "isrunner" .: bool Nothing
  <*> "name" .: text Nothing
```

and the relevant templates in which we use a checkbox:

```
<dfLabel ref="name">Name: </dfLabel>
<dfInputText ref="name" />
<br>
```

```
<dfLabel ref="isrunner">Are you a Runner? </dfLabel>
<dfInputCheckbox ref="isrunner"/>
<br>
```

5.5 Testing Optional Values

5.5.1 optionalText

`optionalText` functions the same way as `text` with one difference; It accepts `Maybe` values (that is, it's for fields that don't need to be filled out).

```
data MRunner = MRunner {
  isMRunner :: Bool,
  mName :: Maybe T.Text
  } deriving (Show)

mRunnerForm :: (Monad m) => Form T.Text m MRunner
mRunnerForm = MRunner
  <$> "isrunner" .: bool Nothing
  <*> "name" .: optionalText Nothing
```

This form can use the same template as a `text` field. For example, here's the template from the `bool` example which had a `text` name:

```
<dfLabel ref="name">Name: </dfLabel>
<dfInputText ref="name" />
<br>

<dfLabel ref="isrunner">Are you a Runner? </dfLabel>
<dfInputCheckbox ref="isrunner"/>
<br>
```

Two potential successful form responses:

```
MRunner {isMRunner = True, mName = Just "Chris"}
MRunner {isMRunner = True, mName = Nothing}
```

5.6 Digestive Splices

The Digestive Splices are what are used to construct templates. If you are familiar with html input tags already, most of them are named the same way as the html tag they generate. Some examples are given here but a complete list can be found in the Heist docs on Hackage.

5.6.1 dfInput

Generates an `<input>` tag with the desired type.

```
<dfInput type="date" ref="date" />
```

5.6.2 dfInputText

This is how most of the other input tags are used. Define a `ref` attribute so that we can access it in our form validation. This ref will be translated to a namespaced equivalent (Something like `myform.name` in this example).

```
<dfInputText ref="username" />
```

becomes

```
<input type="text" id="tweet.username" name="tweet.username" value="">
```

5.6.2.1 A More Complicated Example

```
<dfInput ref="timestamp" type="number" min="0" step="1" pattern="\d+" />
```

becomes

```
<input type="number" min="0" step="1"
 pattern="\d+" id="tweet.timestamp"
 name="tweet.timestamp" value="">
```

5.6.3 Similar Splices

- dfInputTextArea
- dfInputPassword
- dfInputHidden
- dfInputSelect
- dfInputRadio
- dfInputCheckbox

5.6.4 dfInputSubmit

```
<dfInputSubmit value="Submit" />
```

becomes

```
<input value="Submit" type="submit">
```

5.6.5 dfLabel

```
<dfLabel ref="timestamp">Timestamp: </dfLabel>
```

becomes

```
<label for="tweet.timestamp">Timestamp: </label>
```

5.6.6 dfForm

`dfForm` is the equivalent of a `<form>` tag.

```
<dfForm action="/tweet">
```

becomes

```
<form action="/tweet" method="POST" enctype="application/x-www-form-urlencoded">
```

5.6.7 dfErrorList

`dfErrorList` renders the errors for a specific input on the form.

```
<dfErrorList ref="name">
```

5.6.8 dfChildErrorList

`dfChildErrorList` renders as a list of all of the errors in the form.

```
<dfChildErrorList ref="" />
```

could render as

```html
<ul>
<li>Username can not be empty</li>
<li>timestamp must be an Int</li>
<li>Tweet can not be empty</li>
</ul>
```

Chapter 6

Authentication Snaplet

The Auth Snaplet handles user signup, login and route restriction. This chapter uses code from `code/auth-app`.

6.1 Basics

6.1.1 Adding to App Definition

Simply add `_auth` with a type of `Snaplet (AuthManager App)`, we also need the Session Snaplet so we'll add that too. The heist snaplet is not strictly necessary, but we will use it to render splices from the Auth Snaplet.

```
data App = App
    { _heist :: Snaplet (Heist App)
    , _sess :: Snaplet SessionManager
    , _auth :: Snaplet (AuthManager App)
    }
```

6.1.2 Initialization

First we will initialize the Session Snaplet, then use the initialized Session Snaplet to initialize the Authentication Snaplet.

```
app :: SnapletInit App App
app = makeSnaplet "app" "An snaplet example application." Nothing $ do
    h <- nestSnaplet "" heist $ heistInit "templates"
    s <- nestSnaplet "sess" sess $
           initCookieSessionManager "site_key.txt" "sess" (Just 3600)
```

```
    a <- nestSnaplet "auth" auth $
           initJsonFileAuthManager defAuthSettings sess "users.json"
    addRoutes routes
    addAuthSplices h auth
    return $ App h s a
```

6.1.3 Adding Auth to Routes

To use auth-specific functions in routes we use `with`:

```
  ("/login",     with auth handleLoginSubmit)
, ("/logout",    with auth handleLogout)
, ("/new_user", with auth handleNewUser)
```

6.1.4 Handler Type

`with auth` takes a handler with a slightly different signature as an argument and returns a handler of the normal `Handler App App ()` type. This means that the `handle*` functions in the example above are of this type:

```
handleLogout :: Handler App (AuthManager App) ()
handleLogout = logout >> redirect "/"
```

We could rewrite the "`/logout`" handler to make this a bit more clear. We will add a new route "`/hlogout`", split out `with auth handleLogout` into it's own function (with type signature) and use the same `handleLogout` function to see the difference in handler types.

```
, ("/hlogout",    hLogout)
```

```
hLogout :: Handler App App ()
hLogout = with auth handleLogout

handleLogout :: Handler App (AuthManager App) ()
handleLogout = logout >> redirect "/"
```

If we look at our `App` declaration in `code/auth-app/src/Application.hs` we can see that the new type signature for our handlers includes the type of our Auth Snaplet:

```
, _auth :: Snaplet (AuthManager App)
```

6.2 Backends

Backends for the Authentication Snaplet are pluggable. Some of the current options include a flat JSON file and PostgreSQL.

6.2.1 JSON File

The default backend (given when you run `snap init`) is a flat JSON file. It is useful for examining how the system works, but should be replaced by the PostgreSQL backend or another database in production. One reason for this is that the users are stored in a flat file and this can cause issues.

6.2.1.1 Init with JSON

To initialize Auth with a JSON backend we will need to add the following import.

```
import            Snap.Snaplet.Auth.Backends.JsonFile
```

Then we can use `initJsonFileAuthManager` to create the Auth backend inside of our app init code:

```
s <- nestSnaplet "sess" sess $
       initCookieSessionManager "site_key.txt" "sess" (Just 3600)
a <- nestSnaplet "auth" auth $
       initJsonFileAuthManager defAuthSettings sess "users.json"
```

Remember that `nestSnaplet` takes a `ByteString` (the name of our snaplet), a Lensed Snaplet value (the ones we created when we ran `mkLenses` in `Application.hs`), and an init function.

`initJsonFileAuthManager` takes an `AuthSettings`, the Lensed Session Snaplet and the filepath we want to use to store the users.

6.2.2 PostgreSQL

PostgresSQL is one of the other backends available. It is more robust than the JSON file. The Postgres Chapter has more information on configuration.

6.2.2.1 snaplet-postgresql-simple

Add this to `Build-depends` in our `.cabal` file.

```
snaplet-postgresql-simple >= 0.4     && < 0.5
```

6.2.2.2 Adding to App Definition

We need to import the snaplet in `Application.hs`:

```haskell
import Snap.Snaplet.PostgresqlSimple
```

Then we can add `snaplet-postgresql-simple` to our app definition as such.

```haskell
data App = App
    { _heist :: Snaplet (Heist App)
    , _sess :: Snaplet SessionManager
    , _db :: Snaplet Postgres
    , _auth :: Snaplet (AuthManager App)
    }
```

6.2.2.3 Initializing the Backend

In `Site.hs` we will add a few imports.

```haskell
import           Snap.Snaplet.Auth.Backends.PostgresqlSimple
import           Snap.Snaplet.PostgresqlSimple
```

Then we can initialize the database with `pgsInit` and the backend as part of the Auth initialization.

```haskell
app :: SnapletInit App App
app = makeSnaplet "app" "An snaplet example application." Nothing $ do
    h <- nestSnaplet "" heist $ heistInit "templates"
    s <- nestSnaplet "sess" sess $
           initCookieSessionManager "site_key.txt" "sess" (Just 3600)
    d <- nestSnaplet "db" db pgsInit
    a <- nestSnaplet "auth" auth $
           initPostgresAuth sess d
    addRoutes routes
    addAuthSplices h auth
    return $ App h s d a
```

6.2.2.4 Instances

After setting up the initialization we can write an instance that is much like our regular instance:

```
instance HasPostgres (Handler b App) where
    getPostgresState = with db get
```

The new instance will be used inside of handlers with Auth type signatures.

```
instance HasPostgres (Handler App (AuthManager App)) where
    getPostgresState = withTop db get
```

These instances wil need a `{-# LANGUAGE FlexibleInstances #-}` declaration at the top of `Site.hs`.

6.3 Restricted Routes

To restrict a route to only logged in users, we can use `requireUser`. First we'll add a route at `/restricted` that uses the auth snaplet:

```
("/restricted", with auth restrictedHandler)
```

Then we'll write the handler with the auth snaplet in the type signature and a call to `requireUser`. `requireUser` takes a lensed auth snaplet value, such as `auth`, a handler to execute if there is no user logged in and a handler to execute if there is a user logged in.

```
restrictedHandler :: Handler App (AuthManager App) ()
restrictedHandler = requireUser auth noUserHandler userExistsHandler
```

We'll write each of these handlers as a simple ByteString response:

```
noUserHandler :: Handler App (AuthManager App) ()
noUserHandler = writeBS "No User"

userExistsHandler :: Handler App (AuthManager App) ()
userExistsHandler = writeBS "User Exists"
```

Note that `requireUser` just checks to see if there is a user_id in the session. This means there is no database cost.

Chapter 7

Snaplet PostgreSQL Simple

Snaplet PostgreSQL Simple offers a connection to the PostgreSQL database via the PostgreSQL Simple package.

7.1 Basics

Before installing `snaplet-postgresql-simple` you must have postgres installed on your system. Specifically, `pg_config` must be available on your path, which can come in `postgresql-devel`, `libpq-dev` or `postgresql` depending on your operating system of choice.

7.1.1 Add to .cabal

```
Build-depends:
  bytestring                >= 0.9.1   && < 0.11,
  heist                     >= 0.13    && < 0.14,
  MonadCatchIO-transformers >= 0.2.1   && < 0.4,
  mtl                       >= 2       && < 3,
  snap                      >= 0.13    && < 0.14,
  snap-core                 >= 0.9     && < 0.11,
  snap-server               >= 0.9     && < 0.11,
  snap-loader-static        >= 0.9     && < 0.10,
  text                      >= 0.11    && < 1.2,
  time                      >= 1.1     && < 1.5,
  xmlhtml                   >= 0.1,
  snaplet-postgresql-simple >= 0.4     && < 0.5
```

7.1.2 Adding to App

We need to import the snaplet:

```
import Snap.Snaplet.PostgresqlSimple
```

and then we can add `snaplet-postgresql-simple` to our app definition as such.

```
data App = App
    { _heist :: Snaplet (Heist App)
    , _sess :: Snaplet SessionManager
    , _db :: Snaplet Postgres
    , _auth :: Snaplet (AuthManager App)
    }
```

7.1.3 Initialization

We need to add Postgres to our app initialization:

```
app :: SnapletInit App App
app = makeSnaplet "app" "An snaplet example application." Nothing $ do
    h <- nestSnaplet "" heist $ heistInit "templates"
    s <- nestSnaplet "sess" sess $
           initCookieSessionManager "site_key.txt" "sess" (Just 3600)
    d <- nestSnaplet "db" db pgsInit
    a <- nestSnaplet "auth" auth $
           initJsonFileAuthManager defAuthSettings sess "users.json"
    addRoutes routes
    addAuthSplices h auth
    return $ App h s d a
```

The important parts to note are the inclusion of an additional `nestSnaplet` call for the database and the inclusion of the initialized Snaplet in the returned `App` value.

```
d <- nestSnaplet "db" db pgsInit
-- and
return $ App h s d a
```

7.1.4 Configuration

By default, Snaplets create their filesystem on first run of the application *if there are no files already there*. The Postgres files live in `snaplets/postgresql-simple/devel.cfg` and look like this by default:

```
host = "localhost"
port = 5432
user = "postgres"
pass = ""
db = "testdb"

# Nmuber of distinct connection pools to maintain.  The smallest acceptable
# value is 1.
numStripes = 1

# Number of seconds an unused resource is kept open.  The smallest acceptable
# value is 0.5 seconds.
idleTime = 5

# Maximum number of resources to keep open per stripe.  The smallest
# acceptable value is 1.
maxResourcesPerStripe = 20
```

Customize this file to adjust connection preferences and other options.

7.1.5 Instances

Instead of typing `with db` like we do when we need to use the Auth Snaplet, we can write an instance of `HasPostgres`, just like we did with Heist. From `src/Site.hs`:

```haskell
instance HasPostgres (Handler b App) where
    getPostgresState = with db get
```

To write this instance we need to import `get` from `Control.Monad.State.Class`:

```haskell
import Control.Monad.State.Class
```

7.1.6 psql

Since the defaults are a role of `postgres` and a database `testdb`, you may need to run these commands in `psql` to set up Postgres.

```
CREATE ROLE postgres LOGIN;
CREATE DATABASE testdb;
```

7.2 Querying

7.2.1 query

We can use the already set up tables for the auth snaplet (if we are using the Postgres backend) to see an example query.

```
getFromPostgres :: Handler App (AuthManager App) ()
getFromPostgres = do
    results <- query_ "select * from snap_auth_user"
    writeJSON (results :: [AuthUser])
```

Chapter 8

Deploying to Heroku

Quickly deploying to Heroku is fairly simple. We simply need to use a build-pack that has been prepared by the Haskell community.

8.1 Procfile

First, we need to put a `Procfile` in the root of our project; Save this as `Procfile`

```
web: cabal run -- -p $PORT
```

8.2 Build Pack

It is preferable to have a git repo initialized before creating the app with the build pack.

This command will create a new Heroku app with a Haskell buildpack. You can find more information on the buildpack here

```
heroku create --stack=cedar --buildpack https://github.com/ChristopherBiscardi/heroku-buildpack-ghc.git
```

8.3 Pushing

If you had a git repo initialized before running `heroku create`, you now have a `heroku` remote. Just push to Heroku and watch it build.

```
git push heroku
```

Chapter 9

Troubleshooting

9.1 Dependencies

Cause: after running `cabal install` or a similar command you see something similar to this:

```
Resolving dependencies...
cabal: Could not resolve dependencies:
trying: df-one-0.1
rejecting: digestive-functors-heist-0.8.4.1, 0.8.4.0, 0.8.3.1, 0.8.3.0,
0.8.1.0, 0.8.0.0 (conflict: df-one => digestive-functors-heist==0.7.0.0)
trying: digestive-functors-heist-0.7.0.0
rejecting: digestive-functors-0.7.0.0 (conflict: digestive-functors-heist =>
digestive-functors>=0.6.1 && <0.7)
rejecting: digestive-functors-0.6.2.0, 0.6.1.1, 0.6.1.0, 0.6.0.1, 0.6.0.0,
0.5.0.4, 0.5.0.3, 0.5.0.2, 0.5.0.1, 0.5.0.0, 0.4.1.2, 0.4.1.1, 0.4.1.0,
0.4.0.0, 0.3.2.1, 0.3.1.0, 0.3.0.2, 0.3.0.1, 0.3.0.0, 0.2.1.0, 0.2.0.1,
0.2.0.0, 0.1.0.2, 0.1.0.1, 0.1.0.0, 0.0.2.1, 0.0.2.0, 0.0.1 (conflict: df-one
=> digestive-functors>=0.7.0.0 && <0.8.0.0)
```

9.1.1 How to Fix

This means your dependencies are mismatched and cabal can't find a solution. The way to resolve this is to look into the `.cabal` files of the relevant projects and choose versions that are compatible. For example, in the above example, the issue is that we are trying to install `digestive-functors-heist`, but we can't find one that satisfies our .cabal file.

The relevant part of our cabal file (in this example) looks like this under `Build-depends`:

```
digestive-functors        >= 0.7.0.0 && <0.8.0.0,
digestive-functors-heist  == 0.7.0.0
```

and the error tells us

```
rejecting: digestive-functors-0.7.0.0 (conflict: digestive-functors-heist =>
digestive-functors>=0.6.1 && <0.7)
```

What the error is saying here is that cabal is `rejecting digestive-functors` version `0.7.0.0` as a viable dependency because it `conflicts` with `digestive-functors-heist`'s requirements. It then continures to tell us that the relevant requirements for `digestive-functors-heist` include `digestive-functors` versions greater than or equal to `0.6.1` and less than `0.7`.

At this point, we know that since our cabal file says to install a `digestive-functors` version that is `>= 0.7.0.0` and that `digestive-functors-heist` depends on `digestive-functors` being at a version that is less than `0.7.0.0` we have a conflict in the version of `digestive-functors`. The solution, luckily for us, is simple. Since `digestive-functors-heist` needs a specific version range, we just put that in our .cabal file, which now looks like this:

```
digestive-functors        >=0.6.1 && <0.7,
digestive-functors-heist  == 0.7.0.0
```

The version that `digestive-functors-heist` needs will now be installed and the dependency error will go away (unless somthing else in the .cabal file also has errors).