Servant vs. Mu

A Type-Level Battle

Alejandro Serrano @ BOB 2021

🐦 @trupill - 👨‍💻 47 Degrees (Academy)

Servant and Mu

Sets of libraries to develop services in Haskell

Servant and Mu

Sets of libraries to develop services in Haskell

🤖 Servant - `servant.dev`

- Focus on web services: REST, OpenAPI
- Both client and server

Servant and Mu

Sets of libraries to develop services in Haskell

🤖 Servant - `servant.dev`

- Focus on web services: REST, OpenAPI
- Both client and server

Mu - `higherkindness.io/mu`

- Microservices, multi-protocol: gRPC, GraphQL
- Preceded by a Scala sibling

Why compare them?

Focus on a similar tech space

- Choices for developing microservices
- Even more when the protocol is still in flux

Why compare them?

Focus on a similar tech space

- Choices for developing microservices
- Even more when the protocol is still in flux

Both use *type-level techniques*

Using lots of GHC extensions, and some more

- Interesting exploration of the design space
- How much of this is exposed to the user?

👁‍🗨 Disclaimer

I am one of the core developers of Mu

🤖 Servant - `servant.dev`

Your API as a type

```haskell
type UserAPI
  =     "users" :> Get '[JSON] [User]
  :<|> "user"  :> Capture "user_id" Int
               :> Get '[JSON] User
```

defines your API as two routes

```
GET /users
GET /user/:user_id
```

Serving the API

```haskell
type UserAPI
  =     "users" :> Get '[JSON] [User]
  :<|> "user"  :> Capture "user_id" Int
               :> Get '[JSON] User
```

You provide a `Handler` per route

```haskell
server :: Server UserAPI
server = users :<|> user
  where users :: Handler [User]
        users = ...
        user :: Int → Handler User
        user user_id = ...
```

Serving the API

```haskell
server :: Server UserAPI
server = users :<|> user
  where users :: Handler [User]
        users = ...
        user :: Int -> Handler User
        user user_id = ...
```

`Handler` extends `IO` with the ability to stop

```haskell
type Handler = ExceptT ServerError IO
```

Serving the API

```haskell
server :: Server UserAPI
server = users :<|> user
  where users :: Handler [User]
        users = ...
        user :: Int -> Handler User
        user user_id = ...
```

Serialization is handled by the library

- From string to `Int` in a URL part
- Using Aeson's `ToJSON` for `User`

Querying the API

```haskell
type UserAPI
  =    "users" :> Get '[JSON] [User]
  :<|> "user"  :> Capture "user_id" Int
               :> Get '[JSON] User
```

Client code is automatically derived

```haskell
users :: ClientM [User]
user  :: Int → ClientM User

users :<|> user = client (Proxy @UserAPI)
```

Mu - `higherkindness.io/mu`

gRPC service definition

This is `helloworld.proto`,
using Protocol Buffers syntax

```
package helloworld;

message HelloRequest { string name = 1; }
message HelloReply { string message = 1; }

service Greeter {
  rpc SayHello (HelloRequest)
    returns (HelloReply) {}
  rpc SayManyHellos (stream HelloRequest)
    returns (stream HelloReply) {}
}
```

Import the service definition

```haskell
{-# language TemplateHaskell #-}

grpc "Schema" (const "Service") "helloworld.proto"
```

Import the service definition

```haskell
{-# language TemplateHaskell #-}

grpc "Schema" (const "Service") "helloworld.proto"
```

Messages may be mapped to Haskell types

```haskell
data HelloRequestMessage = Req { name :: T.Text }
  deriving (Eq, Show, Generic
           , ToSchema   Schema "HelloRequest"
           , FromSchema Schema "HelloRequest")

data HelloReplyMessage = Reply { message :: T.Text }
  deriving (Eq, Show, Generic
           , ToSchema   Schema "HelloReply"
           , FromSchema Schema "HelloReply")
```

Define the server

```haskell
server = singleService
  ( method @"SayHello" sayHello
  , method @"SayManyHellos" sayManyHellos )
  where
    sayHello
      :: HelloRequest -> ServerErrorIO HelloResponse
    sayHello (HelloRequest nm)
      = pure $ HelloResponse ("hi, " <> nm)

    sayManyHellos
      :: ConduitT () HelloRequest m ()
      -> ConduitT HelloResponse Void m ()
      -> ServerErrorIO ()
    sayManyHellos = ...
```

One server, many protocols

The same server can be exposed through different interfaces, *if compatible*

```
runConcurrently $ (\_ _ _ → ())
  <$> c 50051 (gRpcApp msgProtoBuf server)
  <*> c 50052 (gRpcApp msgAvro     server)
  <*> c 50053 (graphQLApp server (Proxy @...))
  where c port f = Concurrently (run port f)
```

⚔️ Let the battle begin!

Focus #1: server definition

Focus #2: serialization

Focus #3: API representation

⚔️ Focus #1: server definition

🤖 ⚗ The "handler monad"

Both libraries use *simple functions*

- arguments represent the inputs

```
user :: Int → Handler User
user user_id = ...
```

- execute inside a similar monad

```
type Handler = ExceptT ServerError IO
```

🤖 The "handler monad"

```haskell
type Handler = ExceptT ServerError IO
```

From the Servant docs:

> [...] it is the simplest monad that:
>
> - lets us both return a successful result (using `return`) or “fail” with an error (using `throwError`);
> - lets us perform IO, which is absolutely vital since most webservices exist as interfaces to databases that we interact with in `IO`.

🤖 Escaping out of the monad

Using a natural transformation

`forall x. f x → g x`

```
hoistServer
  :: HasServer api '[] ⇒ Proxy api
  → (forall x. m x → n x)
  → ServerT api m → ServerT api n
```

```
runGRpcAppTrans
  :: ( ... ) ⇒ Proxy protocol → Port
  → (forall a. m a → ServerErrorIO a)
  → ServerT chn () pkg m handlers → IO ()
```

🤖 Handlers in order

Handlers in Servant must appear in the same order as they are defined

```haskell
type UserAPI
  =    "users" :> Get '[JSON] [User]
  :<|> "user"  :> Capture "user_id" Int
               :> Get '[JSON] User

server :: Server UserAPI
server = users :<|> user
```

Handlers out of order

Use special functions and type-level strings to figure everything out

```
server = singleService
  ( method @"SayHello" sayHello
  , method @"SayManyHellos" sayManyHellos )
```

Handlers out of order

Use special functions and type-level strings to figure everything out

```
server = singleService
  ( method @"SayHello" sayHello
  , method @"SayManyHellos" sayManyHellos )
```

Internally, this is translated to Servant-style

- Do the "matching" only *once* at compile-time

🤖 ⚔️ Handler order: comparison

🤖 ⚔️ 🟪 Handler order: comparison

🤖 Declaration order

👍 Compile time is decreased

👎 Change in the API $\Rightarrow$ less than trivial error

🤖 ⚔️ Handler order: comparison

🤖 Declaration order

👍 Compile time is decreased

👎 Change in the API ⇒ less than trivial error

Out of order, tagged with names

👍 Readability of the server

🤏 Duplication of names in schema and code

👎 Misuse of combinators ⇒ terrible error

⚔️ Focus #2: serialization

🤖 Serialization

User perspective

(Re)use different classes per content type

- `From/ToHttpApiData` for text in URLs
- `From/ToJSON` (from Aeson) for JSON

Linking them together

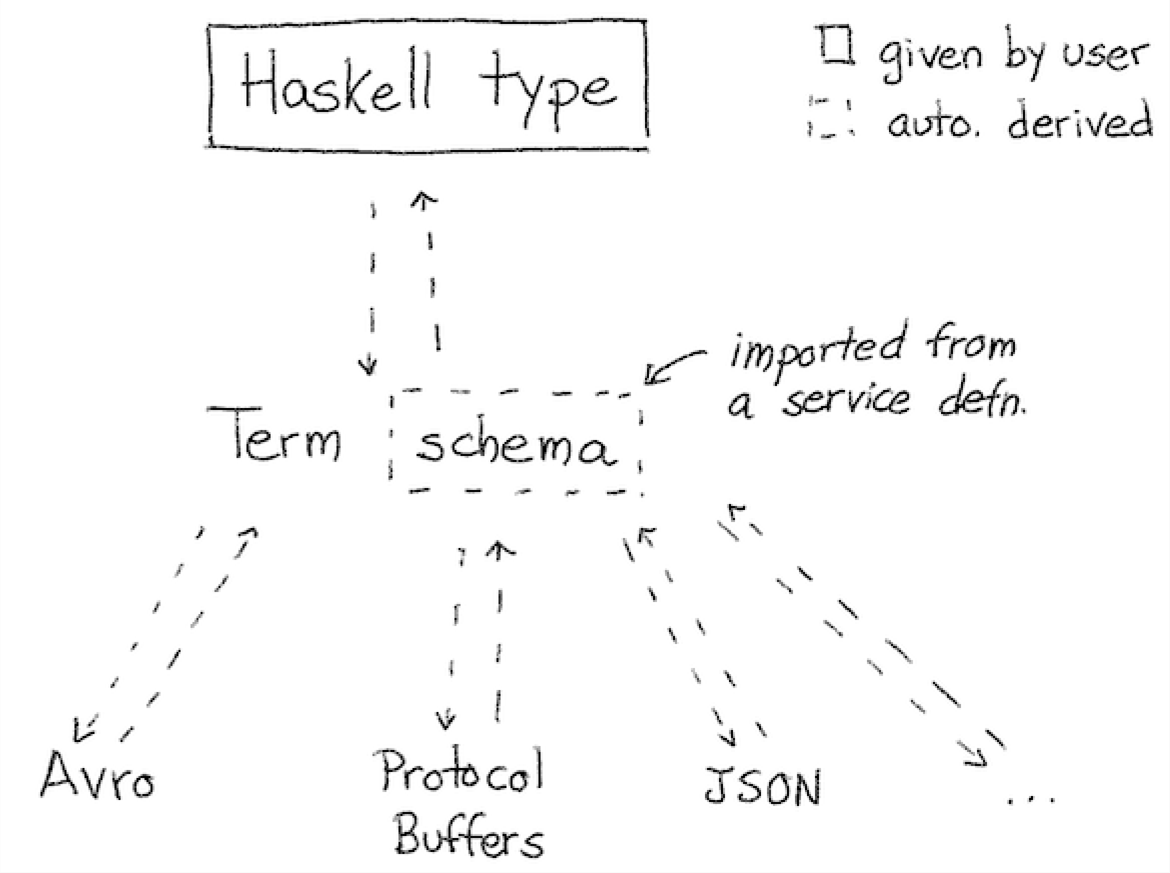
Via the `MimeRender` class and type-level names

```
class MimeRender ctype a where
    mimeRender
      :: Proxy ctype → a → ByteString

data JSON  -- empty data type
instance ToJSON a ⇒ MimeRender JSON a ...
```

Serialization

Use of an intermediate `Term` data type



Serialization

Conversion is automatized using `GHC.Generics`

```
data SchemaType = ...
  deriving (Eq, Show, Generic
           , ToSchema   Schema "SchemaType"
           , FromSchema Schema "SchemaType")
```

Serialization

Conversion is automatized using `GHC.Generics`

```
data SchemaType = ...
  deriving (Eq, Show, Generic
           , ToSchema   Schema "SchemaType"
           , FromSchema Schema "SchemaType")
```

🙈 `mu-schema` is *yet another* generics library

🤖 ⚔️ Serialization: comparison

- 🤖 Need to manually derive each content type
- Only a single `From/ToSchema` is required

🤖 ⚔️ 📦 Serialization: comparison

🤖 Need to manually derive each content type

📦 Only a single `From/ToSchema` is required

📦 Does one size fit all?

👍 No (user) code to move to another protocol

👎 Lack of configurability (e.g., JSON keys)

👎 The `Term` data type is a "Frankenstein" (some protocols support unions, others not...)

🤖 HTML as the content type

Servant has integrations to produce HTML,
a common output of a web service

- `servant-lucid`
- `servant-blaze`

Mu only focuses on data-returning services

⚔️ Focus #3: API representation

🤖 Type as an API

The programmer writes the type *manually*

```
type UserAPI
  =     "users" :> Get '[JSON] [User]
  :<|>  "user"  :> Capture "user_id" Int
                :> Get '[JSON] User
```

🤖 Type as an API

The programmer writes the type *manually*

```haskell
type UserAPI
  =     "users" :> Get '[JSON] [User]
  :<|>  "user"  :> Capture "user_id" Int
                :> Get '[JSON] User
```

👍 Easy to understand

👎 Difficult to share, you need packages such as `servant-js` / `elm` to create clients

Import the service definition

```haskell
{-# language TemplateHaskell #-}

grpc "Schema" (const "Service") "helloworld.proto"
```

Import the service definition

```haskell
{-# language TemplateHaskell #-}

grpc "Schema" (const "Service") "helloworld.proto"
```

which results in schema definitions...

```haskell
type QuickstartSchema
  = '[ 'DRecord "HelloRequest"
                '[ 'FieldDef "name"    ('TPrimitive T.Text) ]
     , 'DRecord "HelloResponse"
                '[ 'FieldDef "message" ('TPrimitive T.Text) ] ]

type instance AnnotatedSchema ProtoBufAnnotation QuickstartSchema
  = '[ 'AnnField "HelloRequest"  "name"    ('ProtoBufId 1)
     , 'AnnField "HelloResponse" "message" ('ProtoBufId 1) ]
```

Import the service definition

```haskell
{-# language TemplateHaskell #-}

grpc "Schema" (const "Service") "helloworld.proto"
```

… and service definitions

```haskell
type QuickstartService
  = 'Service "Greeter"
      '[ 'Method "SayHello" '[]
        '[ 'ArgSingle 'Nothing '[]
             ('FromSchema QuickstartSchema "HelloRequest") ]
        ('RetSingle ('FromSchema QuickstartSchema "HelloResponse")) ]
```

Way more complex than Servant!

Schema-first

👍 Better for sharing across teams

- Well-established gRPC and GraphQL clients

👍 The schema/service definition API is hidden

- We have changed it in every major release without changes to the examples

👎 Inspectability and error reporting

🤖 ⚔️ Single vs. multi-protocol

- 🤖 Focus on HTTP-oriented protocols
- The same code for different protocols

🤖 ⚔️ Single vs. multi-protocol

🤖 Focus on HTTP-oriented protocols

The same code for different protocols

Does one size fit all?

👍 No (user) code to move to another protocol

👎 Hard to diagnose when this is not possible

- RPC protocols differ way more than serialization formats

Bridging both worlds

`mu-servant-server`

🤖 🤝 mu-servant-server

Expose a Mu server as a Servant one

Use the annotations machinery in Mu
to "fill the gaps" about routes

```
type instance AnnotatedPackage ServantRoute Service
  = '[ 'AnnService "Greeter"
          ('ServantTopLevelRoute '["greet"])
     , 'AnnMethod "Greeter" "SayHello"
          ('ServantRoute '["say", "hello"] 'POST 200)
     ]
```

🤖 🤝 mu-servant-server

Expose a Mu server as a Servant one

A type family creates a Servant API type looking up those annotations

```
PackageAPI Service handlers
```

The instances define what can be translated

- Right now, only *one* argument in the *body*

🤖 🤝 mu-servant-server

Expose a Mu server as a Servant one

Help us bridging both worlds!

- Import OpenAPI definitions in Mu
- Support more complex Servant routes

🤩 It's been a pleasure

Enjoy the rest of BOB 2021!