

Going FUNctional full stack using Haskell, Elm, Firebase and RethinkDB

Vilnius, October 2017



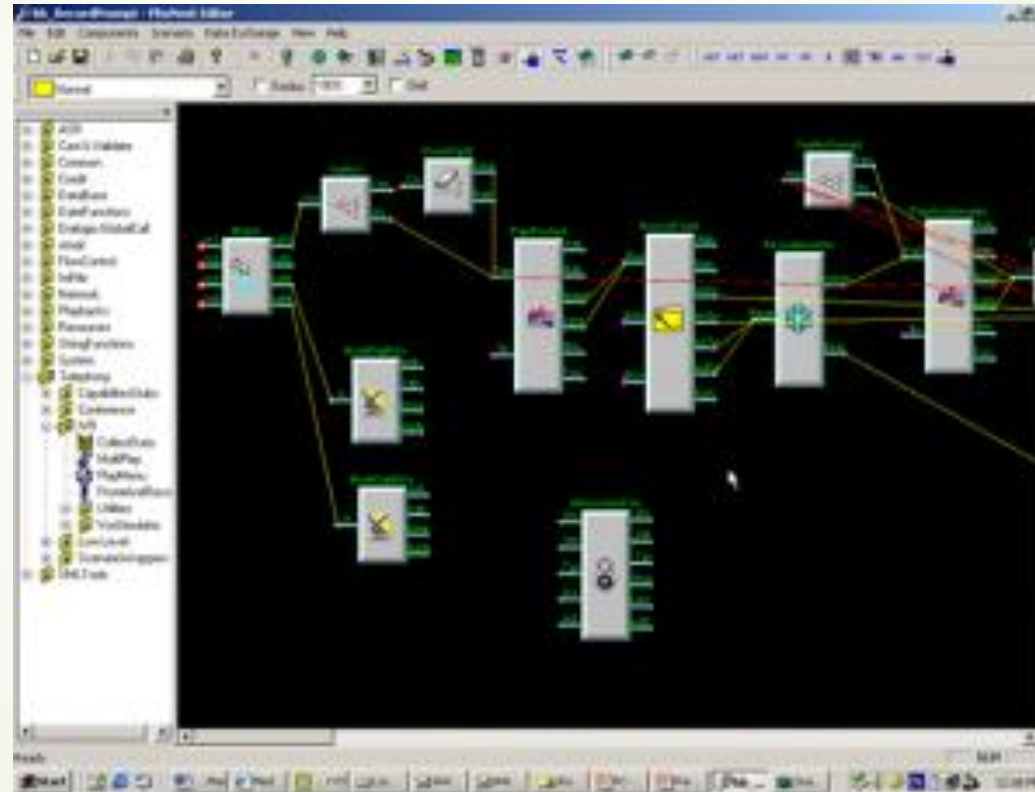
About myself

- Software developer (programmer) for unbelievably many years
- C, C++, C#, Java
- Mainly server side , but also some ASP, ASP.NET, JS, Adobe Flex/Air
- Databases (mostly relational, but also some noSQL)
- Mobile apps

- <https://github.com/unomemento>

Products of long life span

- <http://www.calltech-sw.com/platform/?lang=en>
- Application generator (using Microsoft COM technology) (since 1998):





FP is coming, FP is here

- We hear about FP from everywhere and people are saying it is good
- FP is coming to our traditional languages , even if the features are not called by their real name (lambda functions, C# Linq, Java 8 (Optional, CompletableFuture, Streams), JS (promises), new C++ standards
- Promote best practices that we knew already, which became even more important for concurrency
 - Immutability
 - Statelessness
- Time to learn some FP!



Learning FP Languages

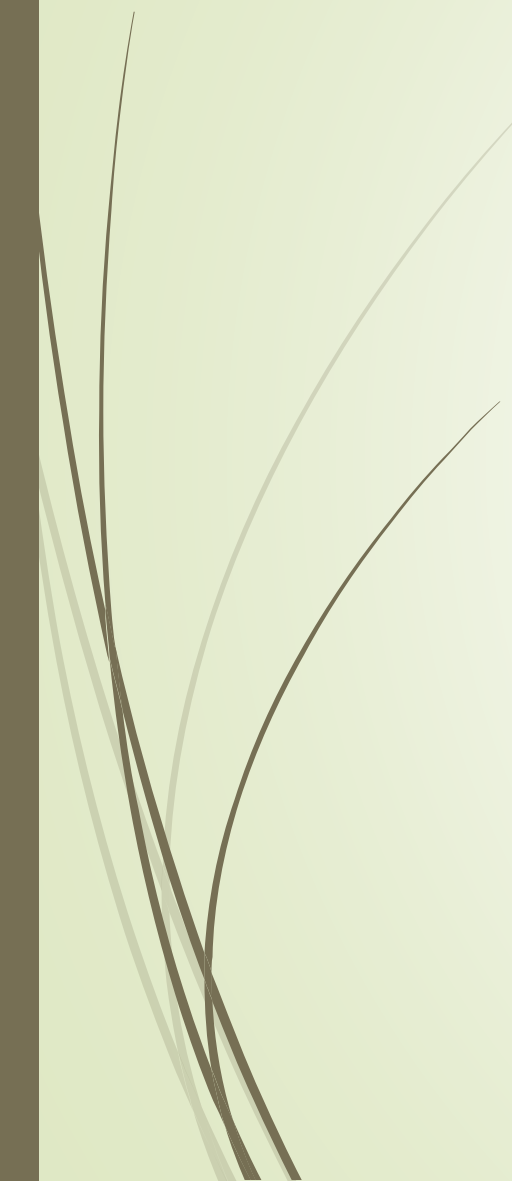
- Erlang (Actor model fault tolerance: failure will happen and it is OK, , dynamically typed)
- JVM
 - Clojure [Rich Hickey] (LISP dialect, everything is function, extremely elegant , dynamically typed)
 - Scala: allows both OO and FP [Martin Odersky]- became pretty popular
 - Very good Coursera courses:
 - Functional and reactive programming
 - Parallel data structures and Spark
 - I liked it a lot, so much nicer than Java!
 - There are very powerful functional libraries (ScalaZ, Cats, etc). Caveat: To use them one needs to understand more advanced concepts such as Monad.

Learning Haskell

- Main initial motivation was to understand some FP concepts such as Monad
- As most people , tried to read <http://learnyouahaskell.com/> . Did not work for me so well
- Then came across YouTube video: LambdaConf 2015 - How to Learn Haskell in Less Than 5 Years by Chris Allen and bought <http://haskellbook.com/>. It made a trick:
 - Slowly builds up knowledge: ADT, Type classes, Monoids, Factors, Applicative, Monad, Monad Transformers, Foldable, Traversable, etc
 - Very important: exercises after every chapters.
 - One finished, you will be empowered to use Haskell for real.
- Now many great online training (Udemy, Stepik (Russian), Glasgow university!)
- Blogs: <http://www.haskellforall.com/> by Gabriel Gonzales and many others
- Haskell weekly (<https://haskellweekly.news/>)



From Gabriel Gonzalez: Advice for Haskell beginners

- Recommend reading the Haskell Programming from first principles book
 - Learn Haskell for the right reasons (Manage your expectation)
 - Avoid big-design-up-front (refactoring is OK)
 - Avoid typeclass abuse (you can do a lot with built in features)
 - Build something useful
- 



Be aware!

- After learning Haskell , your feeling to other languages might be impacted
- 

What you should demand from your server side language

- Powerful type system to enable type driven development
 - Statically typed (types are known at compile time)
 - Strong type inference (you still want to have signatures for all top level functions)
 - Expressive data structures: ADT and especially summary (union) types (to express alternatives)
 - `data Pair a b = Pair a b` -- product type (number inhabitants $a*b$)
 - `data Either a b = Left a | Right b` -- sum (union*) type (number inhabitants $a+b$)
- REPL
- Cross platform
- Garbage collection or some other way of automatic memory management
- Decent ecosystem (quality libraries for all popular tasks)
- “Green” threads: for example, dedicating thread to client session and blocking while waiting for next event or IO completion is acceptable (Erlang, Go, Haskell)
- STM –software transaction memory
- Haskell has it all and much more



Language/Framework for front-end

- Strongly prefer statically typed language. After looking at few options Elm seemed as the good choice because of familiar syntax (belongs to ML family of languages) and really easy to get started.
- Good things mentioned earlier about system: ADT, Sum (Union) types, type inference
- Two in one: both framework (ELM architecture) and language (unlike React)
- Once it compiled , it just works (most of the times)
- Plays nicely with bootstrap (for design challenged people like myself)
- Good documentation, helpful error messages, examples, great community
- Time Traveling Debugger (you can see history of you model modification after every event)
- 100% pure (even more “pure” then Haskell)
- Simple interop with JS using ‘ports’
- Clear separation of concerns (model, view, update)



Demo app (service)

- Requirements:
 - Full stack
 - HTTP(S) server (to serve static content, at least)
 - Bi-directional communication with clients, hence WebSocket
 - Persistence (Database)
 - Share data types across server side, data base and client side
 - Stateful session tied to client websocket connection (Finite State Machine).
 - Horizontal scalability: allow adding more servers if needed.



Type driven design (very informal and simplified)

- Types first: define types that describes
 - Interaction between client and server (protocol)
 - State of client and server
 - Persistence
 - Shared data structures (for example UserInfo across parts of the application)
- Implement logic around these types. Thanks to using Sum types all cases will be covered, guaranteed (or at least assisted by compiler)
- Repeat: Refine and add features by modifying types (for example adding new branches to our Sum types) and let compiler to direct us what code needs to be added /modified.



Finally some Haskell code: Messaging with client

```
data ClientMsg = SessionCln SessionClnPs
               | UserMsgCln UserMsgClnPs
               | CallMsgCln CallMsgClnPs
               deriving (Show, Eq, Generic)
```

```
data ServerMsg = SessionSrv SessionSrvPs
               | UserMsgSrv UserMsgSrvPs
               | CallMsgSrv CallMsgSrvPs
               deriving (Show, Eq, Generic)
```



Session messages (client/server)

```
data SessionClnPs = SessAuthReq SessAuthReqPs
  | SessSetUserState SessSetUserStatePs
  | SessSignoff
  deriving (Show, Eq, Generic)
```

```
data SessionSrvPs = SessAuthAccept SessAuthAcceptPs
  | SessAuthReject SessAuthRejectPs
  | SessUserStateChanged SessSetUserStatePs
  | SessConnected
  | SessReload
  deriving (Show, Eq, Generic)
```




Authentication request/response

```
data SessAuthReqPs = SessAuthReqPs
```

```
{ token::Text  
  , currentEPID :: Maybe Text  
  , epInfo :: EPInfo  
  , version :: Version  
  , queryInv :: Bool  
  } deriving (Show, Eq, Generic)
```

```
data SessAuthAcceptPs = SessAuthAcceptPs
```

```
{ allocatedEPID :: Text  
  , userInfo :: UserInfo  
  , initState :: Maybe UserState  
  , lastInvite :: Maybe (Text, InviteSrvPs)  
  } deriving (Show, Eq, Generic)
```



WEB Socket

- Haskell: Web Socket Server
- ELM: Web Socket Client
- Supported by all modern browsers
- Unlike plain HTTP request/response model, allows bi-directional communication

WebSocket server

```
import qualified Pipes.Concurrent as PC
```

```
type InputOutput i o = (PC.Input i    -- recv :: STM (Maybe i)
                        ,PC.Output o -- send :: o -> STM Bool
                        )
```

```
type NewClientCallback i o = InputOutput i o -> IO ()
```

```
getWSApp :: (ToJSON o, FromJSON i, ToJSON i) => NewClientCallback i o -> IO
WS.ServerApp
```

```
newClientCallback :: NewClientCallback ClientMsg ServerMsg
```

<https://github.com/unomemento/webTools>



Web Socket Server – adding to WAI application

Package: `Network.Wai.Handler.WebSockets`

Upgrade a websockets `ServerApp` to a wai `Application`. Uses the given backup `Application` to handle Requests that are not `WebSocket` requests:

`websocketsOr :: ConnectionOptions ->`

`Network.WebSockets.ServerApp ->`

`Network.Wai.Application ->`

`Network.Wai.Application`



Sharing data types between server and client (Haskell and Elm)

- Write once (in Haskell) and generate:
 - JSON encoder/decoder for Haskell (ready to be used with Web socket server)
 - Data types for ELM
 - JSON encoder decoder for ELM
- Thanks for amazing package <https://hackage.haskell.org/package/elm-bridge> by Alexander Thiemann
- All we need to do:
 - Declare Haskell data type deriving Generic:



elm-bridge

```
{-# LANGUAGE DeriveGeneric #-}  
{-# LANGUAGE RankNTypes #-}  
{-# LANGUAGE TemplateHaskell #-}  
import Elm.Derive  
data UserState = UserOn  
    | UserOff  
    deriving (Show, Eq, Generic)  
  
data UserInfo = UserInfo  
    { userName :: Text  
    , userPhoto :: Maybe Text  
    } deriving (Show, Eq, Generic)  
  
data UserEntry = UserEntry  
    { userID :: Text  
    , userState :: UserState  
    , userCon :: Bool  
    , userInfo :: UserInfo  
    } deriving (Show, Eq, Generic)
```




Elm-bridge

```
deriveBoth defaultOptions "UserState"
deriveBoth defaultOptions "UserInfo"
deriveBoth defaultOptions "UserEntry"
-- and to generate Elm code use:
elmBasicData :: String
elmBasicData = makeElmModule "QvklyData"
    [ DefineElm (Proxy :: Proxy UserInfo)
    , DefineElm (Proxy :: Proxy UserState)
    , DefineElm (Proxy :: Proxy UserEntry)
    ....
    ]
--finally write actual read-to-use Elm module:
storeToFile :: IO ()
storeToFile = writeFile "QvklyData.elm" (Data.Text.pack elmBasicData)
```



Elm module created by elm-bridge

```
type alias UserInfo =  
  { userName: String  
    , userPhoto: (Maybe String)  
  }
```

```
jsonDecUserInfo : Json.Decode.Decoder ( UserInfo )
```

```
jsonDecUserInfo =  
  ("userName" := Json.Decode.string) >=> \puserName ->  
  (Json.Decode.maybe ("userPhoto" := Json.Decode.string)) >=> \puserPhoto ->  
  Json.Decode.succeed {userName = puserName, userPhoto = puserPhoto}
```

```
jsonEncUserInfo : UserInfo -> Value
```

```
jsonEncUserInfo val =  
  Json.Encode.object  
  [ ("userName", Json.Encode.string val.userName)  
    , ("userPhoto", (maybeEncode (Json.Encode.string)) val.userPhoto)  
  ]
```

WebSocket client at ELM

```
sendMsgToServer : ClientMsg -> Cmd msg
```

```
sendMsgToServer clientMsg = WebSocket.send Config.wsServerAddress (JE.encode 0 (jsonEncClientMsg clientMsg))
```

```
websocketSub : Sub WSMsg
```

```
websocketSub = WebSocket.listen Config.wsServerAddress decoderServerMsg
```

```
type alias MsgParseErrParams =
```

```
{ error : String  
  , buffer : String  
}
```

```
type WSMsg
```

```
= MsgReceived ServerMsg  
| MsgParseErr MsgParseErrParams
```

```
decoderServerMsg : String -> WSMsg
```

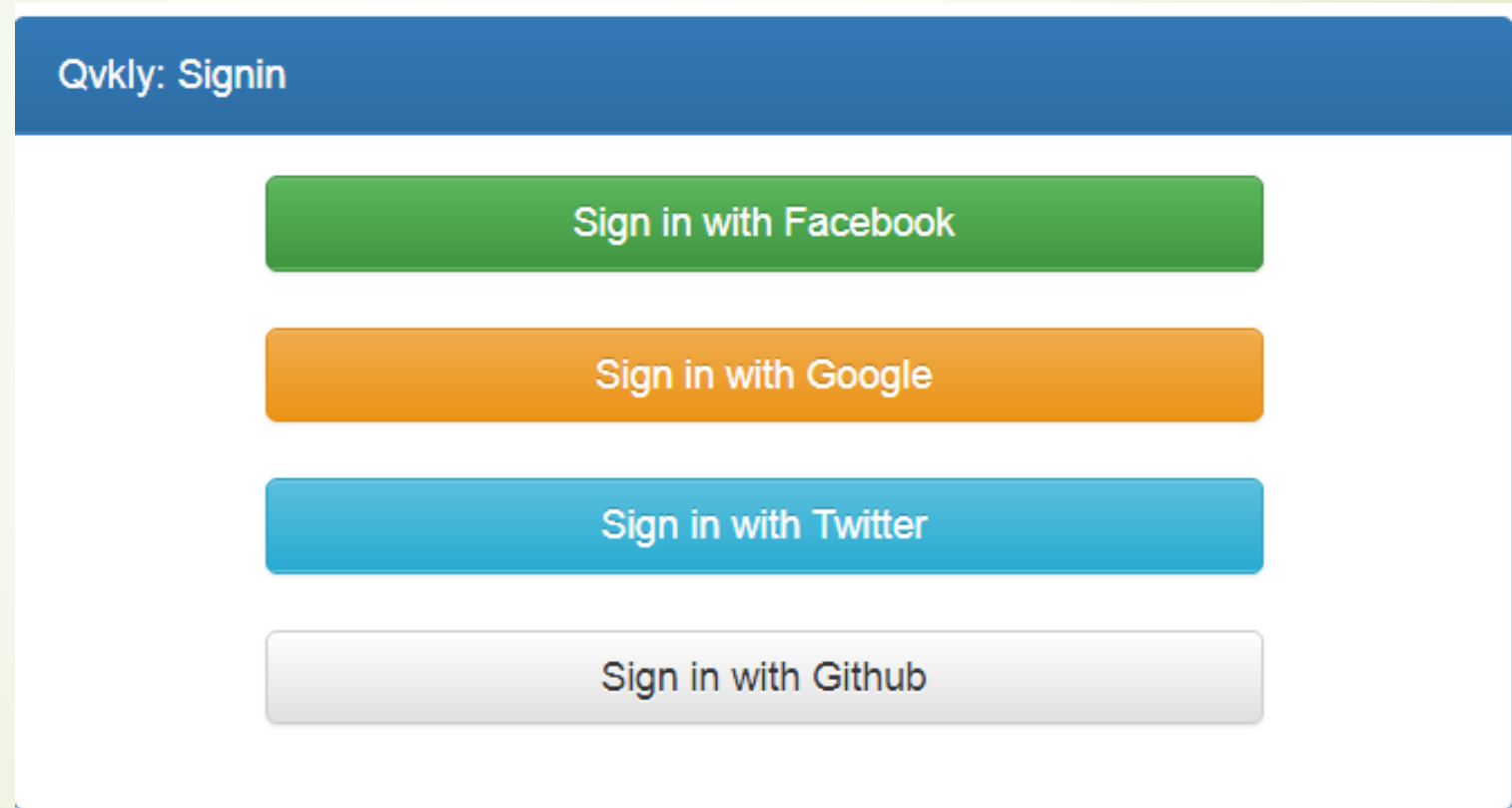
```
decoderServerMsg rcvdStr = case JD.decodeString jsonDecServerMsg rcvdStr of
```

```
  Ok serverMsg -> MsgReceived serverMsg
```

```
  Err err      -> MsgParseErr (MsgParseErrParams err rcvdStr)
```

Using Firebase for user authentication

- We want to allow to login user with their social accounts:



Qvkly: Signin

Sign in with Facebook

Sign in with Google

Sign in with Twitter

Sign in with Github



Google Firebase

- Unified API for authentication with Google, Facebook, Twitter, Github and more.
- Keeps logged in state across browser session (uses local browser store for persistence)
- Anonymous access (when applicable)
- Clear examples and instructions
- Access to public user info (picture and user name). More service specific permissions can be asked.
- Maintains database of all users of our application and generate userID that we can use as primary key in our database as well.
- Part of umbrella of Google Cloud services (that also includes GCM , Realtime Cloud database, Hosting, Storage , Cloud functions and much more): <https://firebase.google.com/products/>
- What is great about Authentication and GCM? They are free!
- API for web authentication is JS. Elm ports to rescue

Firestore authentication (Elm ports)

-- Requests

```
port signInGoogle : () -> Cmd msg
port signInFacebook : () -> Cmd msg
port signInTwitter : () -> Cmd msg
port signInGithub : () -> Cmd msg
port signOff : () -> Cmd msg
port requestMessagingPermission : () -> Cmd msg
port getMessagingToken : () -> Cmd msg
```

-- Subscriptions

```
port userSignedIn : (UserData -> msg) -> Sub msg
port gotUserToken : (String -> msg) -> Sub msg
port noUserSignedIn : (String -> msg) -> Sub msg
port signInError : (Error -> msg) -> Sub msg
port messagingPermissionError : (Error -> msg) -> Sub msg
port messagingTokenAvailable : (String -> msg) -> Sub msg
```


JS should be in sync with our Elm ports

```
function initFirebase(elmApp) {  
  firebase.initializeApp(config);  
  
  firebase.auth().onAuthStateChanged(function(user) {  
    if (user) {  
      console.log("Authenticated user:" + JSON.stringify(user))  
      elmApp.ports.userSignedIn.send(userData);  
      user.getIdToken(true).then(function(idToken) {  
        elmApp.ports.getUserToken.send(idToken);  
      }).catch(function(error) {  
        elmApp.ports.getUserTokenError.send(...)  
      });  
    } else {  
      elmApp.ports.noUserSignedIn.send("");  
    }  
  });  
}
```

Port requests (commands)

```
elmApp.ports.signInGoogle.subscribe(function() {  
  var provider = new firebase.auth.GoogleAuthProvider();  
  firebase.auth().signInWithPopup(provider).then(function(result) {  
    // This gives you a Google Access Token. You can use it to access the Google API.  
    var token = result.credential.accessToken;  
    // The signed-in user info.  
    var user = result.user;  
    // ...  
    //expect onAuthStateChanged to be triggered  
  }).catch(function(error) {  
    // Handle Errors here...  
    elmApp.ports.signInError.send(  
      { errCode : errorCode  
        , errMsg : errorMessage  
      }  
    );  
  }); //catch  
}); //signInGoogle => port signInGoogle : () -> Cmd msg
```

Elm Subscriptions

- Elm subscriptions are used to receive external events or operation completion results. All subscriptions are mapped to Elm messages. Every message must be handled by update function and can update model and/or generate new commands to external systems. For Firebase integration I defined the following messages:

```
type Msg
```

```
  = UserSignedIn UserData
```

```
  | GotUserToken String -- token needs to sent to server, validated and decrypted
```

```
  | NoUserSignedIn
```

```
  | SignInError Error
```

```
  | GetUserTokenError Error
```

```
  | MsgTokenAvailable String
```

```
  | NoMsgTokenAvailable
```

```
  | MsgPermissionError Error
```

```
  | MsgTokenError Error
```

```
  | MessageReceived String
```



Authentication request/response

```
data SessAuthReqPs = SessAuthReqPs
```

```
  { token::Text  
    , currentEPID :: Maybe Text  
    , epInfo :: EPInfo  
    , version :: Version  
    , queryInv :: Bool  
  } deriving (Show, Eq, Generic)
```

```
data SessAuthAcceptPs = SessAuthAcceptPs
```

```
  { allocatedEPID :: Text  
    , userInfo :: UserInfo  
    , initState :: Maybe UserState  
    , lastInvite :: Maybe (Text, InviteSrvPs)  
  } deriving (Show, Eq, Generic)
```

Handling authentication token in Haskell

- Download Google service certificate from:
gCertURL=<https://www.googleapis.com/robot/v1/metadata/x509/securetoken@system.gserviceaccount.com> using HTTPClient: Network.Wreq
- Extract public key using Data.X509 and Crypto.PubKey.RSA
- Decode token using Jose.Jws (rsaDecode key token)
- All that done within (<https://github.com/unomemento/googleTK>):
parseToken :: Text -> TokenParser -> IO (Maybe AuthData)

where:

```
Where data AuthData = AuthData
  { name :: Maybe Text
  , picture :: Maybe Text
  , email :: Maybe Text -- missing for twitter
  , email_verified :: Maybe Bool
  .....
  } deriving (Show, Eq, Generic)
```

- For more details: <https://jwt.io/introduction/>



Database selection

Was looking for document database with ability to store JSON serializable data.

Some options:

- Firebase Realtime (part of Google Cloud offering): easy to use with both client and server API (REST on server); live notification. Cons: free only for limited scale and less control (only hosted option exists)
- Mongo DB – very popular, nice Haskell package. Uses BSON (not JSON) for document representation. No live notification.



RethinkDB: database and much more

Best of all worlds:

- Store any JSON document
- Primary key and indices
- Joins
- Live notifications
- Distributed and highly available
- Very easy to get started: coming with nice web portal.

Very elegant composable query language (ReQL) . ReQL is implemented as AST (Abstract Syntax Tree), hence binding to Haskell was very natural (implemented by Etienne Laurin):

<https://www.youtube.com/watch?v=Qvn8EQfgUCA>



ReQL examples



JS

```
r.table('users').pluck('last_name').distinct().count().run(conn)
```



Haskell

```
run' :: Expr query => RethinkDBHandle -> query -> IO Datum
```

```
run' h $ table "users" # pluck "last_name" # distinct # count
```

Change feeds in Rethink DB

- Change feeds allow clients to receive changes on a table, a single document, or even the results from a specific query as they happen.
- Nearly any ReQL query can be turned into a change feed.

```
cursor <- run h $ table "posts" # changes :: IO (Cursor Datum)  
each :: Cursor a -> (a -> IO b) -> IO ()
```

- spawn thread to monitor changes in background:

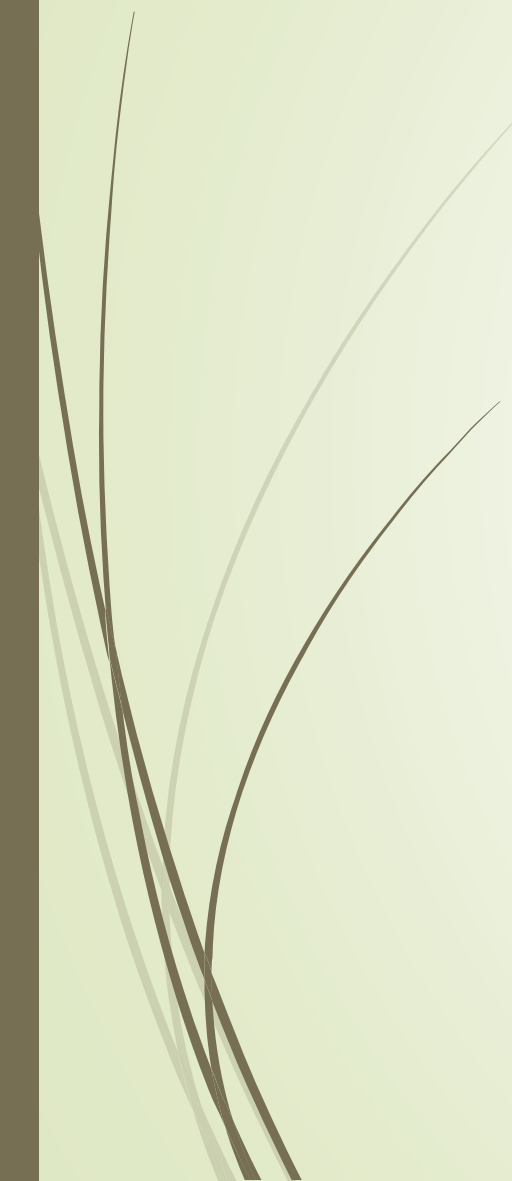
```
asyncTask <- async $ each handler
```

- to stop monitoring kill thread by cancelling async task:

```
cancel asyncTask
```



Using change feeds in real time communication (messaging and presence)

- Tables with new and mid-call messages
 - Once client is connected, change feed monitoring is established and client is notified with details of the message
 - Tables with user connectivity state and presence state (Available/Busy): Implementing presence become straightforward
 - “Unlimited scalability” thanks to distributed deployment
 - Free logging with advanced query possibilities
 - Tables may grow large should be periodically cleared
- 




Indices and Joins

- Unlike many noSQL database , RethinkDB supports indices and joins
- Allows eliminating excessive data de-normalization
- Allows nice separation of relatively static data and data that changes frequently (as user connectivity state)
- Limitation: change feeds are not supported for joins (that is probably good things as it would be very inefficient). The possible approach: establish change feed on small frequently changing tables and once change is detected invoke an additional query to retrieve user's details.



RethinkDB caveats

- As commercial project RethinkDB was not successful and company behind Rethink DB shutdown. Fortunately, the RethinkDB project was released as open source with business friendly license and there is group of enthusiasts including some former employees of the company that continue providing support and development
- The Haskell package is fairly low level and there is no validation that ReQL statement is matching DB structure, therefore run-time errors will happen in case of mismatch. Fortunately, error messages are very clear and errors can be fixed fast. Using REPL allow interactive testing and debugging of queries. Composability is also extremely helpful, as it allows writing incremental fashion.



No FP talk by FP beginner is complete without mentioning monad.

► Douglas Crockford quote:

"In addition to it begin useful, it is also cursed and the curse of the monad is that once you get the epiphany, once you understand - 'oh that's what it is' - you lose the ability to explain it to anybody."

► I believe there are many level of explaining and understanding the monad including one by Stephen Diehl "Monad made difficult" :

<http://www.stephendiehl.com/posts/monads.html>

I will try to present very quick explanation using as an example monad Maybe (just value with effect of possibility of value absence):

data Maybe a = Just a | Nothing

From Functor to Monad (super simple explanation)

- ▶ Functor : applying pure function to value with effect [wrapped value]

`fmap :: (a -> b) -> f a -> f b`

For example:

`fmap (*2) (Just 5)`

`Just 10`

- ▶ Applicative: Applying : applying pure function with 2 or more arguments to values with effect

`liftA2 :: (a -> b -> c) -> f a -> f b -> f c`

For example:

`liftA2 (*) (Just 5) (Just 2)`

`Just 10`

...and finally Monad: like Functor but with join

- Functor : applying pure function to value with effect [wrapped value]

`fmap :: (a -> b) -> f a -> f b`

For example:

`fmap (*2) (Just 5)`

`Just 10`

- Monad : applying function with effect to value with effect. Let's define function with effect:

`doubleIfPositive :: Int -> Maybe Int`

`doubleIfPositive i = if i > 0 then Just (i * 2) else Nothing`

Then let's just use functor's `fmap`:

`fmap doubleIfPositive (Just 5)`

`Just (Just 10)`

What monad brings us on top of functor is an ability to 'join' two effects into one:

`join :: f (f a) = f a`

Or

`join (Just (Just 10))`

`Just 10`

-- Or idiomatic way : `Just 5 >=> doubleIfPositive`

`:t (>=>)`

`(>=>) :: Monad m => m a -> (a -> m b) -> m b` – like Functor with flipped parameters , followed by join



Back to our server app: bring it all together

- ▶ Every user session is handled by independent entity (component) that:
 - ▶ Maintain client session state (FSM: finite state machine) with no concurrency/synchronization concern
 - ▶ Exchanges messages with client
 - ▶ Interacts with database to update user state, preferences, push token etc
 - ▶ Send messages and push notifications to other users
 - ▶ Receive messages from other users
 - ▶ All that could be done based on specific system configuration
- ▶ All that sounds pretty straightforward and many of us have been implementing this kind of task using their preferred imperative languages.
- ▶ There are many way to implement that in Haskell too and I will present one of the possible approaches that I actually used

Let's define context that we can use to handle user's session

```
type SessionHandler = StateT ClientState (ReaderT AppEnv IO)
```

- Which is type alias for monad transformer (stack of monads) that describe computational context that addresses our needs mentioned earlier:
- `StateT ClientState` – indicates that we need to maintain (read/write/modify) state (of type `ClientState`) – State monad
- `ReaderT AppEnv` – indicates that we need access to application environment (of type `AppEnv`)
- `IO` – indicates that our computation is not pure and IO is required



Handling events (client message, remote messages, users' presense notifications, etc)

```
data ClientSignal = ClientComSignal ClientComSignalPs
                  | CallSignal CallSignalPs
                  | UserSignal UserSignalPs
                  deriving (Eq, Show)
```

```
clientSignalHandler :: ClientSignal -> SessionHandler ()
clientSignalHandler clientSignal = do
  sessionDataMaybe <- gets sessionData -- get sessionData from the state
  case sessionDataMaybe of
    (Just sd) -> sessionSignalHandler sd clientSignal
    Nothing -> noSessionSignalHandler clientSignal
```


Start handling of client's connection

Recall from Web Socket server discussion we needed to pass handling function that is invoked when new web socket client gets connected:

```
type InputOutput i o = (PC.Input i    -- recv :: STM (Maybe i)
                        ,PC.Output o -- send :: o -> STM Bool
                        )
```

```
type NewClientCallback i o = InputOutput i o -> IO ()
```

```
getWSApp :: (ToJSON o, FromJSON i, ToJSON i) => NewClientCallback i o -> IO WS.ServerApp
```

```
newClientHandler :: AppEnv -> NewClientCallback ClientMsg ServerMsg
```

```
newClientHandler appEnv (input, output) = do
```

```
  (outSignal, inSignal) <- PC.spawn PC.unbounded --create pipeline from incoming msg toClientSignal
```

```
  void $ async $ P.runEffect $ clientMsgProducer (PC.fromInput input) >-> PC.toOutput outSignal
```

```
  let initState= initClientState output inSignal outSignal
```

```
  void $ async $ void $ -- create pipe line from ClientSignal to signal handling function
```

```
    runReaderT (runStateT (P.runEffect (P.for (PC.fromInput inSignal) effFunc)) initState) appEnv
```

```
effFunc :: ClientSignal -> P.Effect SessionHandler ()
```

```
effFunc clientSignal =lift $ clientSignalHandler clientSignal
```



There is a lot more to share but I guess
you are already tired of me.

Questions?