Developing Web Applications with Haskell

Hello!

I am Alejandro Serrano

Also known as **@trupill** in Twitter, and **serras** in GitHub

PhD student at Utrecht University in The Netherlands

Daily user of Haskell (sometimes Idris, too)

Author of *Beginning Haskell*

Past contributor to EclipseFP and ghc-mod

Format

The workshop is divided in four blocks

- Routing + returning HTML
- Database access
- *Break* (approx. 10 minutes)
- Forms + validation
- Middleware + deployment

Within each block

- General overview of the topics
- Hands-on exercises

Exercises

github.com/serras/lambdaconf-2015-web

- A template for the solution is given
- Beginner and advanced exercises
- You may not have time to finish everything here
- Q+A about exercises in the break
- Suggestion: work in pairs

0 Introduction

How does the web development landscape look like in Haskell?

Lots of libraries

Frameworks:

- Happstack
- Snap
- Yesod
- Scotty
- **Spock**

Databases:

- HaskellDB
- HaSQL
- **Persistent**

HTML / Templating:

- **Blaze**
- Lucid
- **Shakespeare**
- Hastache
- Heist
- HSP

Lots of **modular** libraries

- Persistent was initially part of Yesod
- Blaze templating was developed before most of the web frameworks
- Routing functionality from Spock is a separate library
- And many other examples

How is this achieved?

- Strong contracts via **typing**
- General **concepts**, such as monoids or functors

Not a definite choice

Libraries presented in this talk may not suit the needs of every web application in the wild

- For big projects, use Yesod or Snap

Instead, they were chosen because of

- Simplicity
- Examples of Haskell approach to development
  - Typing to enforce invariants
  - Interface based on common concepts

Running the examples

Each example/exercise is a Cabal project

- Cabal is the Make/Ant/Maven/npm/NuGet of Haskell
- Builds project and downloads dependencies

1. Declare dependencies and targets in `.cabal` file
2. Optionally sandbox: `cabal sandbox init`
3. Install deps: `cabal install --only-dependencies`
4. Build the project: `cabal build`
5. Run the result: `./dist/build/executable/executable`

1 Routing

Hello, $name

```
main :: IO ()
main = runSpock 8080 $ spockT id $ do
  get ("hello" <//> var) $ \name ->
    text ("Hello, " <> name)
```

Spock is the framework we are going to use

- Sinatra/Scalatra/Flask style
- This code starts serving at port 8080

Structure of a handler

A Spock application consists of **handlers**

```
get ("hello" </> var) $ \name ->
    text ("Hello, " <> name)
```

Verb: get, post, put, delete, patch

Route: defined using three combinators

- static, or a literal string
- var to specify placeholders
- </> to separate elements

Action to perform

Routes are type-safe

```
{-# LANGUAGE ScopedTypeVariables #-}

get ("allow" <//> var) $ \(age :: Integer) ->
  if age < 21
     then text "You are not allowed"
     else text "Please, come in"
```

- http://host/allow/3 matches the route
- http://host/allow/Peter does not match

Actions are **monadic**

```
get ("hello" <//> var) $ \name -> do
  setCookie "name" name 3600
  text ("Hello, " <> name)
```

- The response is the combination of all actions
- The most important actions set a content type and the information to be returned
  - `text` returns plain text
  - `html` is used to return HTML

You could do this...

```
get ("hello-html" <//> var) $ \name ->
  html ("<html><body><h1>Hello, "
        <> name
        <> "</h1></body></html>")
```

- Does not enforce well-formed HTML
- It is hard to maintain
- What about injection attacks?

2 Returning HTML

Enforcing well-formedness and increasing maintainability

Let me introduce...

- **Blaze**: based on a pseudo-monad
  - Haskell code imitates HTML structure
  - It is a *domain specific language*
    - All Haskell bells and goodies available
  - **Lucid** is also based on this approach
- **Shakespeare**: uses quasi-quotations
  - Block with its own syntax
  - Translates to Blaze
- **Heist**: templates with bound data
  - Separate from the actual code
  - Similar to JSP/PHP/...

HTML with Blaze

```
get ("hello" <//> var) $ \name ->
  html $ toStrict $ renderHtml $
    B.html $
      B.body $
        B.h1 $ do
          B.text "Hello "
          B.span ! A.style "color: red;" $
            B.text name
```

- `text` takes care of injection attacks
- `(!)` attaches attributes to nodes

Shakespearean HTML

```
get ("allow" </> var) $ \(age :: Integer) ->
  if age < 21
    then html $ toStrict $ renderHtml $
         [shamlet|<html>
                    <body>
                      <h1>
                        You are
                        <span style="color: red;">not
                        allowed
                      <p>You are only #{age} years old |]
    else [shamlet| … |]
```

- This block generates Blaze code
- #{...} uses the value in a binding


Time for fun!

Two exercises

1. Returning JSON instead of HTML
   a. Introduces the `aeson` library
   b. Introduces generics
   c. Work with basic routing
2. Keeping an application state
   a. More advanced stuff
   b. Software Transactional Memory

Don't be shy: ask questions!

- We will share some solutions in the break

3

Databases

You need to save the data somewhere, don't you?

Two approaches

Access to a particular DB:

- postgres-simple
- mysql-simple
- sqlite
- mongoDB
- hedis
- rethinkdb

Common access layer:

- Relational algebra
  - HaskellDB
  - HaSQL
- "Functional ORM"
  - **Persistent**

Define a schema, 1/2

```
mkPersist sqlSettings [persistLowerCase|
User json
  firstName  String
  lastName   String
  UniqueName firstname lastname
  deriving Show
Task json
  title  String
  user   UserId
  deriving Show
|]
```

This quasi-quoter generates a lot of code!

Define a schema, 2/2

- A `User` data type with two fields
  - This is the one we work with
- A `UserKey` data type to encode identifiers
- A data type `EntityField User` which allows us to refer to a specific field in a record
  - `UserId | UserFirstname | UserLastname`
- A data type representing unique constraints
  - `UniqueName String String`
- Instances used for DB and JSON serialization

And the same for `Task`!

Handling DB connections

```
runNoLoggingT $
  Sqlite.withSqlitePool "example.db" 10 $ \pool ->
    let withDb f = liftIO $ runSqlPersistMPool f pool
     in NoLoggingT $ runSpock 8080 $ spockT id $ do
          get ...
```

- Initialize logging (required by Persistent)
- Create a new pool of 10 connections to example.db
- Run a transaction in the context of that pool

liftIO

```
runNoLoggingT $
  Sqlite.withSqlitePool "example.db" 10 $ \pool ->
    let withDb f = liftIO $ runSqlPersistMPool f pool
     in NoLoggingT $ runSpock 8080 $ spockT id $ do
          get ...
```

Database access is not pure and involves I/O

- We need to run it inside the `IO` monad
- Handlers run in their own `SpockT` monad
- `liftIO` bridges between both monads

Insertion

```
get ("user" <//> "new" <//> var <//> var) $
  \fname lname -> do
     user <- withDb $ insertUnique (User fname lname)
     case user of
       Nothing -> text "Duplicate user"
       Just k  -> text ("New user id " <> pack (show
k))
```

- insertUnique :: e -> m (Maybe (Key e))
  - Returns Nothing if uniqueness is violated
- insert :: e -> m (Key e) is an alternative

Query by identifier

```
get ("user" <//> var) $ \userId -> do
  user <- withDb $ get (UserKey $ SqlBackendKey userId)
  case user of
    Nothing -> setStatus status404
    Just u  -> text (userFirstname u)
```

- get :: Key e -> m (Maybe e)
  - Returns Nothing if not found
- Keys are strongly-typed, you have different ones for different entities in your database

Query by field data, 1/2

```
get ("user" <//> "by-name" <//> var) $ \name -> do
  users <- withDb $
    selectList ([UserFirstName ==. name]
                ||. [UserLastName ==. name])
               []
  -- ... Do things with users ...
```

- selectList obtains all records with match the query
- Conditions have the general form

  EntityField operator value

- AND is represented by list, OR by (||.)

Query by field data, 2/2

```
get ("user" <//> "by-name" <//> var) $ \name -> do
  users <- withDb $
    selectList ([UserFirstName ==. name]
                ||. [UserLastName ==. name])
               [Asc UserFirstName, LimitTo 10]
```

`selectList` takes a list of options to the query

- `Asc` and `Desc` order by a field
- `LimitTo` gives a maximum amount of records
- `OffsetBy` states a starting point for querying


Time for fun!

Two exercises more

3. Working with Persistent
    a. Adding fields
    b. Updating a record
    c. Deleting a record
4. One step further: the Esqueleto library
    a. Syntax similar to SQL
    b. Allows performant joins

Don't be shy: ask questions!

- We will share some solutions in the break

Questions + Answers 1

4

Forms and Validation

Yes, this data is really mandatory...

What if...

- ... the user does not exist?
- ... the task title is empty?

We need validation!

Doing it by hand

```
validate :: Thing -> Bool
-- or instead
validate :: Thing -> Maybe Thing
```

This is a great idea

Maybe is very **composable** :)

The Maybe Applicative

The value is OK

```
pure :: a -> Maybe a
```

Mapping a function over a Maybe value

```
(<$>) :: (a -> b) -> Maybe a -> Maybe b
```

Composition

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

The rule is: Nothing in, Nothing out

The Applicative pattern

Some basic data and validation

```
title :: Maybe String
user :: Maybe User  -- from database
nonEmpty :: Maybe String -> Maybe String
```

```
Task <$> nonEmpty title <*> user
    :: Maybe Task
```

POST parameters

```
post ("user" <//> "new") $ do
  firstname <- param "firstname"
  lastname  <- param "lastname"
  let user = User <$> notEmpty firstname
                  <*> notEmpty lastname
  case user of
    Just u -> do userId <- withDb $ insertUnique u
      case userId of
        Just uid -> text "Registration successful"
        Nothing  -> text "That user already exists"
    Nothing -> text "Wrong user data"
```

Maybe is not the best

1. No error messages
2. How should the information be displayed?
3. How to connect with parameters?

Solved by `digestive-functors`

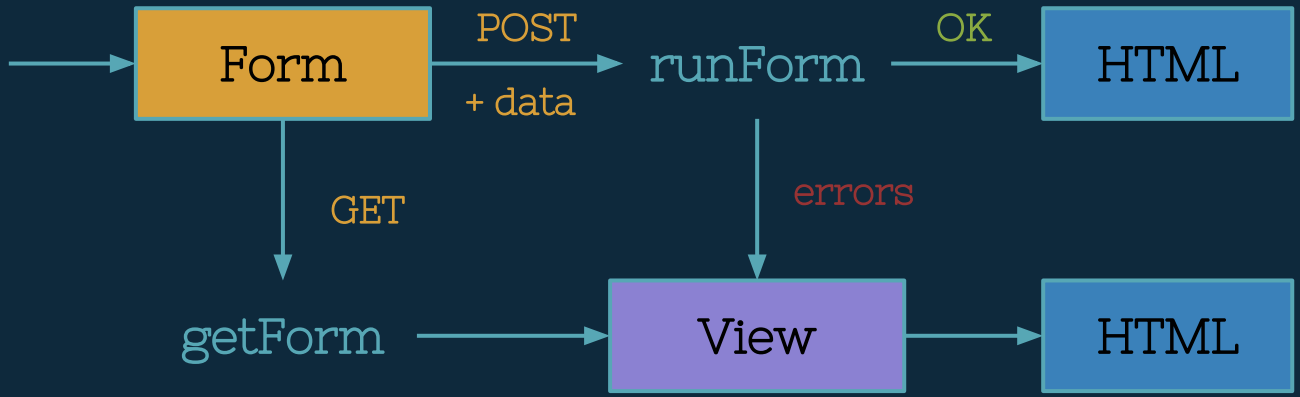
1. Define `Form`s with error information
2. Define `View`s + export to Blaze
3. Direct connection via `Spock-digestive`

Define a Form

```
userForm withDb =
  "u" .: checkM "User already exists"
          (\(User fn ln) -> isNothing <$>
              (withDb $ getBy $ UniqueName fn ln))
          (User <$> "firstname" .: check "Can't be empty"
                                         (not . null)
                                         (string Nothing)
                <*> "lastname"  .: check "Can't be empty"
                                         (not . null)
                                         (string Nothing))
```

- We refer to field by identifiers
- Pure validation uses `check`, monadic uses `checkM`

Form/View lifecycle



Form/View lifecycle, code

```
get ("user" <//> "new") $ do
  v <- getForm "u-data" (userForm withDb)
  html $ toStrict $ renderHtml $
    B.html $ B.body $ userView v

post ("user" <//> "new") $ do
  (v, newU) <- runForm "u-data" (userForm withDb)
  case newU of
    Just u  -> ...
    Nothing -> html $ toStrict $ renderHtml $
                 B.html $ B.body $ userView v
```

View to Blaze

```
userView :: View B.Html -> B.Html
userView view = do
  form view "/user/new" $ do
    label "u.firstname" view "First name: "
    inputText "u.firstname" view
    errorList "u.firstname" view
    B.br
    label "u.lastname" view "Last name: "
    inputText "u.lastname" view
    errorList "u.lastname" view
    B.br
    inputSubmit "Register!"
    errorList "u" view
```


Time for fun!

Just one this time

5. Forms for creating tasks
    a. Given an user identifier
    b. By choosing an user from a list

Extra. Use `Forms` to work with JSON data

Don't be shy: ask questions!

- ◇ We will share some solutions at the end

5

Middleware

Let other web modules help

Web Application Interface

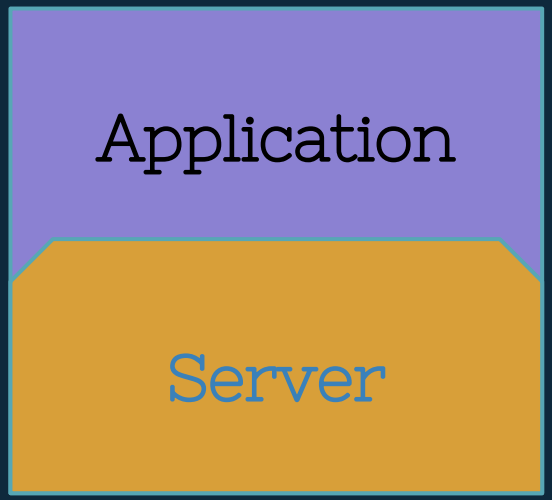
Spock, Scotty and Yesod build over a common API

- Web Application Interface, or WAI

The same application runs in several contexts

- As a stand-alone web server, via Warp
- Through FastCGI
- Linked with WebKit

WAI applications



Uses Spock, Scotty, Yesod

WAI

Warp, WebKit

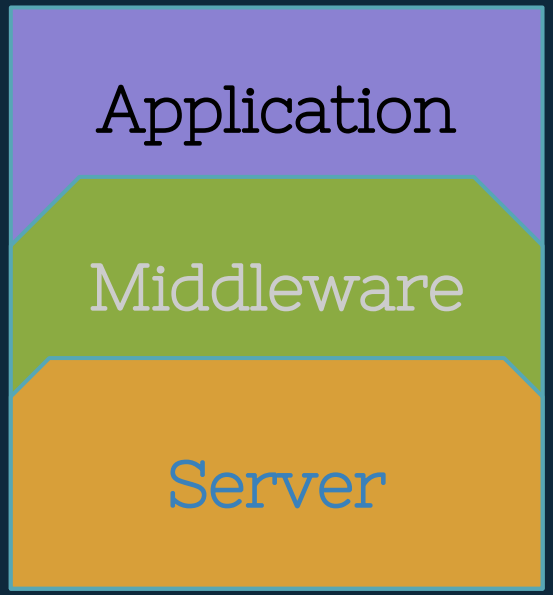
WAI applications

An `Application` is basically a function of type

```
Request -> IO Response
```

This is very composable!

WAI middleware



wai-middleware static

- Serve static files

wai-middleware-preprocessor

- Runs a command before serving

wai-extra

- GZip compression
- HTTP Basic Authentication
- and many more!

Middleware in Spock

```
main = runSpock 8080 $ spockT id $ do
  middleware $ staticPolicy (addBase "static")
  get ...
```

- `middleware` injects one in a Spock application
- `staticPolicy` comes from `wai-middleware-static`
  - Files should be looked for in the static folder
- If the middleware does not capture the request, then the next routes are tried out

6 Deployment

Our application finally leaves its nest :')

Questions + Answers 2

Thanks!

You can find me at:

- The rest of LambdaConf'15
- @trupill
- github.com/serras
- trupill@gmail.com

Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Photographs by Unsplash

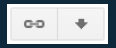
Presentation design

This presentation uses the following typographies and colors:

- Titles: Nixie One
- Body copy: Muli

You can download the fonts on this page:

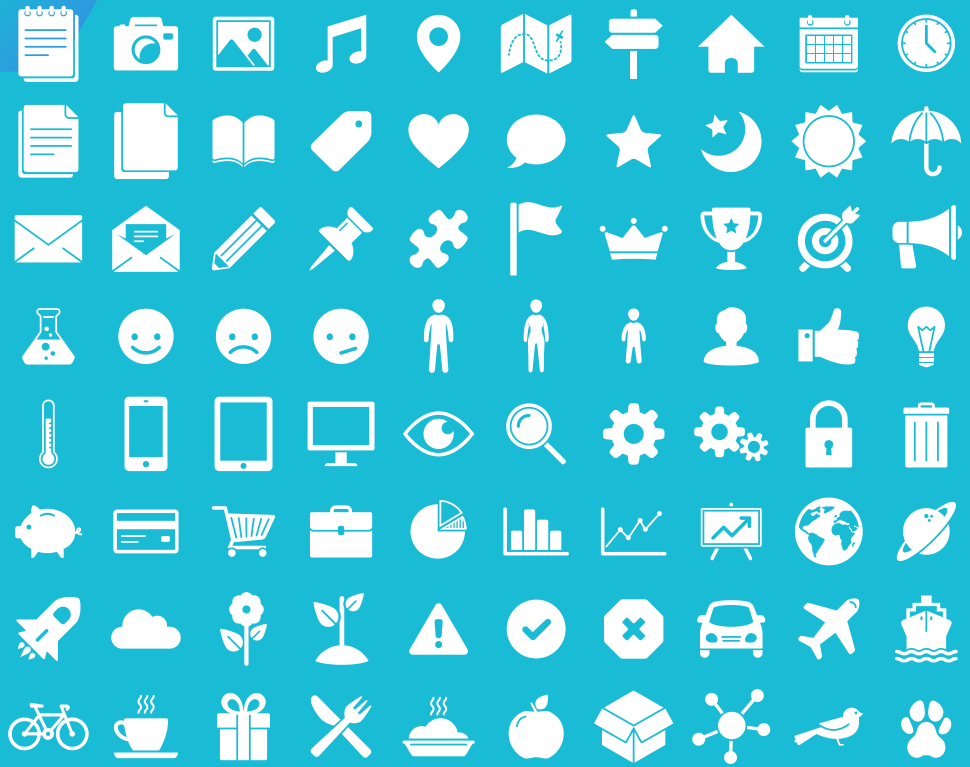
http://www.google.com/fonts#UsePlace:use/Collection:Nixie+One|Muli:300,400,300italic,400italic

Click on the “arrow button” that appears on the top right 

Aquamarina **#00e1c6** / Turquoise **#19bbd5** / Skyblue **#2c9dde** /

Light gray **#c6daec** / Dark blue **#0e293c**

You don’t need to keep this slide in your presentation. It’s only here to serve you as a design guide if you need to create new slides or download the fonts to edit the presentation in PowerPoint®



SlidesCarnival icons are editable shapes.

This means that you can:

- Resize them without losing quality.
- Change fill color and opacity.
- Change line color, width and style.

Isn't that nice? :)

Examples: