

Haskell und das Web mit Hapstack

Haskell-Stammtisch

Dimitri Scheftelowitsch

01. Juni 2011

Motivation

... oder: **Warum überhaupt?**

Primäres Ziel: mehr Haskell-Praxis

Motivation

... oder: **Warum überhaupt?**

Primäres Ziel: mehr Haskell-Praxis

Sekundäres Ziel: ein halbwegs sinnvolles Programm schreiben

Motivation

... oder: **Warum überhaupt?**

Primäres Ziel: mehr Haskell-Praxis

Sekundäres Ziel: ein halbwegs sinnvolles Programm schreiben

Überlegungen zum UI:

- Desktop-GUI nicht sinnvoll nutzbar

Motivation

... oder: **Warum überhaupt?**

Primäres Ziel: mehr Haskell-Praxis

Sekundäres Ziel: ein halbwegs sinnvolles Programm schreiben

Überlegungen zum UI:

- Desktop-GUI nicht sinnvoll nutzbar
- Konsole **uninteressant**

Motivation

... oder: **Warum überhaupt?**

Primäres Ziel: mehr Haskell-Praxis

Sekundäres Ziel: ein halbwegs sinnvolles Programm schreiben

Überlegungen zum UI:

- Desktop-GUI nicht sinnvoll nutzbar
- Konsole **uninteressant**
- Web bleibt **einzig Alternative**

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod
- Happstack

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod
- Happstack
- Snap

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod
- Happstack
- Snap
- ...

Warum ausgerechnet Happstack?

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod
- Happstack
- Snap
- ...

Warum ausgerechnet Happstack?

- Viel Literatur

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod
- Happstack
- Snap
- ...

Warum ausgerechnet Happstack?

- Viel Literatur
- Sehr **stabil**

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod
- Happstack
- Snap
- ...

Warum ausgerechnet Happstack?

- Viel Literatur
- Sehr **stabil**
- Viele **Features** (Routing, Server-Infrastruktur etc.)

Werkzeuge I: Framework

Haskell bietet mehrere Web-Frameworks:

- Haskell on a Horse
- Yesod
- Happstack
- Snap
- ...

Warum ausgerechnet Happstack?

- Viel Literatur
- Sehr **stabil**
- Viele **Features** (Routing, Server-Infrastruktur etc.)
- Eines der Happstack-Beispiele war Grundlage

Werkzeuge II: Ausgabe

Happstack alleine reicht leider nicht. . .

Problem: **HTML**

Werkzeuge II: Ausgabe

Happstack alleine reicht leider nicht. . .

Problem: **HTML**

Es gibt prinzipiell mehrere Lösungen:

- HSP (HTML **inline**, wie im Fall von PHP / JSP / . . .)

Werkzeuge II: Ausgabe

Happstack alleine reicht leider nicht. . .

Problem: **HTML**

Es gibt prinzipiell mehrere Lösungen:

- HSP (HTML **inline**, wie im Fall von PHP / JSP / . . .)
- tagsoup (Template-Engine)

Werkzeuge II: Ausgabe

Happstack alleine reicht leider nicht. . .

Problem: **HTML**

Es gibt prinzipiell mehrere Lösungen:

- HSP (HTML **inline**, wie im Fall von PHP / JSP / . . .)
- tagsoup (Template-Engine)
- Text.XHtml (HTML in Kombinatoren verpackt)

Werkzeuge II: Ausgabe

Happstack alleine reicht leider nicht. . .

Problem: **HTML**

Es gibt prinzipiell mehrere Lösungen:

- HSP (HTML **inline**, wie im Fall von PHP / JSP / . . .)
- tagsoup (Template-Engine)
- Text.XHtml (HTML in Kombinatoren verpackt)

In meinem Fall: Letzte Variante hat sich angeboten (weil besonders unkompliziert)

Werkzeuge III: Datenbank

Auch hier gibt es eine Fülle von Varianten:

- acid-state, happstack-state und Co: Auto-Serialisierung

Werkzeuge III: Datenbank

Auch hier gibt es eine Fülle von Varianten:

- acid-state, happstack-state und Co: Auto-Serialisierung
- Klassische Datenbanken, etwa JDBC

Werkzeuge III: Datenbank

Auch hier gibt es eine Fülle von Varianten:

- acid-state, happstack-state und Co: Auto-Serialisierung
- Klassische Datenbanken, etwa HDBC

Hier: sqlite3 über HDBC (theoretisch erweiterbar auf alles, was HDBC kann)

Die wichtigste Frage

Was soll's denn eigentlich werden?

Die wichtigste Frage

Was soll's denn eigentlich werden?

Ein **Imageboard!**

Die wichtigste Frage

Was soll's denn eigentlich werden?

Ein **Imageboard!**

Warum eigentlich?

Die wichtigste Frage

Was soll's denn eigentlich werden?

Ein **Imageboard!**

Warum eigentlich?

- sehr **Web 2.0**

Die wichtigste Frage

Was soll's denn eigentlich werden?

Ein **Imageboard!**

Warum eigentlich?

- sehr **Web 2.0**
- **einfaches** Datenmodell

Die wichtigste Frage

Was soll's denn eigentlich werden?

Ein **Imageboard!**

Warum eigentlich?

- sehr **Web 2.0**
- **einfaches** Datenmodell
- **einfache** HTML-Struktur

Die wichtigste Frage

Was soll's denn eigentlich werden?

Ein **Imageboard!**

Warum eigentlich?

- sehr **Web 2.0**
- **einfaches** Datenmodell
- **einfache** HTML-Struktur
- hinreichend **komplexe** Logik

Datenmodell

Das Datenmodell für ein Board ist eher einfach:

```
1 data Post = Post { postId :: Integer
2     , postTitle  :: String
3     , postTimestamp :: UTCTime
4     , postBoard  :: String
5     , postAttachment :: Maybe String
6     , postContents :: String
7     , postUser   :: String }
8
9 data User = User { userName :: String
10    , userPass  :: String }
```

Datenmodell II

Außerdem brauchen wir eine Art Config-Struktur:

```
1 data Config = Config { cport :: Int
2     , cmasterKey :: String
3     , cboardName :: String
4     , cdefaultUser :: String
5     , cAuthPolicy :: AuthPolicy }
6     deriving Show
7
8 data AuthPolicy = AnonymousOnly | DenyRegistration |
9     AllowFromAll
10    deriving (Show, Read, Eq)
```

Configs schreiben

Wo wir schon Configs erwähnt haben, wollen wir sie auch extern speichern:

```
[board]
masterkey = secret!
port = 3000
boardname = Winged doom
authpolicy = AllowFromAll
```

Gelesen wird dann mit `Data.ConfigFile`:

```
1 import Data.ConfigFile
2 ...
3 type SectionSpec = String
4 type OptionSpec = String
5 get :: MonadError CPError m => ConfigParser -> SectionSpec
   -> OptionSpec -> m a
6
7 readConfig file = do
8   config <- runErrorT $ do
9     conf <- join $ liftIO $ readfile emptyCP file
10    let param = get conf "board"
11        liftM5 Config (liftM read (param "port"))
12                    (param "masterkey")
13                    (param "boardname")
14                    (param "anonymous")
15                    (liftM read (param "authpolicy"))
16    case config of
17      Left e -> error $ show e
18      Right conf -> return conf
```

Happstack

Happstack lebt in `ServerMonad`. Dahin kommt man etwa mit:

```
1 simpleHTTP :: (ToMessage a) => Conf -> ServerPartT IO a ->
  IO ()
```

Happstack

Happstack lebt in `ServerMonad`. Dahin kommt man etwa mit:

```
1 simpleHTTP :: (ToMessage a) => Conf -> ServerPartT IO a ->
   IO ()
```

Zum Beispiel starten wir unsere Board mit

```
1 simpleHTTP nullConf { port = portNumber } (boardApp db conf)
```

Routing

Ein richtiges Web-Framework soll die Anfrage sinnvoll verarbeiten können:

Beispiel

- `/login` → Anmeldung

Routing

Ein richtiges Web-Framework soll die Anfrage sinnvoll verarbeiten können:

Beispiel

- `/login` → Anmeldung
- `/games/0` → Poker

Routing

Ein richtiges Web-Framework soll die Anfrage sinnvoll verarbeiten können:

Beispiel

- `/login` → Anmeldung
- `/games/0` → Poker
- `/games/1` → Globaler thermonuklearer Krieg

Routing

Ein richtiges Web-Framework soll die Anfrage sinnvoll verarbeiten können:

Beispiel

- `/login` → Anmeldung
- `/games/0` → Poker
- `/games/1` → Globaler thermonuklearer Krieg

Happstack übernimmt für uns einen Großteil der Arbeit:

```
1 route = msum
2       [ dir "conf" showSetup
3         , dir "games" $ path game ]
4
5 game "0" = showPoker
6 game "1" = showWarGame
```

Routing II

In unserem Fall sieht das Routing wie folgt aus:

```
1 boardApp :: (IConnection a) => a -> Config -> ServerPart
  Response
2 boardApp db conf = msum
3                   [ methodOnly GET  >> path (handleShow db
4                     conf)
5                   , methodOnly POST >> path (handlePost db
6                     conf) ]
7 handleShow :: (IConnection d) => d -> Config -> String ->
  ServerPart Response
8 handleShow _ _ "favicon.ico" = serveFile (
  guessContentTypeM mimeTypes) "favicon.ico"
9 handleShow _ _ "_res" = serveDirectory
  DisableBrowsing [] "_res"
10 handleShow db conf board = ... -- weitere Routing-Ebenen
```

Anfrageverarbeitung

Weiteres wichtiges Element einer Web-Anwendung ist die
Umwandlung Anfrage \rightsquigarrow Daten:

Anfrageverarbeitung

Weiteres wichtiges Element einer Web-Anwendung ist die
Umwandlung Anfrage \rightsquigarrow Daten:

Dafür existiert die Typklasse `FromData`:

```
1 class FromData a where
2   fromData :: RqData a
3
4   withData :: (HasRqData m, MonadIO m, FromData a, MonadPlus m
   , ServerMonad m) => (a -> m r) -> m r
```

Anfrageverarbeitung

Weiteres wichtiges Element einer Web-Anwendung ist die Umwandlung Anfrage \rightsquigarrow Daten:

Dafür existiert die Typklasse `FromData`:

```
1 class FromData a where
2   fromData :: RqData a
3
4   withData :: (HasRqData m, MonadIO m, FromData a, MonadPlus m
   , ServerMonad m) => (a -> m r) -> m r
```

Die Verarbeitung funktioniert wie folgt:

```
1 instance FromData User where
2   fromData = do
3     uname <- look "uname"
4     upass <- look "upass"
5     return nullUser { userName = uname
6                       , userPass = upass }
7   ...
8 handler = withData $ (\user -> do ...)
```

Serverantwort

Jetzt wollen wir eine Antwort erzeugen.

Serverantwort

Jetzt wollen wir eine Antwort erzeugen.
Dafür hat Happstack folgende Hilfsfunktionen:

```
1 resp :: FilterMonad Response m
2   => Int -- response code
3     -> b -- value to return
4     -> m b
5
6 ok = resp 200
7 unauthorised = resp 401
8 forbidden = resp 403
9 notFound = resp 404
```

Kleine Nettigkeit: Es ist `mzero = notFound`

Mehr zu Daten

Häufiges Idiom:

```
1 return nullUser { userName = uname, userPass = upass }
```

Mehr zu Daten

Häufiges Idiom:

```
1 return nullUser { userName = uname, userPass = upass }
```

Was ist überhaupt nullUser?

Mehr zu Daten

Häufiges Idiom:

```
1 return nullUser { userName = uname, userPass = upass }
```

Was ist überhaupt nullUser?

```
1 nullUser = User { userName = undefined, userPass = undefined  
  }
```

Mehr zu Daten

Häufiges Idiom:

```
1 return nullUser { userName = uname, userPass = upass }
```

Was ist überhaupt nullUser?

```
1 nullUser = User { userName = undefined, userPass = undefined  
  }
```

Benutzung von \perp liefert bei einer falschen Eingabe einen Fehler

Mehr zu Daten

Häufiges Idiom:

```
1 return nullUser { userName = uname, userPass = upass }
```

Was ist überhaupt nullUser?

```
1 nullUser = User { userName = undefined, userPass = undefined  
  }
```

Benutzung von \perp liefert bei einer falschen Eingabe einen Fehler
Dieser wird dann von der Server-Monade bearbeitet

HTML

Wie versprochen: HTML wird mit Text.XHtml erzeugt, etwa so:

```
1 [ thediv ! [theclass "message" ]
2   << [ anchor ! [theclass "postId"] $ toHtml $ show $
3     postId post
4     , toHtml " "
5     , thespan ! [theclass "username"] $ toHtml $ postUser
6       post
7       , toHtml " "
8       , thespan ! [theclass "timestamp"] $ toHtml
9         formattedTime
10      , toHtml " "
11      , thespan ! [theclass "messageheader"] $ toHtml $
12        postTitle post
13        , toHtml " "
14        , deleteCheckbox ]
```

Datenbank

Der Zugriff zur Datenbank erfolgt über JDBC. Am Beispiel der Benutzerbehandlung sieht das so aus:

```
1 addUserToDb :: (IConnection d, MonadIO m, MonadPlus m)
2             => d -> User -> m Integer
3 addUserToDb db u = do
4   liftIO $ quickQuery db "INSERT INTO users (name, hash)
5     VALUES (?, ?)" [toSql name, toSql $ show $ md5 $ Str
6     pass]
7   [[uid]] <- liftIO $ quickQuery db "select
8     last_insert_rowid()" []
9   return $ fromSql uid
10  where (name, pass) = (userName u, userPass u)
```

Datenbank

Der Zugriff zur Datenbank erfolgt über JDBC. Am Beispiel der Benutzerbehandlung sieht das so aus:

```
1 addUserToDb :: (IConnection d, MonadIO m, MonadPlus m)
2             => d -> User -> m Integer
3 addUserToDb db u = do
4   liftIO $ quickQuery db "INSERT INTO users (name, hash)
5     VALUES (?, ?)" [toSql name, toSql $ show $ md5 $ Str
6     pass]
7   [[uid]] <- liftIO $ quickQuery db "select
8     last_insert_rowid()" []
9   return $ fromSql uid
10  where (name, pass) = (userName u, userPass u)
```

Man beachte das toSql: Dies ist keine besonders typsichere Lösung...

Bessere Datenbank: happstack-state

Das gleiche hätte man auch besser lösen können:
happstack-state ist eine verbesserte State-Monade, die das Speichern übernimmt

Bessere Datenbank: happstack-state

Das gleiche hätte man auch besser lösen können:

`happstack-state` ist eine verbesserte State-Monade, die das Speichern übernimmt

Dann würde die Funktion `addUserToDb` wie folgt aussehen:

```
1 storeUser u = do
2   appState <- get
3   let newUsers = u:(users appState)
4   put $ appState {users = newUsers}
5
6 addUserToDb u = update (StoreUser u)
```

Bessere Datenbank: happstack-state

Das gleiche hätte man auch besser lösen können:

`happstack-state` ist eine verbesserte State-Monade, die das Speichern übernimmt

Dann würde die Funktion `addUserToDb` wie folgt aussehen:

```
1 storeUser u = do
2   appState <- get
3   let newUsers = u:(users appState)
4   put $ appState {users = newUsers}
5
6 addUserToDb u = update (StoreUser u)
```

`StoreUser` wäre ein von `TH` erzeugter Typ:

```
1 $(mkMethods ''AppState ['storeUser, ...])
```

Authentifizierung

Ab hier lohnt sich Haskell wirklich:

```
1 authenticate :: (IConnection d, MonadIO m, MonadPlus m)
2               => d -> Config -> User -> m AuthData
3 authenticate db conf u = do
4   let policy = cAuthPolicy conf
5       auth <- authenticateUserFromDb db u
6       case userName u == cdefaultUser conf of
7         True -> return Anonymous
8         _ -> case (auth, policy) of
9             (Unregistered , AllowFromAll ) -> liftM AuthUser $
10                addUserToDb db u
11             (Unregistered , _             ) -> return AuthError
12             (AuthUser _    , AnonymousOnly) -> return AuthError
13             (Anonymous    , AnonymousOnly) -> return Anonymous
14             (_             , AllowFromAll ) -> return auth
15             (AuthError    , _             ) -> return AuthError
```

Codefragment

```

1 showBoard :: IConnection d => d -> Config -> String ->
      Integer -> ServerPart Response
2 showBoard db conf board page = do
3   view <- boardView db conf board page
4   boards <- getBoardsFromDb db
5   let boardFound = filter (\z@(a,b) -> a == board) boards
6       if [] == boardFound
7         then mzero -- 404
8         else do header <- genHeader db conf $ Just board
9                 ok $
10                toResponse $
11                concatHtml [ header
12                             , h3 $ toHtml $ "/" ++ board ++ "/"
13                               -- " ++ snd (head $ filter (\z@(
14                               a,b) -> a == board) boards)
15                             , replyForm [] board Nothing
16                               nullPost
17                             , view
18                             , genFooter ]

```

Mehr Happstack!

Das war noch lange nicht alles...

Happstack hat noch einiges zu bieten:

Das war noch lange nicht alles...

Happstack hat noch einiges zu bieten:

- Authentifizierung und Sessions „von Haus aus“

Das war noch lange nicht alles...

Hapstack hat noch einiges zu bieten:

- Authentifizierung und Sessions „von Haus aus“
- Automatische Serialisierung des Zustands

Das war noch lange nicht alles...

Happstack hat noch einiges zu bieten:

- Authentifizierung und Sessions „von Haus aus“
- Automatische Serialisierung des Zustands
- Verwaltung von Plugins

Das war noch lange nicht alles...

Happstack hat noch einiges zu bieten:

- Authentifizierung und Sessions „von Haus aus“
- Automatische Serialisierung des Zustands
- Verwaltung von Plugins
- Typsicheres Routing (mit `web-routes`)

Das war noch lange nicht alles...

Happstack hat noch einiges zu bieten:

- Authentifizierung und Sessions „von Haus aus“
- Automatische Serialisierung des Zustands
- Verwaltung von Plugins
- Typsicheres Routing (mit `web-routes`)
- ...

Ergebnisse

Die Erfahrung hat einige Erkenntnisse geliefert:

- SQLite ist das größte Bottleneck des Programms

Ergebnisse

Die Erfahrung hat einige Erkenntnisse geliefert:

- SQLite ist das größte Bottleneck des Programms
- highlighting-kate ist 85 MB groß

Ergebnisse

Die Erfahrung hat einige Erkenntnisse geliefert:

- SQLite ist das größte Bottleneck des Programms
- `highlighting-kate` ist 85 MB groß
- Profiling lohnt sich wirklich
- `hlint` ist hilfreich, hat aber Bugs

Ergebnisse

Die Erfahrung hat einige Erkenntnisse geliefert:

- SQLite ist das größte Bottleneck des Programms
- `highlighting-kate` ist 85 MB groß
- Profiling lohnt sich wirklich
- `hlint` ist hilfreich, hat aber Bugs
- Das schwierigste Fragment beim Debuggen

Ergebnisse

Die Erfahrung hat einige Erkenntnisse geliefert:

- SQLite ist das größte Bottleneck des Programms
- `highlighting-kate` ist 85 MB groß
- Profiling lohnt sich wirklich
- `hlint` ist hilfreich, hat aber Bugs
- Das schwierigste Fragment beim Debuggen waren die CSS

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Insgesamt ist das Programm (ohne Kommentare) etwa 430 Zeilen lang. Es kann unter Anderem:

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Insgesamt ist das Programm (ohne Kommentare) etwa 430 Zeilen lang. Es kann unter Anderem:

- Highlighting von speziellen **Formatierungen**

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Insgesamt ist das Programm (ohne Kommentare) etwa 430 Zeilen lang. Es kann unter Anderem:

- Highlighting von speziellen **Formatierungen**
- **Thumbnails** verwalten

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Insgesamt ist das Programm (ohne Kommentare) etwa 430 Zeilen lang. Es kann unter Anderem:

- Highlighting von speziellen **Formatierungen**
- **Thumbnails** verwalten
- triviales **Authentifizierungs-Modell**

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Insgesamt ist das Programm (ohne Kommentare) etwa 430 Zeilen lang. Es kann unter Anderem:

- Highlighting von speziellen **Formatierungen**
- **Thumbnails** verwalten
- triviales **Authentifizierungs-Modell**

Es ist aber bei Weitem **nicht fertig**, es fehlt etwa

- Bessere Datenbank (typischer, schneller, einfacher: die SQL-Wrapper sind etwa **1/3** des Programms)

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Insgesamt ist das Programm (ohne Kommentare) etwa 430 Zeilen lang. Es kann unter Anderem:

- Highlighting von speziellen **Formatierungen**
- **Thumbnails** verwalten
- triviales **Authentifizierungs-Modell**

Es ist aber bei Weitem **nicht fertig**, es fehlt etwa

- Bessere Datenbank (typischer, schneller, einfacher: die SQL-Wrapper sind etwa **1/3** des Programms)
- Admin-Interface

Fazit

Wichtigste Feststellung: Happstack ist dazu geeignet, komplexe Web-Anwendungen zu konstruieren

Insgesamt ist das Programm (ohne Kommentare) etwa 430 Zeilen lang. Es kann unter Anderem:

- Highlighting von speziellen **Formatierungen**
- **Thumbnails** verwalten
- triviales **Authentifizierungs-Modell**

Es ist aber bei Weitem **nicht fertig**, es fehlt etwa

- Bessere Datenbank (typischer, schneller, einfacher: die SQL-Wrapper sind etwa **1/3** des Programms)
- Admin-Interface
- Spielereien mit AJAX

Quellen

- Der Happstack-Crashkurs:
`http://happstack.com/docs/crashcourse/index.html`
- Ein Pasteboard mit Happstack:
`http://gitit.net/paste.lhs`
- Die Happstack-Doku: `http://happstack.com/docs`