APIs in Servant

WHO AM I?

Owen Harvey
Software Developer
CSIRO Data61: E&D

ASSUMED KNOWLEDGE

Servant requires a fair degree of comfort with some advanced Haskell concepts.

Assumptions going into this talk

- Typeclasses
- mtl/transformers
- handwaving extensions

Assume this throughout

```
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE FlexibleContexts      #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings     #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE TypeOperators         #-}
```

WHAT IS SERVANT

Type driven library for writing web clients and servers

WHY YOU SHOULD CARE

- Composable APIs
- Client generation from API type
- Swagger/OpenAPI output (with a util library)
- Enforced types from top to bottom

HOW SERVANT WORKS

Built around a set of combinators for building an API

Path, Query, Body, Method, Headers, etc.

WRITING A SERVANT API

```haskell
import Servant ((:>), Get, JSON)

-- create an endpoint at /hello/world
-- GET /hello/world returns a JSON String
type TestAPI = "hello" :> "world" :> Get '[JSON] String
```

:> joins segments into a route

'[JSON] defines the encodings we can use.

String says we will be returning a String to the user.

Multiple routes can be defined in a single type using :<|>

```
import Servant ((:>), Get, JSON, (:<|>))

type Foo = "foo" :> Get '[JSON] ()
type Bar = "bar" :> Get '[JSON] ()
type Baz = "baz" :> Post '[JSON] ()

-- These are identical
type FooBarAPI = Foo :<|> Bar :<|> Baz
type FooBarAPI =
  "foo" :> Get '[JSON] () :<|>
  "bar" :> Get '[JSON] () :<|>
  "baz" :> Post '[JSON] ()
```

Routes with a common root can be combined

```
type CommonPrefix = "prefix":> (Foo :<|> Bar :<|> Baz)
```

This is especially useful for handling authentication

ANATOMY OF A SERVANT SERVER

Servant servers are usually split into 3 broad sections

- API type
- Server function mapping
- Application types and code

THE API TYPE

What we looked at before

- Routes
- Path and Query Names
- Request Body
- Encodings
- Methods
- Headers

ROUTES

Defined using the combinator :>
Joined by :<|> to form a larger API

PATH AND QUERY NAMES

Capture and QueryParam
Capture "name" Type : Required value
QueryParam "name" Type : Optional Value

REQUEST BODY

ReqBody '[Encodings] Type
Takes a list of acceptable encodings and a type to decode to

ENCODINGS

Represent the Accept and Content-Type headers
Defined in lists of formats passed to other combinators
'[JSON, PlainText, OctetStream]

ENCODINGS

Heavily dependent on typeclasses to handle encodings
Most basic types are handled for you
Some encodings can be derived by the compiler or provided by hand

ENCODINGS

JSON

ToJSON a / FromJSON a

PlainText

String and Text

FormUrlEncoded

FromForm a / ToForm a

OctetStream

ByteString

ACCEPTABLE METHODS

Methods for a route mirror the HTTP verb
Get, Put, Delete, GetNoContent
Takes both a set of response encodings and a type for the content

REQUEST HEADERS

Header "name" Type
Captures an optional header from a request

RESPONSE HEADERS

Wrapper over the route response type
Headers '[Header "a" Type] ResponseType
Can defined multiple headers at once

SERVER FUNCTION MAPPING

Telling servant what to run
Uses the same combinators as the type definition

```haskell
type FooBarAPI =
 "foo" :> Capture "x" Bool :> Get '[JSON] String :<|>
 "bar" :> Get '[JSON] Integer :<|>
 "baz" :> ReqBody '[JSON] [String] :> Post '[JSON] [String]

foo :: MonadIO m => Bool -> m String
bar :: MonadIO m => m Integer
baz :: MonadIO m => [String] -> m [String]

fooBarServer :: MonadIO m => ServerT FooBarAPI m
fooBarServer = foo :<|> bar :<|> baz
```

Rules of Thumb

Each top level type joined by :<|> requires a function

Values are passed in the order they are defined

Must all return the same `m`

RUNNING A SERVER

servant-server is designed around the WAI specification
We'll be using the Warp server in this talk

```
serve :: HasServer api '[] =>
  Proxy api -> ServerT api Handler -> Application
```

turns an API and a server into a runnable Application

Handler is a monad that gives servant the functionality it needs to run requests

Sometimes Handler is undesirable

- IO free application code
- Value passing with ReaderT

```
hoistServer :: HasServer api '[] =>
  Proxy api -> (forall x. m x -> n x) ->
  ServerT api m -> ServerT api n
```

usually converting

```
ServerT api m -> ServerT api Handler
```

```haskell
type FooBarAPI =
  "foo" :> Capture "x" Bool :> Get '[JSON] String :<|>
  "bar" :> Get '[JSON] Integer :<|>
  "baz" :> ReqBody '[JSON] [String] :> Post '[JSON] [String]

fooBarAPI    :: Proxy FooBarAPI
foo          :: MonadIO m => Bool -> m String
bar          :: MonadIO m => m Integer
baz          :: MonadIO m => [String] -> m [String]
fooBarServer :: MonadIO m => ServerT FooBarAPI m

app = serve fooBarAPI fooBarServer

main = Network.Wai.Handler.Warp.run 8080 app
```

SERVANT CLIENTS

Clients are split into two parts

- API type
- ClientM functions

API clients are built around ClientM and runClientM
ClientM represents a request and all that entails
runClientM executes a request and returns results

Like servers, functions are extracted by patterns

```
type FooBarAPI =
  "foo" :> Capture "x" Bool :> Get '[JSON] String :<|>
  "bar" :> Get '[JSON] Integer :<|>
  "baz" :> ReqBody '[JSON] [String] :> Post '[JSON] [String]

data ApiCalls = ApiCalls
  { foo :: Bool -> ClientM String
  , bar :: ClientM Integer
  , baz :: [String] -> ClientM [String]
  }

apiCalls = ApiCalls foo bar
  where
    foo :<|> bar :<|> baz = client (Proxy :: Proxy FooBarAPI)
```

runClientM needs a ClientEnv

ClientEnv contains basic request data

- Host
- Cookies
- TLS settings
- Timeouts

```haskell
clientEnv :: IO ClientEnv
clientEnv = do
  manager <- newManager defaultManagerSettings
  let baseUrl = BaseUrl Http "localhost" 8080 ""
  pure $ mkClientEnv manager baseUrl
```

Putting the ClientEnv into a ReaderT is very helpful

```haskell
type AppM m a = ReaderT ClientEnv (ExceptT ClientError m) a
getFoo :: MonadIO m => AppM m String
getFoo = do
  env <- ask
  res <- liftIO $ runClientM (foo apiCalls True) env
  either throwError pure res
getBar          :: MonadIO m => AppM m Integer
postBaz         :: MonadIO m => AppM m [String]
someAppFunction :: MonadIO m => AppM m (String, ...)

main = do
  env <- clientEnv
  res <- runExceptT $ runReaderT someAppFunction clientEnv
  either throwError pure res
```

ERROR HANDLING

MonadError and ExceptT are your friends
Servers use ServerError
Clients use ClientError
Keeps clients and servers seperate

Server error handling

Application error types can be mapped using withExceptT

```haskell
data AppError
  = ErrorA ByteString
  | ErrorB Int

type AppM a = ExceptT AppError IO a

toServantError :: AppError -> ServerError
toServantError e = case e of
  ErrorA s -> err400 { errBody = s }
  ErrorB i -> err500 { errBody = pack $ show i }

toHandler :: AppM a -> Handler a
toHandler = Handler . withExceptT toServantError
```

Client error handling

```
toAppError :: ClientError -> AppError
toAppError (ConnectionError e) = AppException e
...

type AppM a = ExceptT AppError IO a

mapError :: ClientM a -> AppM a
mapError client = do
  e <- runClientM client env
  either
    (throwError . toAppError)
    pure
    e
```

SWAGGER

Automatic Swagger/OpenAPI generation
Completely type driven

```haskell
module Main where

import Data.Aeson
import Data.ByteString.Lazy.Char8
import Data.Proxy
import Servant
import Servant.Swagger

type FooBarAPI =
  "foo" :> Capture "x" Bool :> Get '[JSON] String :<|>
  "bar" :> Get '[JSON] Integer :<|>
  "baz" :> ReqBody '[JSON] [String] :> Post '[JSON] [String]

main = putStrLn . encode $ toSwagger @(FooBarAPI) Proxy
```

OTHER COOL THINGS

JS CLIENT GENERATION

```
apiJS :: Text
apiJS = jsForAPI api vanillaJS
```

apiJS will contain the generated code

```
var getPoint = function(onSuccess, onError)
{
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/point', true);
  ...
```

This works with jQuery, vanillaJS (XMLHttpRequest), Axios, and Angular

LINKS

Servant

docs.servant.dev

Hackage

hackage.haskell.org/package/servant

hackage.haskell.org/package/servant-server

hackage.haskell.org/package/servant-client

hackage.haskell.org/package/servant-swagger

hackage.haskell.org/package/servant-js

Slides

github.com/lepsa/servant-talk