Web Services

Web Application Interface (WAI)

- Standard interface for any HTTP app
- Application: takes request, sends response
- Handler: runs application (for some definition of *run*)
- Middleware: modifies an application

Example handlers

- Warp (standalone web server)
- FastCGI
- Testing
- wai-handler-launch
- wai-handler-webkit (electron before it was cool?)

Example middleware

- Gzip compression
- Autohead (answer HEAD requests based on GET responses)
- Request logger

Building applications

- Do it by hand (we'll see that shortly)
- Use a web framework
  - Yesod
  - Servant
  - Scotty
  - Spock
  - Don't hate me for not listing others
- Not all frameworks in Haskell use WAI
  - Snap
  - Happstack
  - Probably some others

Hello WAI

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types

main :: IO ()
main = run 3000 $ \req send -> send $ responseBuilder
  status200
  (case lookup "marco" $ requestHeaders req of
     Nothing -> []
     Just val -> [("Polo", val)])
  "Hello WAI!"
```

```
$ curl -H Marco:foo -i http://localhost:3000
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Fri, 19 May 2017 11:23:15 GMT
Server: Warp/3.2.11.2
Polo: foo

Hello WAI!
```

Question Notice the lower case `marco` in the code, yet it matches. What black magic is this?

Application

```
type Application
   = Request
  -> (Response -> IO ResponseReceived)
  -> IO ResponseReceived

-- Basically a more complicated version of
type SimpleApp = Request -> IO Response
```

- CPS transformed to allow the application to acquire a scarce resource
- For example

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import qualified Data.ByteString.Lazy as BL
import System.IO

main :: IO ()
main = run 3000 $ \_req send -> withBinaryFile "Main.hs" ReadMode $ \h -> do
  lbs <- BL.hGetContents h -- evil lazy I/O! We'll do better soon
  send $ responseLBS
    status200
    [("Content-Type", "text/plain")]
    lbs
```

Exercise Write a function `unsimpleApp :: SimpleApp -> Application`.

Request

Lots of fields in `Request`, let's do some simple routing:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types

main :: IO ()
main = run 3000 $ \req send ->
  case pathInfo req of
    [] -> send $ responseBuilder
      status303
      [("Location", "/home")]
      "Redirecting"
    ["home"] -> send $ responseBuilder
      status200
      [("Content-Type", "text/plain")]
      "This is the home route"
```

```
$ curl -i http://localhost:3000
HTTP/1.1 303 See Other
Transfer-Encoding: chunked
Date: Fri, 19 May 2017 11:31:25 GMT
Server: Warp/3.2.11.2
Location: /home

$ curl http://localhost:3000/home
This is the home route

$ curl http://localhost:3000/foo
Something went wrong
```

Exercise Write an application that somehow responds to query string parameters.

Response

A few core smart constructors for `Response`:

```
responseFile
  :: Status
  -> ResponseHeaders
  -> FilePath
  -> Maybe FilePart
  -> Response`

responseBuilder
  :: Status
  -> ResponseHeaders
  -> Builder
  -> Response`

-- Just a wrapper for `responseBuilder`
responseLBS
  :: Status
  -> ResponseHeaders
  -> ByteString
  -> Response

responseStream
  :: Status
  -> ResponseHeaders
  -> StreamingBody
  -> Response

type StreamingBody
   = (Builder -> IO ()) -- send a chunk
  -> IO ()              -- flush the buffer
  -> IO ()

-- Useful for WebSockets in particular
responseRaw
  :: (   IO ByteString         -- receive from client
      -> (ByteString -> IO ()) -- send to client
      -> IO ())
  -> Response
  -> Response
```

Let's send a file the right way (`responseFile`):

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types

main :: IO ()
main = run 3000 $ \_req send -> send $ responseFile
    status200
    [("Content-Type", "text/plain")]
    "Main.hs"
    Nothing
```

And with streaming

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import qualified Data.ByteString as B
import Data.ByteString.Builder (byteString)
import System.IO
import Data.Function (fix)
import Control.Monad (unless)

main :: IO ()
main = run 3000 $ \_req send -> withBinaryFile "Main.hs" ReadMode $ \h ->
  send $ responseStream
    status200
    [("Content-Type", "text/plain")]
    $ \chunk _flush -> fix $ \loop -> do
      bs <- B.hGetSome h 4096
      unless (B.null bs) $ do
        chunk $ byteString bs
        loop
```

`responseFile` is better, it can use `sendfile` system call optimization.

Exercise Use streaming to send two files concatenated together. Now generalize that to an arbitrarily sized list.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import qualified Data.ByteString as B
import Data.ByteString.Builder (byteString)
import System.IO
import Data.Function (fix)
import Control.Monad (unless, forM_)

files :: [FilePath]
files = ["file1.txt", "file2.txt"]

withBinaryFiles :: [FilePath] -> IOMode -> ([Handle] -> IO a) -> IO a
withBinaryFiles fps mode inner =
  loop fps id
  where
    loop [] front = inner $ front []
    loop (x:xs) front =
      withBinaryFile x mode $ \h ->
      loop xs (front . (h:))

main :: IO ()
main = run 3000 $ \_req send -> withBinaryFiles files ReadMode $ \hs ->
  send $ responseStream
    status200
    [("Content-Type", "text/plain")]
    $ \chunk _flush -> forM_ hs $ \h -> fix $ \loop -> do
      bs <- B.hGetSome h 4096
      unless (B.null bs) $ do
        chunk $ byteString bs
        loop
```

Question This implementation uses too many of some resource, what is it? How can we work around it?

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import qualified Data.ByteString as B
import Data.ByteString.Builder (byteString)
import System.IO
import Data.Function (fix)
import Control.Monad (unless, forM_)
import Control.Monad.Trans.Resource
import Control.Monad.IO.Class
import UnliftIO.Exception (bracket)

files :: [FilePath]
files = ["file1.txt", "file2.txt"]

main :: IO ()
main = run 3000 $ \_req send ->
  bracket createInternalState closeInternalState $ \is ->
  send $ responseStream
  status200
  [("Content-Type", "text/plain")]
  $ \chunk _flush -> runInternalState (forM_ files $ \file -> do
      (releaseKey, h) <- allocate
        (openBinaryFile file ReadMode)
        hClose
      liftIO $ fix $ \loop -> do
        bs <- B.hGetSome h 4096
        unless (B.null bs) $ do
          chunk $ byteString bs
          loop
      release releaseKey) is
```

- That's a mouthful
- Yesod+Conduit makes this kind of thing much easier

aeson response body

Let's be a little more pragmatic...

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import Data.Aeson

main :: IO ()
main = run 3000 $ \_req send -> send $ responseBuilder
  status200
  [("Content-Type", "application/json")]
  $ fromEncoding $ toEncoding $ object
      [ "foo" .= (5 :: Int)
      , "bar" .= True
      ]
```

Exercise Send a YAML response instead. What do you think the performance difference will be here vs the code above?

aeson request body

- Again, a lot easier with Yesod, but let's make life difficult.
- Client code:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Aeson
import Network.HTTP.Simple

main :: IO ()
main = do
  let req = setRequestMethod "POST"
          $ setRequestBodyJSON (object ["hello" .= (1 :: Int)])
            "http://localhost:3000"
  res <- httpJSON req
  print (res :: Response Value)
```

And server code

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Conduit
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import Data.Aeson
import Data.Aeson.Parser (json)
import Data.Aeson.Types
import Data.Conduit
import Data.Conduit.Attoparsec (sinkParser)

newtype Body = Body Int

instance ToJSON Body where
  toJSON (Body i) = object ["hello" .= i]
instance FromJSON Body where
  parseJSON = withObject "Body" $ \o -> Body <$> o .: "hello"

main :: IO ()
main = run 3000 $ \req send -> do
  val <- runConduit
       $ sourceRequestBody req
      .| sinkParser json
  let Success (Body i) = fromJSON val
  send $ responseBuilder
    status200
    [("Content-Type", "application/json")]
    $ fromEncoding $ toEncoding $ Body $ i + 1
```

Exercise Do this with much better error handling

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Conduit
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import Data.Aeson
import Data.Aeson.Parser (json)
import Data.Aeson.Types
import Data.Conduit
import Data.Conduit.Attoparsec (sinkParser)
import UnliftIO.Exception
import qualified Data.ByteString.Lazy.Char8 as BL8

newtype Body = Body Int

instance ToJSON Body where
  toJSON (Body i) = object ["hello" .= i]
instance FromJSON Body where
  parseJSON = withObject "Body" $ \o -> Body <$> o .: "hello"

main :: IO ()
main = run 3000 $ \req send -> do
  eres <- tryAnyDeep $ do
    val <- runConduit
         $ sourceRequestBody req
        .| sinkParser json
    -- this is still bad! But tryAnyDeep hides it
    let Success (Body i) = fromJSON val
    return i
  send $ case eres of
    Left e -> responseLBS
      status500
      [("Content-Type", "text/plain")]
      $ BL8.pack $ "Exception occurred: " ++ show e
    Right i -> responseBuilder
      status200
      [("Content-Type", "application/json")]
      $ fromEncoding $ toEncoding $ Body $ i + 1
```

Throw in some middleware

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.Wai.Middleware.Autohead
import Network.Wai.Middleware.Gzip
import Network.Wai.Middleware.RequestLogger
import Network.HTTP.Types

main :: IO ()
main = run 3000
  $ logStdoutDev
  $ gzip def
  $ autohead
  $ \req send -> send $ responseBuilder
  status200
  (case lookup "marco" $ requestHeaders req of
     Nothing -> []
     Just val -> [("Polo", val)])
  "Hello WAI!"
```

Yesod?

- I'm giving a talk on Conduit and Yesod at the conference itself
- Feel free to attend... or if everyone wants I'll give a sneak peek now

Exercises

- Write a simple file server WAI app, that serves files in the current directory
    - *Security challenge time*
- Write an echo server, which sends back the same headers and body

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import qualified Data.ByteString as B
import Data.Function (fix)
import Control.Monad (unless)
import Data.ByteString.Builder (byteString)

main :: IO ()
main = run 3000 $ \req send -> send $ responseStream
  status200
  (requestHeaders req)
  $ \chunk _flush -> fix $ \loop -> do
    bs <- requestBody req
    unless (B.null bs) $ do
      chunk $ byteString bs
      loop
```

Problems:

- Some request headers should *not* be echoed (like `content-length`)
- Reading request body while writing response body concurrently may not be supported by all clients or WAI handlers, caveat empto

Additional material

- Web Application Interface
- Yesod Web Framework