Fast and Fearless Evolution of Server-Side Web Applications

Oskar Wickström

Outline

- Introduction
- Web Applications
- Writing Web Applications with Haskell
  - Scotty
  - Yesod
  - Airship
- Client-Side Technologies

- Today, after a high-level introduction, we'll talk about web applications
- Specifically, we will talk about writing webapps in Haskell
- I'll demonstrate some frameworks available
- And say a few words about client-side technologies

Introduction

Evolving Software

- New features
- Bug fixes
- Refactoring
- External factors
  - New regulation
  - Deprecation of software and services
- Recruiting

- There are many reasons to evolve software
- We will probably have ...

Risks

- Evolving software, we face risks:
  - Delay
  - Error
  - Burnout
- These risks make it scary to evolve freely

- When evolving software, we risk: ...
- These things hurt not only ourselves, but: Customers, partners, internal relations
- The risks make it scary to evolve our software freely
- The change you want to make can be too risky
- I think the fear of evolving software freely has a huge impact on our systems

Tooling

- We can reduce fear with better tools
  - Stronger correctness guarantees
  - Robustness
  - Faster feedback
  - Communicates intent
- Functional programming
- Type systems
- Error handling

Web Applications

Web Applications

- Many of us work with the web somehow
- Single-page apps (SPAs) are in vogue
  - More like desktop apps
  - Reinventing parts of the browser
  - No Javascript, no application
- Universal webapps (aka “isomorphic”)

- Many of us work with the web in one way or another ... (internet)
- Currently, single-page apps are trendy. These work more like ...
- Now, you might think "What about universal webapps?"
- They are about initial rendering, not about transparently running the same application client-side and server-side

Reminder

$$\text{Newer} \not\Longrightarrow \text{Better}$$

@owickstrom

- This is my friendly reminder: Never does not imply better
- While the single-page app tech is very new and flashy, it might not be a good choice for your project

Server-Side Web Applications

- Do not dismiss server-side web applications
- Progressive enhancement
- 80/20 rule
- Use client-side code where you need it!
- PJAX

- I urge you not to dismiss server-side web applications
- Rather, have that as a default choice
- You can use what's known as "Progressive Enhancement", where ...
- Also important to recognize: All code is not equally valuable
- Pages for Settings, Login, Documentation, etc
- Use client-side code where you need it, where you get a return on the investment
- If you want more snappy navigation, things like PJAX go a long way

Static Typing for Server-Side Web

- Compile-time checking
  - Run-time robustness with defined behaviour
  - Use types for correct-by-construction
  - Machine-verified living documentation, communicates intent
- Safely evolve our codebase
  - Reduce fear of change throughout the codebase
  - Modify core domain, follow the errors
  - Not split by an API
- Focus tests on our domain
  - No need to write tests for type errors
  - Domain code free of side effects

- Combining server-side web with static typing, we get a lot of benefits

Functional Statically Typed Web

- Many languages, many frameworks!
- Look for the patterns and safety
- Less power is more power
- Today's focus is Haskell

- There are many languages and frameworks in this spirit
- Base your decisions on the underlying patterns and safety guarantees
- This often comes down to: Less power is more power
- As Michael explained this morning, as you can't sneak in side-effects in Haskell, libraries like STM can give strong guarantees
- My examples will use Haskell for that reason

Writing Web Applications with Haskell

Underpinnings

- Web Application Interface (WAI)
  - Common interface betwen web applications and web servers
  - Mix frameworks in one application
  - Comparable with Java Servlet API
- Warp
  - WAI web server
  - Uses GHC's lightweight threads

- The Haskell web frameworks we'll look at all build on WAI, ...
- Warp is a popular and fast web server for WAI

Frameworks

- Scotty
- Spock
- Yesod
- Happstack
- Snap
- Airship
- Servant
- MFlow

- In the Haskell ecosystem, there are many web frameworks.
- This list is not exhaustive
- We will look at three of these frameworks: Scotty, Yesod, and Airship

Scotty

Scotty

- Inspired by Ruby's Sinatra
- Features
  - Routing and parameters
  - Web server setup
  - Extensible
- "Build your own framework"

- Inspired by Ruby's Sinatra
- It provides routing, parameters, and form parsing
- It is easy to get started, setting up a web server
- Scotty is extensible. I'm not going to say it, but it has to do with the M-word.
- Scotty is very small, and if you build something bigger, you'll likely have to "build your own framework"

Scotty Routing

```haskell
app :: ScottyM ()
app = do

  get "/" $
    html "Welcome!"

  get "/greet/:who" $ do
    who <- param "who"
    html ("Hello, " <> who <> "!")
```

@owickstrom

- This is a Scotty app with two routes
- (explain code)

Scotty Server

```haskell
main :: IO ()
main = scotty 8080 app
```


- This is how we run it
- Now, usually you need to render larger chunks of HTML

HTML Templates

```
get "/greet-with-template/:who" $ do
  who <- param "who"
  html $
       "<!DOCTYPE html>\
       \<html lang=\"en\">\
       \<head>\
       \  <meta charset=\"UTF-8\">\
       \  <title>My Page</title>\
       \  <link rel=\"stylesheet\"\
       \        href=" <> bootstrapCss <> "\">\
       \</head>\
       \<body>\
       \  <div class=\"jumbotron\">\
       \     <h1>Hello, " <> who <> "!</h1>\
       \  </div>\
       \</body>\
       \</html>"
```

@owickstrom

- Let's say we do this
- It is very hard to read
- Sure, we could refactor to separate view functions
- The bigger issue is that we're doing stringly-typed programming
- **Can anyone tell me what's wrong here?**

HTML Template Error!

```
get "/greet-with-template/:who" $ do
  who <- param "who"
  html $
       "<!DOCTYPE html>\
       \<html lang=\"en\">\
       \<head>\
       \  <meta charset=\"UTF-8\">\
       \  <title>My Page</title>\
       \  <link rel=\"stylesheet\"\
       \       href=" <> bootstrapCss <> "\">\
       \</head>\
       \<body>\
       \  <div class=\"jumbotron\">\
       \     <h1>Hello, " <> who <> "!</h1>\
       \  </div>\
       \</body>\
       \</html>"
```

@owickstrom

- We are missing an escaped double quote here
- The string literal's quote makes it extra hard to see
- So, let's not do this.

DSLs for HTML

- Instead of HTML in strings, we use DSLs
- Embedded:
  - Blaze
  - Lucid
- External:
  - Heist
  - Hamlet
- Type safety
- Composable

- Instead we use a markup DSL
- There are embedded and external DSLs for HTML
- Embedded means the markup is written in regular Haskell, in Haskell source files
- Two popular libraries are Blaze and Lucid
- For external HTML templating languages, we can use Heist or Hamlet
- The external ones are typically written in separate files, but can also be embedded using quasi-quoting
- These languages give us type-safe templates that are composable
- They help us produce valid HTML

Lucid HTML Template

```
homeView :: Text -> Html ()
homeView who =
  html_ [lang_ "en"] $ do
    head_ $ do
      meta_ [charset_ "UTF-8"]
      title_ "My Page"
      link_ [rel_ "stylesheet", href_ bootstrapCss]

    body_ $
      div_ [class_ "jumbotron"] $
        h1_ ("Hello, " <> toHtml who <> "!")
```

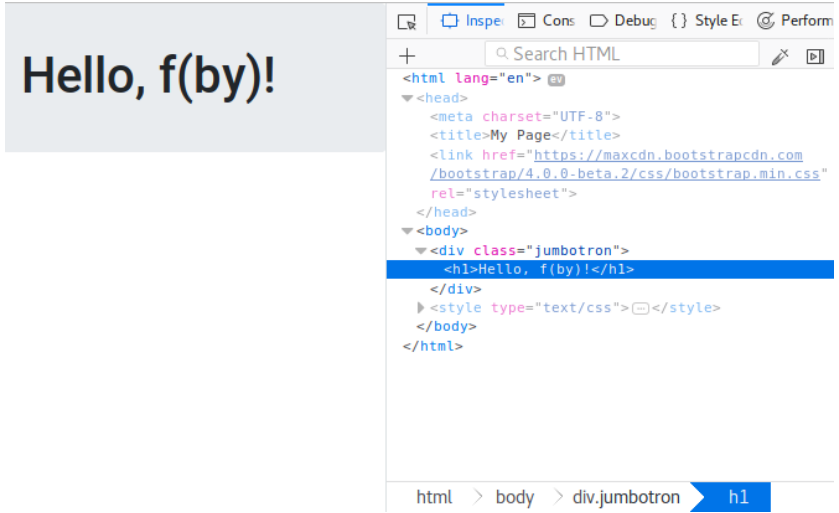

- Here we see the equivalent template in Lucid
- Elements are nested using function application
- Elements are juxtaposed using do notation
- Attributes are set using a list of pairs
- Notice how some functions do not take any child content
- ‘meta‘ and ‘link‘ in HTML are empty elements
- In this way, Lucid and the type system help us construct valid HTML

Rendering Lucid with Scotty

```
get "/greet-with-lucid/:who" $ do
  who <- param "who"
  html (renderText (homeView who))
```


- We can render Lucid in a handler like this

Result



@owickstrom

- Looking at the result in a web browser, we can inspect the rendered HTML

Side Effects in Scotty

- We need more than sending HTML responses
- We want to do IO:
  - Database queries
  - Logging
  - External service calls
- IO in Scotty handlers using `liftIO`

- So far, we have only sent HTML over the wire
- We most likely need side-effects to do something useful
- (read list)
- We use ‘liftIO‘ in Scotty handlers to do IO

IO in Scotty

- Given these definitions:

```
type ArticleId = Text

addNewComment :: ArticleId -> Text -> IO ()
```

- We can *lift* the IO action into a handler:

```
post "/articles/:article-id/comments" $ do
  articleId <- param "article-id"
  -- accepts a form or query parameter "comment"
  comment <- param "comment"
  liftIO (addNewComment articleId comment)
  redirect ("/articles/" <> articleId)
```


- Let's look at an example of using 'liftIO'
- Given these definitions... (explain rest)

Starting with Scotty

- Easy to get started, learn the basics
- What you don't get:
  - Templating
  - Sessions
  - Authentication and Authorization
  - Logging
  - Persistence
- Have a look at Spock[1] for more features

[1] https://www.spock.li

- I recommend starting out with Scotty if you're new to Haskell web development
- Once your applications grows, you will probably need to bring in libraries
- You might need to bring in ... (read list)
- For a slightly larger feature set, and type-safe routing, have a look at Spock

Yesod

Yesod

- “One-stop shop” for Haskell web development
  - A framework
  - Batteries included
  - Still very modular
- Also runs on WAI

- Yesod can be called a "one-stop shop" for Haskell web development
- It is a framework with many batteries included
- Still, it is implemented to be modular
- Things are there by default, but you can swap them out if you need
- Yesod runs on WAI using the Warp server

Batteries Included with Yesod

- Type-safe routing
- External templates for:
  - HTML
  - CSS
  - Javascript (and TypeScript)
- Widgets
- Forms
- Sessions
- Integration with Persistent
- Authentication and Authorization
- Internationalization
- Logging
- Configuration
- Auto-reloading web server

- These are some of the features you get ... (read list)

Getting Started

- Use a template (see `stack templates`)
- There will be things you don't understand at first
- Start out exploring:
  - Routing
  - Templates (HTML, CSS, Javascript)
  - The "Foundation" type
  - Getting something done!
- Over time, you'll understand the scaffolding
- Use the auto-reloading web server
  - Install `yesod-bin`, run `yesod devel`

- I recommend starting by using a template
- (read list)

Routes Configuration

```
/ HomeR GET

/articles/#ArticleId ArticleR GET
```


- Routes are configured in a separate file
- We define a root path ...

A Simple Handler

```
getHomeR :: Handler Html
getHomeR = do
  articles <- allArticles
  defaultLayout $ do
    setTitle "My Blog"
    $(widgetFile "homepage")
```


- The handler for a route is found by a naming convention
- "HomeR GET" corresponds to the "getHomeR" definition

Hamlet Template

```
<h1>My Blog
<ul>
  $forall article <- articles
    <li>
      <a href=@{ArticleR (articleId article)}>#{articleTitle article}
```

@owickstrom

- This is "homepage.hamlet"

Routing with Path Pieces

```
getArticleR :: ArticleId -> Handler Html
getArticleR id' = do
  article <- getArticle id'
  comments <- getArticleComments id'
  defaultLayout $ do
    setTitle (Html.text (articleTitle article))
    $(widgetFile "article")
```

@owickstrom

- Routes with "PathPieces" captured have handlers with arguments
- The route for a particular article captured an "ArticleId" in the routes file
- This handler therefore has an "ArticleId" argument

Article Hamlet Template

```
<h1>#{articleTitle article}
<p>#{articleContent article}

<div .comments>
  <h2>Comments
  <ul>
    $forall comment <- comments
      <li .comment>
        <span .author>#{commentAuthor comment}
        <span .content>#{commentContents comment}
```

Widgets

- Reusable components of HTML, CSS, and Javascript
- We used widgets in handlers:

```
$(widgetFile "article")
```

- Yesod tries to find matching widget files:

```
templates/article.hamlet
templates/article.cassius
templates/article.lucius
templates/article.julius
```

- Can refer to bindings in Haskell code
- Only include small parts, or use external resources

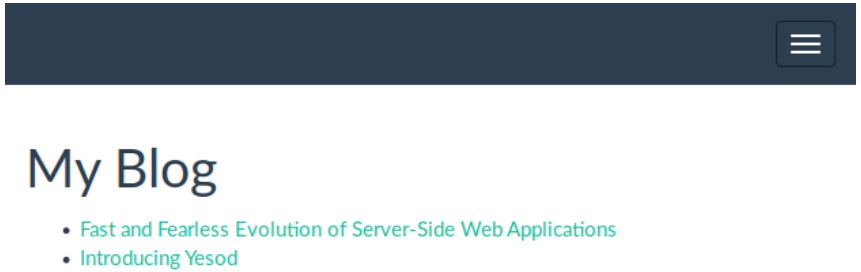
- Let's have a look at the concept of a "Widget"
- (read list)

Lucius (CSS Templates)

```
.comments {
  margin-top: 3em;
}
.comments ul {
  list-style-type: none;
  padding: 0;
}
.comment {
  background: #eee;
  padding: .5em;
  margin-bottom: 1em;
}
.author {
  font-weight: bold;
}
.author:after {
  content: ':';
}
```


- Here's some simple styling of comments
- This is a Lucius file
- You can interpolate Haskell values, but I'm not doing that here

Home Page Result



- Using a web browser, the home page looks like this

Article Page Result

Introducing Yesod

Lorem ipsum dolor sit amet...

Comments

Carol: Wonderful post! Keep them coming.

Mallory: I have have opinions. You suck.

@owickstrom

- And the article page now has comments rendered

Yesod Forms

- Write forms using applicative or monadic style
- Use the same structure for rendering, parsing, and validation
- There are various renderers available

- The next powerful feature of Yesod that I want to show is forms
- (read list)

Comment Form

```
commentForm :: AForm Handler Comment
commentForm =
  Comment
  <$> areq textField (named "Name") Nothing
  <*> (unTextarea
       <$> areq textareaField (named "Comment") Nothing)
```


- This is a definition of a Yesod Form
- It is a form for Comments
- It has a text field for the commenter name
- And it has a text area for the comment contents
- This is a single definition for both rendering the form, and parsing incoming form data

Rendering a Form

```haskell
getArticleWithFormR :: ArticleId -> Handler Html
getArticleWithFormR id' = do
  article <- getArticle id'
  comments <- getArticleComments id'

  (commentFormWidget, commentFormEnc) <-
    generateFormPost (renderForm commentForm)

  defaultLayout $ do
    setTitle (Html.text (articleTitle article))
    $(widgetFile "article-with-form")
```


- Here's how we use the form in a handler
- First, we get the article and its comments
- Then, we use "generateFormPost"
- We get a widget back, that is included in the HTML template below

Including The Form Widget

```
<form role=form
      method=post
      action=@{ArticleCommentsR id'}
      enctype=#{commentFormEnc}>
  ^{commentFormWidget}
  <button type="submit" .btn .btn-default>Submit
```

Parsing and Validating the Form

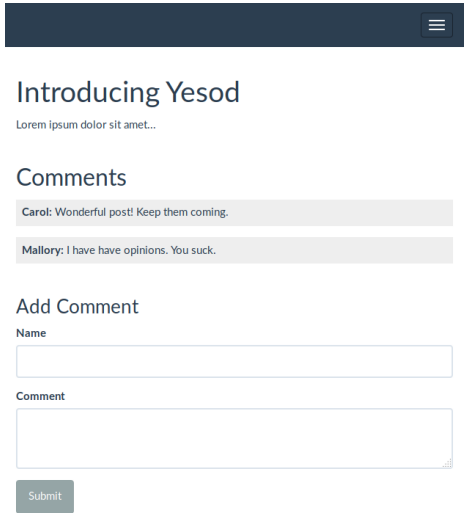
```
postArticleCommentsR :: ArticleId -> Handler Html
postArticleCommentsR id' = do
  article <- getArticle id'
  comments <- getArticleComments id'

  ((result, commentFormWidget), commentFormEnc) <-
    runFormPost (renderForm commentForm)

  case result of
    FormSuccess comment -> do
      addArticleComment id' comment
      redirect (ArticleWithFormR id')
    _ ->
      defaultLayout $ do
        setTitle (Html.text (articleTitle article))
        $(widgetFile "article-with-form")
```


- The POST handler uses "runFormPost" to parse the form data, and render a new form
- If it was success in parsing it, we add the comment and redirect back
- Otherwise, we rerender the form
- The neat thing is that Yesod will render validation errors automatically

Comment Form Result



@owickstrom

- In the browser, we see the rendered form

Yesod Recap

- Very capable, hit the ground running
- We only looked at some core features
- Worth learning

- To summarize the part on Yesod ...
- Yesod has much of you might need
- You don't have to build your own framework, at least not for some time
- I think it's worth learning if you want to build web apps in Haskell
- But if you're beginning Haskell, start with Scotty or Spock

Airship

Airship

- Inspired by Webmachine from Erlang
- Define RESTful resources
- Override fields in the default resource
- Tie together resources with routing

- Airship is inspired by Webmachine from Erlang
- It is centered around RESTful resources
- You use the default resource, which does all the sensible defaults
- Then you override methods to implement your resource
- Resources are tied together using a routing DSL (...)

Airship Routes

```
appRoutes :: Resource IO -> RoutingSpec IO ()
appRoutes static = do
  "articles" </> var "articleId" #> articleResource
  "static"   </> star            #> static
```


- In this example, we define two routes ...

Defining Resources

```
articleResource :: Resource IO
articleResource =
  defaultResource
  {
    -- overrides ...
  }
```


- A resource overrides fields in the default resource

resourceExists

```
...

  , resourceExists =
      routingParam "articleId" >>= articleExists

...
```

@owickstrom

- This overrides basically answers: is this a 404 Not Found?

contentTypesProvided

```
...

  , contentTypesProvided =
    let htmlResponse (Just article) =
          return (textResponse (renderArticle article))
        htmlResponse Nothing =
          return response404
    in return [("text/html", routingParam "articleId"
                             >>= getArticle
                             >>= htmlResponse)]

...
```


- This override is a bit more involved
- Here we bind a special function for constructing an HTML response
- Then, in case the content type is text/html, we get the article, and render it
- If it has another content type, based on the Accept header, we do nothing
- Airship handles that for us

404 Not Found

```
$ curl -i 'localhost:3000'
HTTP/1.1 404 Not Found
Transfer-Encoding: chunked
Date: Tue, 12 Dec 2017 15:43:29 GMT
Server: Warp/3.2.13
Content-Type: text/html

Not found!
```


- Let's test our server using curl
- Requesting a non-existing resource we get 404 Not Found

405 Method Not Allowed

```
$ curl -i -X PUT 'localhost:3000/articles/1'
HTTP/1.1 405 Method Not Allowed
Transfer-Encoding: chunked
Date: Tue, 12 Dec 2017 15:44:21 GMT
Server: Warp/3.2.13
Allow: GET,HEAD,POST
```


- Requesting the article with PUT we get 405 Method Not Allowed

406 Not Acceptable

```
$ curl -i -H 'Accept: text/plain' 'localhost:3000/articles/1'
HTTP/1.1 406 Not Acceptable
Transfer-Encoding: chunked
Date: Tue, 12 Dec 2017 15:48:27 GMT
Server: Warp/3.2.13
```


- Passing "Accept: text/plain", we get 406 Not Acceptable

200 OK

```
$ curl -i 'localhost:3000/articles/1'
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Tue, 12 Dec 2017 15:45:29 GMT
Server: Warp/3.2.13
Content-Type: text/html

<h1>Airship Webmachines!</h1><p>Lorem ipsum...</p>
```


- And finally, we can do a GET request and get the HTML

Airship Overrides

```
allowMissingPost
allowedMethods
contentTypesAccepted
contentTypesProvided
deleteCompleted
deleteResource
entityTooLarge
  forbidden
generateETag
implemented
isAuthorized
isConflict
knownContentType
lastModified
languageAvailable
malformedRequest
movedPermanently
movedTemporarily
multipleChoices
previouslyExisted
  processPost
resourceExists
serviceAvailable
uriTooLong
validContentHeaders
```


- I only showed you two, but there are many more overrides

Airship Considerations

- It is more low-level/barebones
- Again, “build your own framework”
- Suited for RESTful APIs

- Airship is a bit rough on the edges
- I didn’t show you the setup to get it running
- It requires some effort to integrate with other libraries
- But if you’re building a RESTful API, Airship might be a great choice!

Client-Side Technologies

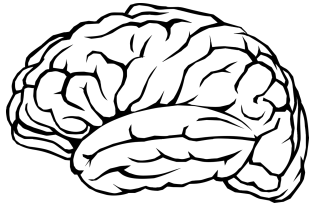
Client-Side Technologies

Need to do a single-page app?

- PureScript, Elm, etc
- Consider Haskell for your backend
- With Servant, you can use `servant-purescript` or `servant-elm`

- If you need to do a single-page app, or a single-page component of a greater system, consider...
- These are in the same spirit, more or less
- Consider Haskell for your backend, even it's only serving JSON
- You can use all the frameworks I've shown to build web APIs
- Also, there is Servant, to get a lot of type-safety in web APIs
- Servant can be integrated with PureScript and Elm to share types

Summary



Evolve software fearlessly using better tools
for modeling and communication.

@owickstrom

- Evolve software fearlessly using better tools for modeling and communication.
- Your program is a communication between you, your colleagues, and the computer
- Use tools that support that communication
- Use tools that support **evolving** your ideas, not only implementing your first idea

Spend your complexity budget carefully.

@owickstrom

- Spend your complexity budget carefully.
- Large parts of your web application are of lower value than core business parts
- Reach for simple tools with less risk in those areas
- When needed, use more advanced and complex tools where you get return on investment

Explore the wonderful world of functional
and statically typed server-side web.

@owickstrom

- There is so much good stuff in statically typed functional programming
- Combine that with server-side web development and you have a very good toolbox
- Thank you for listening! Here are links to...

Links

- Slides and code:
  github.com/owickstrom/fast-and-fearless-evolution-of-server-side-webapps
- Website: https://wickstrom.tech
- Twitter: @owickstrom

Questions?