Servant - Handout

A quick run-through of Servant and why

Cyrill Brunner

3rd October 2022

- Servant is a type-level DSL

- Servant is a type-level DSL
- Written in Haskell

- Servant is a type-level DSL
- Written in Haskell
- Encoding as much as possible on the type-level

- Servant is a type-level DSL
- Written in Haskell
- Encoding as much as possible on the type-level
- Enables automatic derivation of:

- Servant is a type-level DSL
- Written in Haskell
- Encoding as much as possible on the type-level
- Enables automatic derivation of:
- Server-side function signatures

- Servant is a type-level DSL
- Written in Haskell
- Encoding as much as possible on the type-level
- Enables automatic derivation of:
- Server-side function signatures
- Client-side accessor functions (in Haskell, but also JS etc.)

- Servant is a type-level DSL
- Written in Haskell
- Encoding as much as possible on the type-level
- Enables automatic derivation of:
- Server-side function signatures
- Client-side accessor functions (in Haskell, but also JS etc.)
- Documentation

- Servant is a type-level DSL
- Written in Haskell
- Encoding as much as possible on the type-level
- Enables automatic derivation of:
- Server-side function signatures
- Client-side accessor functions (in Haskell, but also JS etc.)
- Documentation
- Swagger/OpenAPI Specification

- Written to be extensible and pluggable

- Written to be extensible and pluggable
- Ad-hoc extension/wrapping of existing features supported!

- Written to be extensible and pluggable
- Ad-hoc extension/wrapping of existing features supported!
- Easy introduction of new API parameter/return types – e.g. just auto-derive ToJSON/FromJSON instances!

- Written to be extensible and pluggable
- Ad-hoc extension/wrapping of existing features supported!
- Easy introduction of new API parameter/return types – e.g. just auto-derive ToJSON/FromJSON instances!
- Require authentication for an entire section of your server? Just pluck `Auth '[JWT, Cookie] LoginToken` before it

- Written to be extensible and pluggable
- Ad-hoc extension/wrapping of existing features supported!
- Easy introduction of new API parameter/return types – e.g. just auto-derive ToJSON/FromJSON instances!
- Require authentication for an entire section of your server? Just pluck `Auth '[JWT, Cookie] LoginToken` before it
- `LoginToken` not enough? Easy, write a wrapper that queries your database and verifies the token automatically!

- Written to be extensible and pluggable
- Ad-hoc extension/wrapping of existing features supported!
- Easy introduction of new API parameter/return types – e.g. just auto-derive ToJSON/FromJSON instances!
- Require authentication for an entire section of your server? Just pluck `Auth '[JWT, Cookie] LoginToken` before it
- `LoginToken` not enough? Easy, write a wrapper that queries your database and verifies the token automatically!
- And so on...

Check https://docs.servant.dev/ – The tutorial introduces core concepts, the Cookbook shows popular libraries, patterns etc. contributed by the community.

Haskell I

- Pure – No mutation in the language

Haskell I

- Pure – No mutation in the language
- Lazy – Evaluation only when necessary

Haskell I

- Pure – No mutation in the language
- Lazy – Evaluation only when necessary
- Functional – Write your program by composing functions

Haskell I

- Pure – No mutation in the language
- Lazy – Evaluation only when necessary
- Functional – Write your program by composing functions
- How to **do** Anything $\rightarrow$ Effects!

Haskell II

- `IO` is the catch-all effect for Input/Output

Haskell II

- `IO` is the catch-all effect for Input/Output
- `IORef`, `MVar` – mutable "cells" for state

Haskell II

- `IO` is the catch-all effect for Input/Output
- `IORef`, `MVar` – mutable "cells" for state
- `Reader` – Immutable, read-only environment (usable for config)

Haskell II

- `IO` is the catch-all effect for Input/Output
- `IORef`, `MVar` – mutable "cells" for state
- `Reader` – Immutable, read-only environment (usable for config)
- `State` – Mutable environment by passing it along implicitly

Haskell II

- `IO` is the catch-all effect for Input/Output
- `IORef`, `MVar` – mutable "cells" for state
- `Reader` – Immutable, read-only environment (usable for config)
- `State` – Mutable environment by passing it along implicitly
- `Except` – Exceptions and catching them

Haskell II

- `IO` is the catch-all effect for Input/Output
- `IORef`, `MVar` – mutable "cells" for state
- `Reader` – Immutable, read-only environment (usable for config)
- `State` – Mutable environment by passing it along implicitly
- `Except` – Exceptions and catching them
- `List` – Usable for non-determinism

Haskell II

- `IO` is the catch-all effect for Input/Output
- `IORef`, `MVar` – mutable "cells" for state
- `Reader` – Immutable, read-only environment (usable for config)
- `State` – Mutable environment by passing it along implicitly
- `Except` – Exceptions and catching them
- `List` – Usable for non-determinism
- `Writer` – Collect output (generally not for Logging)

Haskell III

- `RWS` – ReaderWriterState

Haskell III

- `RWS` – ReaderWriterState
- `Accum` – Append-only version of State/Writer with ability to see previous output

Haskell III

- `RWS` – ReaderWriterState
- `Accum` – Append-only version of State/Writer with ability to see previous output
- `Cont` – Continuations, don't @me

Haskell III

- `RWS` – ReaderWriterState
- `Accum` – Append-only version of State/Writer with ability to see previous output
- `Cont` – Continuations, don't @me
- `Select` – Selection, for search algorithms, first time I've heard it pleasedon'taskmeIdon'tknow

Haskell III

- `RWS` – ReaderWriterState
- `Accum` – Append-only version of State/Writer with ability to see previous output
- `Cont` – Continuations, don't @me
- `Select` – Selection, for search algorithms, first time I've heard it pleasedon'taskmeIdon'tknow
- `SMT` – Software Transactional Memory $\rightarrow$ deadlock-free, race-free atomic concurrent state

Haskell III

- `RWS` – ReaderWriterState
- `Accum` – Append-only version of State/Writer with ability to see previous output
- `Cont` – Continuations, don't @me
- `Select` – Selection, for search algorithms, first time I've heard it pleasedon'taskmeIdon'tknow
- `SMT` – Software Transactional Memory $\rightarrow$ deadlock-free, race-free atomic concurrent state
- `Logging` – Well, Logging

Haskell III

- `RWS` – ReaderWriterState
- `Accum` – Append-only version of State/Writer with ability to see previous output
- `Cont` – Continuations, don't @me
- `Select` – Selection, for search algorithms, first time I've heard it pleasedon'taskmeIdon'tknow
- `SMT` – Software Transactional Memory $\rightarrow$ deadlock-free, race-free atomic concurrent state
- `Logging` – Well, Logging
- *Roll your own*

Example

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE BlockArguments #-}
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE DerivingStrategies #-}
module Main where

import Control.Monad.IO.Class   ( liftIO )
import Data.Aeson               ( ToJSON, FromJSON )
import Data.IORef               ( IORef, newIORef
                                , readIORef, writeIORef
                                , modifyIORef
                                )
import Data.Proxy               ( Proxy )
import GHC.Generics             ( Generic )
import Network.Wai.Handler.Warp ( run )
import Servant.API
import Servant.Server
```

Example

```
data Counter = Counter { count :: Int, lastModification :: Int }
  deriving stock (Eq, Show, Generic)
  deriving anyclass (FromJSON, ToJSON)

type API =
  -- GET /count, returns { "count": 20, "lastModification": 0 }
  "count" :> Get '[JSON] Counter
  -- POST /add with Int as JSON, returns new Counter
  :<|> "add" :> ReqBody '[JSON] Int :> Post '[JSON] Counter
  -- POST /subtract with Int as JSON, returns nothing
  :<|> "subtract" :> ReqBody '[JSON] Int :> NoContentVerb 'POST
```

Example

```
-- Server ("count" :> Get '[JSON] Counter)
getCount :: IORef Counter -> Handler Counter
getCount ref = liftIO $ readIORef ref

-- Server ("add" :> ReqBody '[JSON] Int :> Post '[JSON] Counter)
addCount :: IORef Counter -> Int -> Handler Counter
addCount ref i = do
  Counter {..} <- liftIO $ readIORef ref
  let counter = Counter { count = count + i
                        , lastModification = i
                        }
  liftIO $ writeIORef ref counter
  return counter
```

Example

```
-- Server ("subtract" :> ReqBody '[JSON] Int :> NoContentVerb 'POST)
subCount :: IORef Counter -> Int -> Handler NoContent
subCount ref i = do
  liftIO
    $ modifyIORef ref
    $ \Counter {..} -> Counter { count = count-i
                               , lastModification = (-i)
                               }
  return NoContent

server :: IORef Counter -> Server API
server ref = getCount ref :<|> addCount ref :<|> subCount ref
```

Example

```
main :: IO ()
main = do
  ref <- newIORef $ Counter 0 0
  run 8080 $ serve (Proxy :: Proxy API) $ server ref
```