Snaplets: composable and reusable web components

9th Ghent Functional Programming Group meeting

Jurriën Stutterheim

October 4, 2011

Mainstream web development

- Dynamically typed languages (PHP, Ruby, Python)
  - Very low entry barrier
  - Many free/open source frameworks available
- Statically typed languages (Java, C#)
  - Mostly used by companies

What about Haskell?

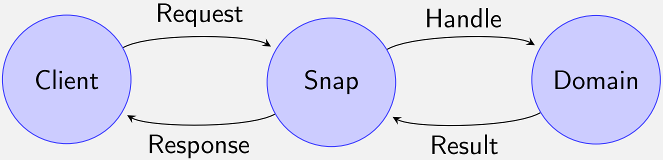
- Not used for web apps a lot (yet!)
  - Steep learning curve compared to PHP et al.
  - Few frameworks available (but a lot of very specialised packages)
  - Frameworks are not as feature-rich as their PHP (et al.) counterparts (yet!)
- Makes a great web language
  - Type-safe
  - Fast
  - Web-model fits Haskell nicely
    - Parse text, manipulate data, pretty-print
    - As opposed to being confined in IO by (possibly many) application windows

Haskell web frameworks

Major Haskell web frameworks:

- Snap Framework
- Yesod
- Happstack

Snap Framework



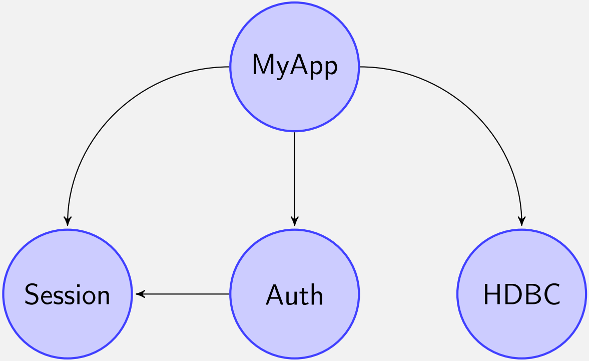
Today: snaplets

- The upcoming Snap 0.6 release introduces snaplets
- Improve reusability by creating composable components
- Today we will see two kinds of snaplet:
  - Guestbook application snaplet
  - Reusable HDBC snaplet

What are snaplets?

- An application is a snaplet, snaplets can be applications
- Can also be a reusable component
  - Sessions, database connections, athentication, etc.
- Self-contained
  - Can have handlers to handle requests
  - Can have local templates/CSS/JS, routes, state, etc.
- Can be nested in other snaplets (and hence applications)

Example snaplet configuration



Top-level snaplet initialization

```
guestbook :: SnapletInit App App
guestbook = makeSnaplet "guestbook"
  "An example guestbook application."
  Nothing $ do
    ...
```

- Initializer is the snaplet's entry point
- Configure snaplet name and paths
- Setup routes, sub-snaplets, etc.
- Start/finalise snaplet-wide sessions, connections, etc.

SnapletInit type

$$guestbook :: SnapletInit\ App\ App$$

$$newtype\ SnapletInit\ b\ v = \ldots$$

- b is the state type of the top-most snaplet (usually left variable in reusable snaplets).
- b can also be the current snaplet state type, if it is the top-most snaplet (i.e. your application)
- v is the state type of the current snaplet.

SnapletInit type (contd.)

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype SnapletInit b v =
  SnapletInit (Initializer b v (Snaplet v))
newtype Initializer b v a = ...
  deriving (MonadIO, ...)
```

Application state

State type for our example top-level application snaplet:

```
data App = App
  { _dbConn  :: Snaplet (HdbcSnaplet Connection)
  , _session :: Snaplet SessionManager
  , _auth    :: Snaplet (AuthManager App)
  , ... }
makeLens '' App
```

Lenses (an abstraction of accessor and mutator functions) are generated, which have the same name as the records, minus the underscore. They are used to get access to subsnaplet functions.

Using lenses

$$session :: Lens\ App\ (Snaplet\ SessionManager)$$

A lens can be seen as a pair of two functions:

$$\begin{array}{l}(App \to Snaplet\ SessionManager \\ , Snaplet\ SessionManager \to App \to App)\end{array}$$

With this we can use the *setInSession* function from the *SessionManager* snaplet using *with*:

$$with\ session\ \$\ setInSession\ \texttt{"login-failed"}\ \texttt{"1"}$$

Configuring our snaplet: routing

Things like routing, connections etc. are set up in the initializer.

A snaplet is responsible for routing requests to the appropriate handler

```
guestbook :: SnapletInit App App
guestbook = ... do
  addRoutes [("/",           ifTop indexHandler)
            ,("/delete/:id", deleteHandler)
            ,...]
  ...
```

(App)Handler

```
indexHandler :: AppHandler ()
type AppHandler a = Handler App App a
newtype Handler b v a = ...
  deriving (MonadIO, MonadSnap, ...)
```

- b and v serve the same purpose as in $SnapletInit$
- a is the handler return type (which is often ())

Example index handler

Guestbook messages are retrieved with *getMessages*, which uses the HDBC snaplet.

```
indexHandler :: AppHandler ()
indexHandler = do
  msgs ← getMessages
  blaze $ renderIndex msgs

getMessages :: HasHdbc m c ⇒ m [Message]

renderIndex :: [Message] → Html

blaze :: MonadSnap m ⇒ Html → m ()
```

MonadSnap can be used to access the request and response

Reading the messages

```
getMessages :: HasHdbc m c ⇒ m [Message]
```

The HDBC snaplet defines *HasHdbc*

```
class (IConnection c, MonadIO m) ⇒
  HasHdbc m c | m → c where
  getHdbc :: m c
```

HDBC snaplet state

The HDBC snaplet also has a state. We use it to store an HDBC connection:

```
data HdbcSnaplet = IConnection c ⇒ HdbcSnaplet {
  hdbcConn :: c}
```

which we can obtain from our application using the *HasHdbc* typeclass and the *dbConn* lens

```
instance HasHdbc AppHandler Connection where
  -- getHdbc :: AppHandler Connection
  getHdbc = with dbConn $ gets hdbcConn
```

Note: this is exactly the type of our handlers!

Initializing the HDBC snaplet

Reusable snaplet initialization is almost the same as application snaplet initialization

```haskell
hdbcInit :: IConnection c ⇒ c
         → SnapletInit b (HdbcSnaplet c)
hdbcInit conn = makeSnaplet "hdbc"
  "HDBC abstraction" Nothing $ do
    onUnload $ HDBC.disconnect conn
    return $ HdbcSnaplet conn
```

Wrap HDBC functions

We can now wrap HDBC functions to eliminate the need for passing the connection explicitly

```
withHdbc :: HasHdbc m c ⇒ (c → IO a) → m a
withHdbc f = do
  conn ← getHdbc
  liftIO $ f conn
getTables :: HasHdbc m c ⇒ m [String]
getTables = withHdbc HDBC.getTables
```

Original *getTables* type:

```
getTables :: IConnection c ⇒ c → IO [String]
```

Snaplet convenience function

The HDBC snaplet offers some convenience functions

```
query :: HasHdbc m c ⇒ String → [SqlValue]
        → m Integer
query sql bind = withTransaction $ λconn → do
  stmt ← HDBC.prepare conn sql
  liftIO $ HDBC.execute stmt bind
```

Initializing the HDBC snaplet

We connect to SQLite and pass the connection to the HDBC snaplet

```
guestbook :: SnapletInit App App
guestbook = ... do
  ...
  conn ← liftIO $ connectSqlite3
                    "resources/guestbook.db"
  hdbc ← nestSnaplet "hdbc" dbConn $
                         hdbcInit conn
  ...
  return $ App hdbc ...
```

Inserting DB rows

We can now use the HDBC snaplet in our application

```
indexHandler :: AppHandler ()
indexHandler = do
  . . .
  _ ← addMessage someMessage
  . . .
addMessage :: HasHdbc m c ⇒ Message → m Integer
addMessage (Message _ title body author) = query
  ("INSERT INTO messages (title, body, author)"
    ++ " VALUES (?, ?, ?)")
  [toSql title, toSql body, toSql author]
```

Serving the application

```
guestbook :: SnapletInit App App

main :: IO ()
main = serveSnaplet defaultConfig guestbook
```

Concluding

- Snaplets offer a powerful way to think about and build web applications and reusable components
- Currently there are not many 3rd party snaplets available
- Now you can write your own, so start hacking! ;)
- Code used in these slides is available on GitHub
  - https://github.com/norm2782/snap-guestbook
  - https://github.com/norm2782/snaplet-hdbc
- Check out the snap 0.6 branch (currently unstable!)
  - https://github.com/snapframework/snap/tree/0.6