Your Web Service as a Type

(Typing REST APIs with Servant)

I am

- Christian Marie (pingu on IRC).

I am

- Christian Marie (pingu on IRC).
- Employed by Anchor Systems, a managed cloud hosting provider.

I am

- Christian Marie (pingu on IRC).
- Employed by Anchor Systems, a managed cloud hosting provider.
- Recently, a developer of Servant.

Every time you try to webservice

Your REST API decides to...

- Break in subtle ways

Every time you try to webservice

Your REST API decides to...

- Break in subtle ways
- Mix boilerplate with important business logic

Every time you try to webservice

Your REST API decides to. . .

- Break in subtle ways
- Mix boilerplate with important business logic
- Become infinitely complex

Every time you try to webservice

Your REST API decides to...

- Break in subtle ways
- Mix boilerplate with important business logic
- Become infinitely complex
- Become partially and/or inconsistently documented

Servant - type combinators for webservice APIs

- A collection of libraries built around the concept of typed APIs.

Servant - type combinators for webservice APIs

- A collection of libraries built around the concept of typed APIs.
- Six developers

Servant - type combinators for webservice APIs

- A collection of libraries built around the concept of typed APIs.
- Six developers
- At least two commercial users (Zalora, Anchor)

Servant - type combinators for webservice APIs

- A collection of libraries built around the concept of typed APIs.
- Six developers
- At least two commercial users (Zalora, Anchor)
- About to hit a 0.3 release with some major improvements.

Your API wants types

REST problems

- Break in subtle ways

Types can fix that!

Your API wants types

REST problems

- Break in subtle ways
- Boilerplate gets mixed with important business logic

Types can fix that!

Your API wants types

REST problems

- Break in subtle ways
- Boilerplate gets mixed with important business logic
- Complexity becomes nightmare to maintain

Types can fix that!

Your API wants types

REST problems

- Break in subtle ways
- Boilerplate gets mixed with important business logic
- Complexity becomes nightmare to maintain
- Becomes partially and/or inconsistently documented

Types can fix that!

Your API wants types

REST problems

- Break in subtle ways
- Boilerplate gets mixed with important business logic
- Complexity becomes nightmare to maintain
- Becomes partially and/or inconsistently documented

Types can fix that!

- Explode at compile time

Your API wants types

REST problems

- Break in subtle ways
- Boilerplate gets mixed with important business logic
- Complexity becomes nightmare to maintain
- Becomes partially and/or inconsistently documented

Types can fix that!

- Explode at compile time
- Make generic programming an option

Your API wants types

REST problems

- Break in subtle ways
- Boilerplate gets mixed with important business logic
- Complexity becomes nightmare to maintain
- Becomes partially and/or inconsistently documented

Types can fix that!

- Explode at compile time
- Make generic programming an option
- Provide a framework for complexity

Your API wants types

REST problems

- Break in subtle ways
- Boilerplate gets mixed with important business logic
- Complexity becomes nightmare to maintain
- Becomes partially and/or inconsistently documented

Types can fix that!

- Explode at compile time
- Make generic programming an option
- Provide a framework for complexity
- Provide documentation, with 100% coverage

How do you even API as type?

Figure 1: It is okay. I might know how to do this.

Thought experiment: Your API as a tree.

- Leaves are endpoints (GET, POST, etc)

Thought experiment: Your API as a tree.

- Leaves are endpoints (GET, POST, etc)
- Internal nodes “modify” the endpoint that they lead to.

APIs have shapes

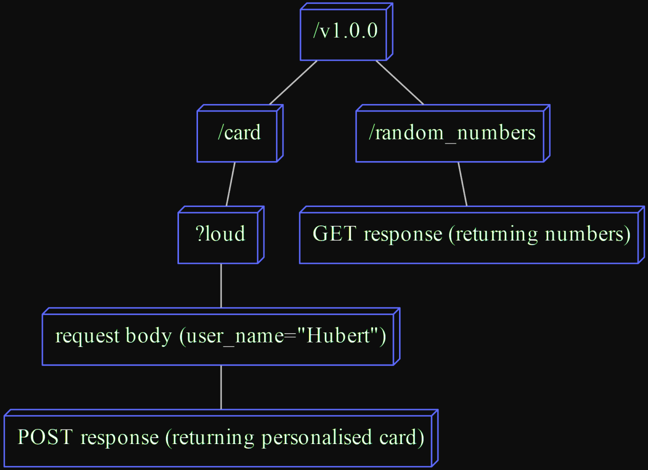

/v1.0.0
/card
/random_numbers
?loud
GET response (returning numbers)
request body (user_name="Hubert")
POST response (returning personalised card)

Figure 2: Your API as a tree

Types have shapes (type operators)

head :> tail

- For joining nodes

```haskell
data (path :: k) :> a
infixr 9 :>
```

branch1 :<|> branch2

```haskell
data a :<|> b = a :<|> b
infixr 8 :<|>
```

Types have shapes (type operators)

head :> tail

- For joining nodes
- Constructor for a type level non-empty list

```haskell
data (path :: k) :> a
infixr 9 :>
```

branch1 :<|> branch2

```haskell
data a :<|> b = a :<|> b
infixr 8 :<|>
```

Types have shapes (type operators)

head :> tail

- For joining nodes
- Constructor for a type level non-empty list
- Not directly inhabitable

```
data (path :: k) :> a
infixr 9 :>
```

branch1 :<|> branch2

```
data a :<|> b = a :<|> b
infixr 8 :<|>
```

Types have shapes (type operators)

head :> tail

- For joining nodes
- Constructor for a type level non-empty list
- Not directly inhabitable

```
data (path :: k) :> a
infixr 9 :>
```

branch1 :<|> branch2

- For branching

```
data a :<|> b = a :<|> b
infixr 8 :<|>
```

Types have shapes (type operators)

head :> tail

- For joining nodes
- Constructor for a type level non-empty list
- Not directly inhabitable

```
data (path :: k) :> a
infixr 9 :>
```

branch1 :<|> branch2

- For branching
- Constructor for alternatives (disjunction)

```
data a :<|> b = a :<|> b
infixr 8 :<|>
```

Types have shapes (type operators)

$\mathit{head} :> \mathit{tail}$

- For joining nodes
- Constructor for a type level non-empty list
- Not directly inhabitable

```
data (path :: k) :> a
infixr 9 :>
```

$\mathit{branch1} :<|> \mathit{branch2}$

- For branching
- Constructor for alternatives (disjunction)
- Inhabitable via :<|>

```
data a :<|> b = a :<|> b
infixr 8 :<|>
```

APIs have shapes

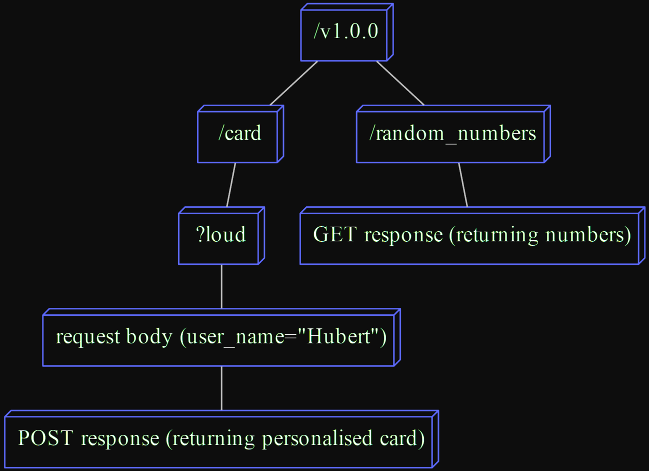

/v1.0.0
/card
/random_numbers
?loud
GET response (returning numbers)
request body (user_name="Hubert")
POST response (returning personalised card)

Figure 3: Your API as a tree

Shape as a type!

```
type MakeCard =
    "card"
    :> QueryFlag "loud"
    :> ReqBody '[FormUrlEncoded, JSON] Name
    :> Post '[JSON] PersonalisedCard

type RandomInt =
    "random_number" :> Get '[JSON] Int

type CardAPI = "v1.0.0" :> (MakeCard :<|> RandomInt)
```

How would a typed API even work?

Before we can type the APIs, I have to explain some “fundamentals”:

- DataKinds

How would a typed API even work?

Before we can type the APIs, I have to explain some “fundamentals”:

- DataKinds
- PolyKinds

How would a typed API even work?

Before we can type the APIs, I have to explain some “fundamentals”:

- DataKinds
- PolyKinds
- Data.Proxy

How would a typed API even work?

Before we can type the APIs, I have to explain some "fundamentals":

- DataKinds
- PolyKinds
- Data.Proxy
- GHC.TypeLits

How would a typed API even work?

Before we can type the APIs, I have to explain some “fundamentals”:

- DataKinds
- PolyKinds
- Data.Proxy
- GHC.TypeLits
- TypeFamilies

DataKinds, PolyKinds, Proxy & TypeLits

```haskell
import Data.Proxy
import GHC.TypeLits

-- | A concrete, poly-kinded proxy type
data Proxy a = Proxy

stringProxy :: Proxy "I AM A TYPE-LEVEL STRING!"
stringProxy = Proxy

listProxy :: Proxy '[Int, Bool, String]
listProxy = Proxy

symbolVal :: KnownSymbol str => Proxy str -> String
```

TypeFamilies

- Just functions at the type level

TypeFamilies

- Just functions at the type level
- We will use them in the associated form (appearing in a type class).

TypeFamilies

- Just functions at the type level
- We will use them in the associated form (appearing in a type class).
- These are called “associated type synonyms”.

TypeFamilies

- Just functions at the type level
- We will use them in the associated form (appearing in a type class).
- These are called “associated type synonyms”.
- They are a specific case of top-level “open” or “closed” type families, but give better errors and are clearer in their intentions.

Silly type family example

```haskell
class Frobable a where
  type FrobingResult a -- Associated type synonym

  frob :: Proxy a -> FrobingResult a

data MeaningOfLife

instance Frobable MeaningOfLife where
  type FrobResult MeaningOfLife = Int

  frob :: Proxy MeaningOfLife -> FrobResult MeaningOfLife
  frob _ = 42
```

Silly type family example

```
data EatsBools

widget :: Proxy (EatsBools :> MeaningOfLife)
widget = Proxy

instance Frobable rem => Frobable (EatsBools :> rem) where
  type FrobResult (EatsBools :> rem) =
    Bool -> Maybe (FrobResult rem)

  frob :: Proxy (EatsBools :> rem)
       -> FrobResult (EatsBools :> rem)
  frob _ True = Just $ frob (Proxy :: Proxy rem)
  frob _ False = Nothing
```

The results

```
> :t frob
frob :: Frobable a => Proxy a -> FrobResult a

> :t widget
widget :: Proxy (EatsBools :> MeaningOfLife)

> :t frob widget
frob widget :: FrobResult (EatsBools :> MeaningOfLife)

> let x = frob widget
> :t x
```

The results

```
> :t frob
frob :: Frobable a => Proxy a -> FrobResult a

> :t widget
widget :: Proxy (EatsBools :> MeaningOfLife)

> :t frob widget
frob widget :: FrobResult (EatsBools :> MeaningOfLife)

> let x = frob widget
> :t x

x :: Bool -> Maybe Int
```

Recap

- Your API has a tree-like shape.

Recap

- Your API has a tree-like shape.
- The tree-like shape of your API can be expressed with a type.

Recap

- Your API has a tree-like shape.
- The tree-like shape of your API can be expressed with a type.
- Servant defines some type operators: (:>) and (:<|>).

Recap

- Your API has a tree-like shape.
- The tree-like shape of your API can be expressed with a type.
- Servant defines some type operators: (:>) and (:<|>).
- DataKinds, TypeLits and Proxies help us write this type.

Recap

- Your API has a tree-like shape.
- The tree-like shape of your API can be expressed with a type.
- Servant defines some type operators: (:>) and (:<|>).
- DataKinds, TypeLits and Proxies help us write this type.
- Type families allow us to take an API type and manipulate it.

Types for clarity

Let's take what we've learned and see if we can express our business logic by itself, free of boilerplate.

HasServer, a dumping ground for boilerplate

```haskell
class HasServer layout where
  type Server layout :: *
  route :: Proxy layout
        -> Server layout
        -> RoutingApplication

instance HasServer Delete where
  type Server Delete = EitherT (Int, String) IO ()

  route Proxy action request respond
    | pathIsEmpty request
    && requestMethod request == methodDelete = do
        e <- runEitherT action
        ...
```

Distribute your alternatives

```haskell
instance (HasServer a, HasServer b) =>
        HasServer (a :<|> b) where

  type Server (a :<|> b) = Server a :<|> Server b

  route Proxy (a :<|> b) request respond =
    route pa a request $ \ mResponse ->
      if isMismatch mResponse
        then route pb b request $ \mResponse' ->
                respond (mResponse <> mResponse')
        else respond mResponse

    where pa = Proxy :: Proxy a
          pb = Proxy :: Proxy b
```

Unravelling the type one step at a time

```
instance (KnownSymbol sym, FromText a, HasServer sub)
      => HasServer (QueryParam sym a :> sub) where

  type Server (QueryParam sym a :> sub) = Maybe a -> Server sub

  route Proxy subserver req respond = do
    let query = parseQueryText $ rawQueryString req
        paramname = cs $ symbolVal ps
        param = fmap fromText
              . join $ lookup paramname query
    route (Proxy :: Proxy sub)
          (subserver param)
          request respond
          ...
```

But how does the content-typing work?

```haskell
type MakeCard =
    "card"
    :> QueryFlag "loud"
    :> ReqBody '[FormUrlEncoded, JSON] Name
    :> Post '[JSON] PersonalisedCard

type RandomInt =
    "random_number" :> Get '[JSON] Int

type CardAPI = "v1.0.0" :> (MakeCard :<|> RandomInt)
```

We seperate handling of content types

```
instance ToFormUrlEncoded Name where
    toFormUrlEncoded (Name full) =
      [("full_name", full)]

instance FromFormUrlEncoded Name where
    fromFormUrlEncoded xs =
        Name <$> note "specify full_name" (lookup "full_name" xs)

instance FromJSON PersonalisedCard
instance ToJSON PersonalisedCard
```

. . .

Business logic is now isolated

```
server :: Server CardAPI
server = makeCard :<|> randomNumber

makeCard :: Monad m
         => Bool -> Name -> m PersonalisedCard
makeCard loud (Name full_name) =
    return . PersonalisedCard $
      if loud
        then "HELLO " <> toUpper full_name <> "!!1"
        else "Hello " <> full_name <> "."

randomNumber :: Monad m => m Int
randomNumber = return 4
```

API type to documentation.

```haskell
docs :: HasDocs layout => Proxy layout -> API

instance ToParam (QueryFlag "loud") where
  toParam _ =
    DocQueryParam "loud"
                  ["true", "false"]
                  "Get the personalised card loudly.\
                  \ Default is false."
                  Flag
```

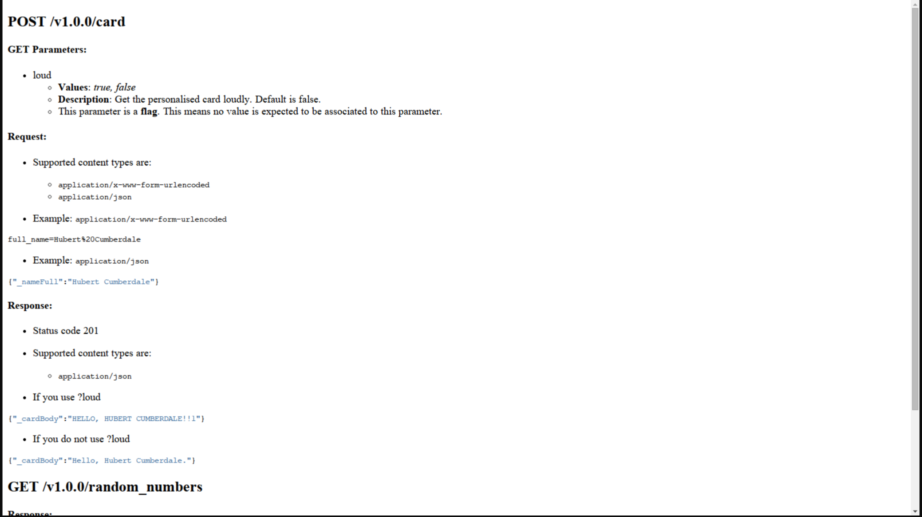
Type errors will make you define instances

```
instance ToSample Int where
  toSample = Just 4 -- Fair dice roll

instance ToSample Name where
  toSample = Just $ Name "Hubert Cumberdale"

instance ToSample PersonalisedCard where
  toSamples =
    [ ("If you use ?loud",
      , PersonalisedCard "HELLO, HUBERT CUMBERDALE!!1")
    , ("If you do not use ?loud"
      , PersonalisedCard "Hello, Hubert Cumberdale.")
    ]
```

Now you can markdown the things

```
docs :: HasDocs layout => Proxy layout -> API

markdown :: API -> String
```

Converted to HTML



POST /v1.0.0/card

GET Parameters:

- loud
  - **Values**: *true, false*
  - **Description**: Get the personalised card loudly. Default is false.
  - This parameter is a **flag**. This means no value is expected to be associated to this parameter.

Request:

- Supported content types are:
  - `application/x-www-form-urlencoded`
  - `application/json`
- Example: `application/x-www-form-urlencoded`

```
full_name=Hubert%20Cumberdale
```

- Example: `application/json`

```
{"_nameFull":"Hubert Cumberdale"}
```

Response:

- Status code 201
- Supported content types are:
  - `application/json`
- If you use ?loud

```
{"_cardBody":"HELLO, HUBERT CUMBERDALE!!1"}
```

- If you do not use ?loud

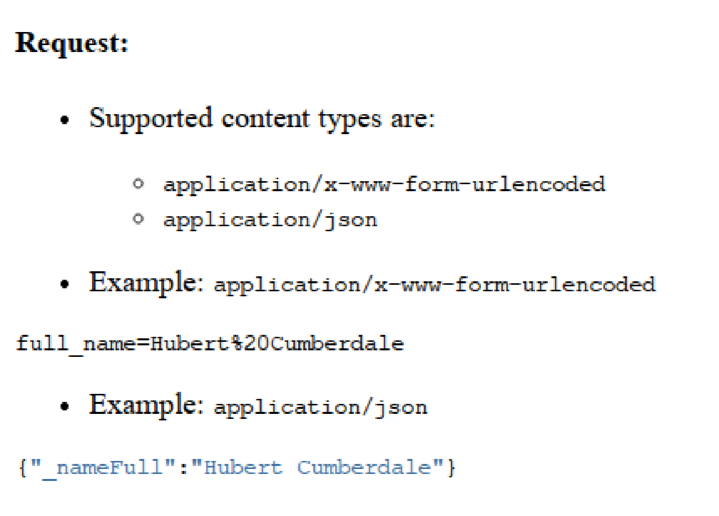
```
{"_cardBody":"Hello, Hubert Cumberdale."}
```

GET /v1.0.0/random_numbers

Response:

Figure 4: Auto-generated docs

Converted to HTML



Request:

- Supported content types are:
  - `application/x-www-form-urlencoded`
  - `application/json`
- Example: `application/x-www-form-urlencoded`

```
full_name=Hubert%20Cumberdale
```

- Example: `application/json`

```
{"_nameFull":"Hubert Cumberdale"}
```

Figure 5: Auto-generated docs (zoomed to request)

Clients for free (tackling complexity)

Consider an unversioned API that has:

- Three breaking changes

How many changes must you make to fix all of the things?

Clients for free (tackling complexity)

Consider an unversioned API that has:

- Three breaking changes
- Six users

How many changes must you make to fix all of the things?

Clients for free (tackling complexity)

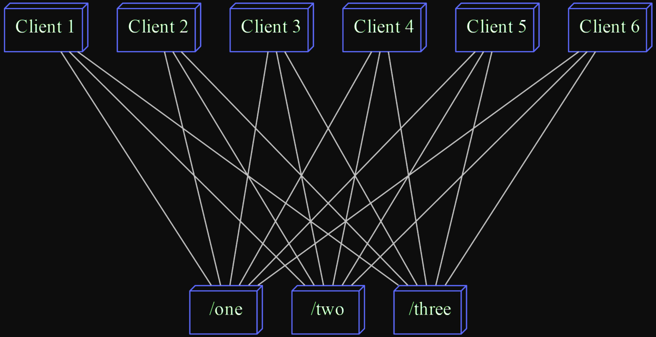


Figure 6: Complexity to maintain

Writing clients, the lazy way

```
createCard
    :: Bool
    -> Name
    -> BaseUrl
    -> EitherT ServantError IO PersonalisedCard

getDice
    :: BaseUrl
    -> EitherT ServantError IO [Int]

(createCard :<|> getDice) = client cardApi
```

How could such a magical unicorn exist?

```
client
  :: HasClient layout => Proxy layout -> Client layout
```

The magic: distribute (:<|>)

```
class HasClient layout where
  type Client layout :: *
  clientWithRoute
    :: Proxy layout -> Req -> Client layout


  instance (HasClient a, HasClient b)
        => HasClient (a :<|> b) where
    type Client (a :<|> b) = Client a :<|> Client b
    clientWithRoute Proxy req =
      clientWithRoute (Proxy :: Proxy a) req :<|>
      clientWithRoute (Proxy :: Proxy b) req
```

Clients for free!

```haskell
createCard
    :: Bool
    -> Name
    -> BaseUrl
    -> EitherT ServantError IO PersonalisedCard

getDice
    :: BaseUrl
    -> EitherT ServantError IO [Int]

(createCard :<|> getDice) = client cardApi
```

Type safe URLs

```
safeLink
    :: forall endpoint api. ( IsElem endpoint api
                            , HasLink endpoint)
    => Proxy api
    -> Proxy endpoint
    -> MkLink endpoint
```

With input!

```haskell
let nums = Proxy :: Proxy ("v1.0.0" :> RandomInts)
print $ safeLink cardApi nums
```

With input!

```
let nums = Proxy :: Proxy ("v1.0.0" :> RandomInts)
print $ safeLink cardApi nums

>> v1.0.0/random_numbers
```

With input!

```
let nums = Proxy :: Proxy ("v1.0.0" :> RandomInts)
print $ safeLink cardApi nums
```

```
>> v1.0.0/random_numbers
```

```
let make_card = Proxy :: Proxy ("v1.0.0" :> MakeCard)
let f :: Bool -> URI = safeLink cardApi make_card
traverse_ print [f True, f False]
```

With input!

```
let nums = Proxy :: Proxy ("v1.0.0" :> RandomInts)
print $ safeLink cardApi nums

>> v1.0.0/random_numbers


let make_card = Proxy :: Proxy ("v1.0.0" :> MakeCard)
let f :: Bool -> URI = safeLink cardApi make_card
traverse_ print [f True, f False]

>> v1.0.0/card?loud
>> v1.0.0/card
```

Conclusion

- Types and webservices can be friends

Conclusion

- Types and webservices can be friends
- By defining your API as a type, you can get for free:

Conclusion

- Types and webservices can be friends
- By defining your API as a type, you can get for free:
  - Server boilerplate

Conclusion

- Types and webservices can be friends
- By defining your API as a type, you can get for free:
  - Server boilerplate
  - Documentation

Conclusion

- Types and webservices can be friends
- By defining your API as a type, you can get for free:
  - Server boilerplate
  - Documentation
  - Clients (Haskell, jquery, PureScript)

Conclusion

- Types and webservices can be friends
- By defining your API as a type, you can get for free:
  - Server boilerplate
  - Documentation
  - Clients (Haskell, jquery, PureScript)
  - Safe links