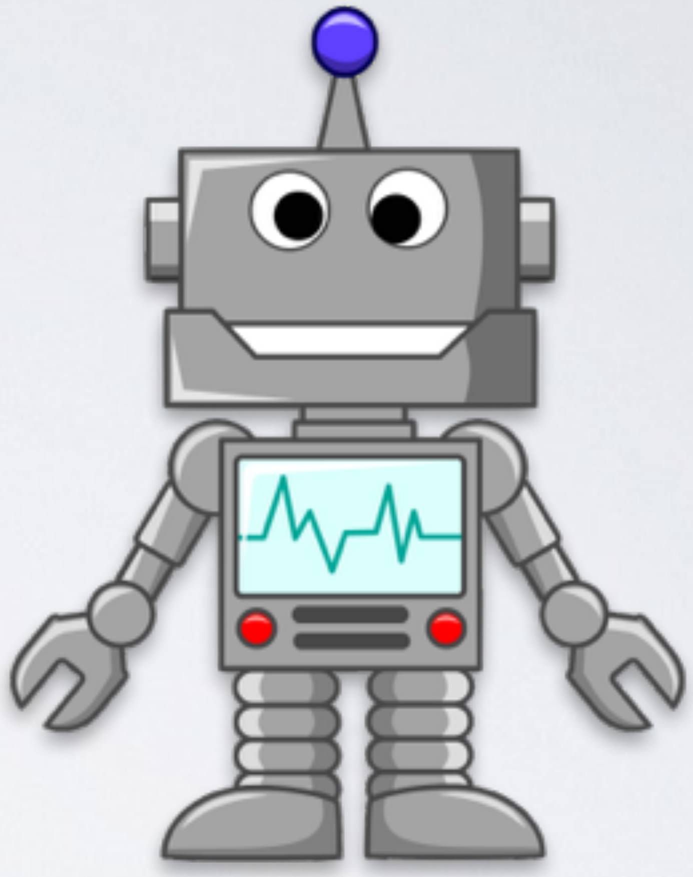
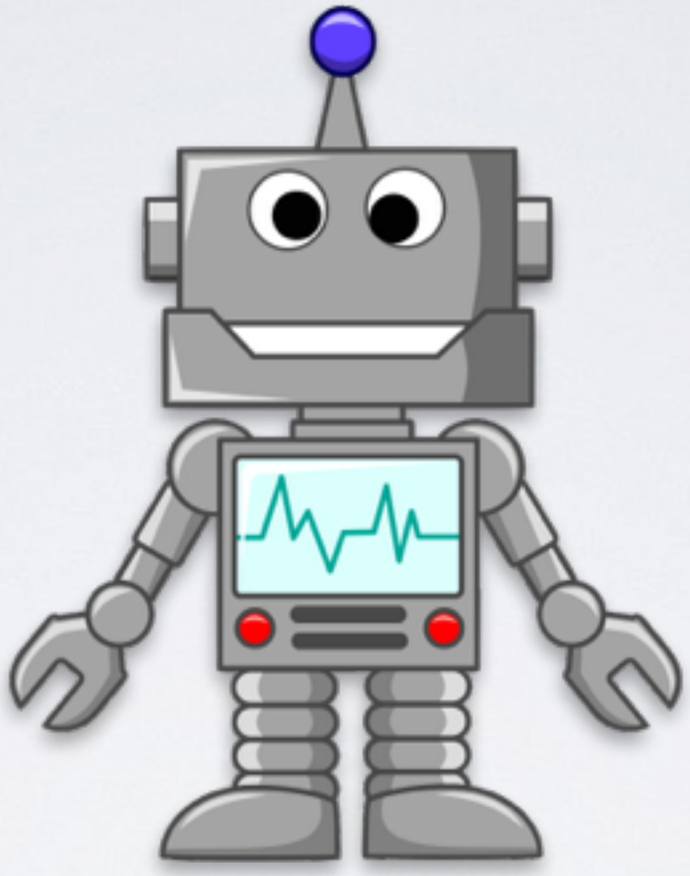


SERVANT

Type-Level DSLs for Web APIs



AUTHORS

Julian K. Arni, Alp Mestanogullari, Sönke Hahn

The boss says he wants a web service.

You start writing the code

Client + Tests

```haskell
import Test.Hspec

main = hspec $
  describe "Web Test" $
    it "Should say hello world" $
     (clientGet "/") `shouldReturn`
       Right (“Hello World” :: Html)
```

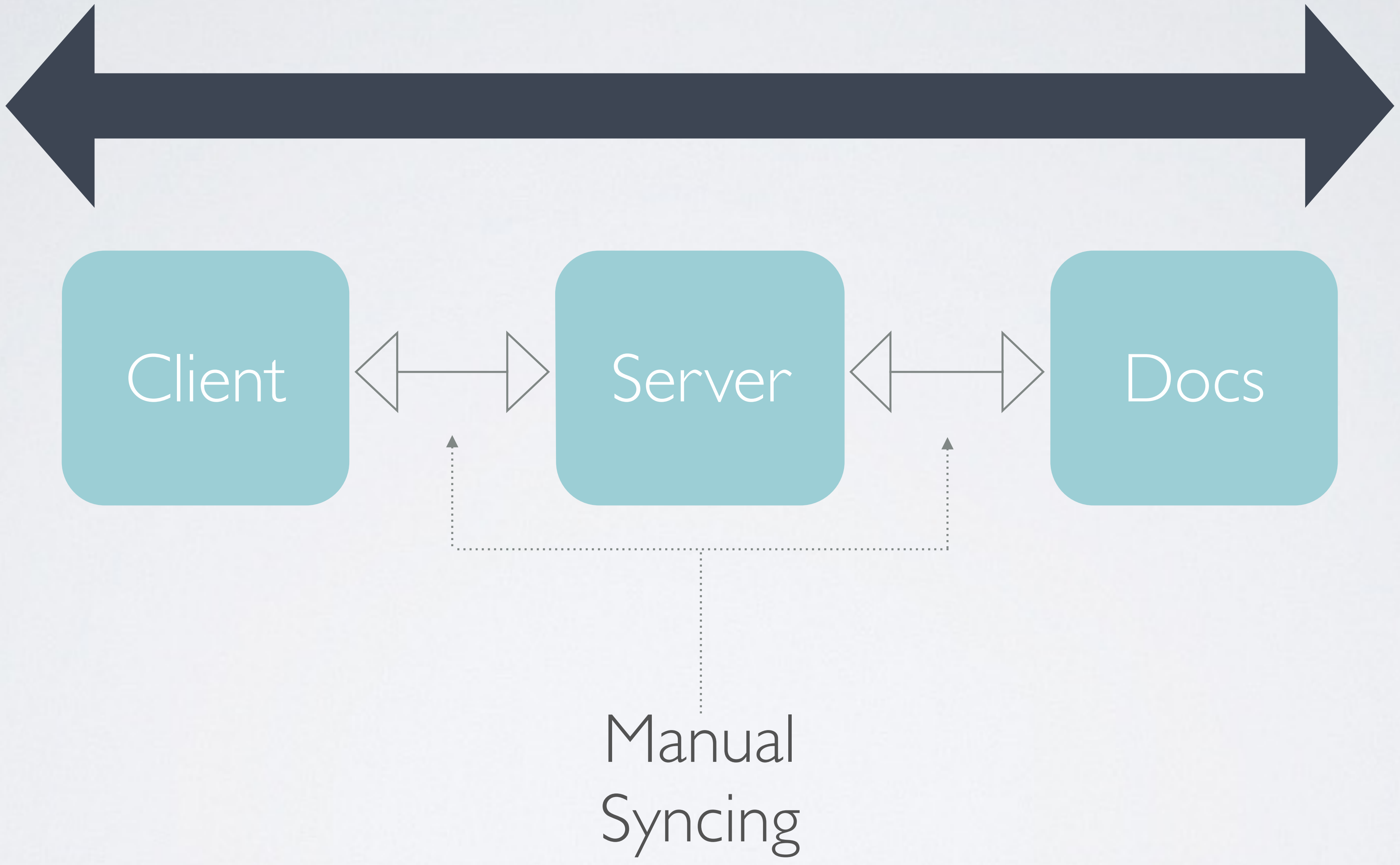
Server

```haskell
import Web.Scotty

main = scotty 80 $ do
  get "/" $ do
    html "Hello World!"
```

Docs

```
# API Docs
==============

## GET “/“ - HTML
  - 200: “Hello World”
```

3 code bases

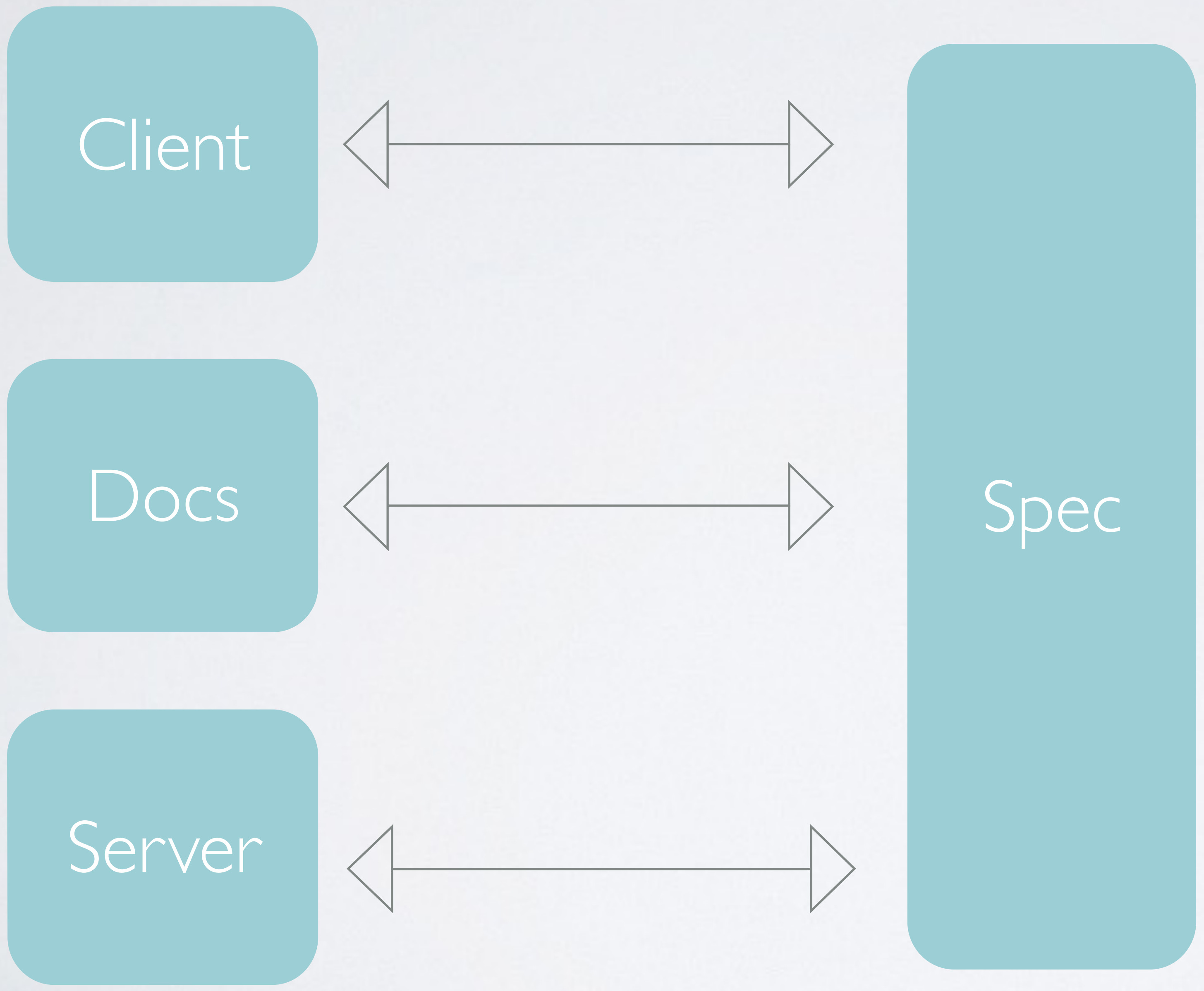


Maintenance Woes

- Repetitious, violates ‘ DRY ‘ principle
- Error prone
- Tedious, time-wasting
- Complex, cognitive overhead

Solution

API specification as first class citizen



- Paths
  - Variable captures
- Query Param(s)/ Flags
- Models
  - Request bodies
  - Response bodies
- Content-types
- Response codes

API Specification Promotion

```haskell
type TodoAPI =
      "todo" :> Capture "id" TodoId :> Get '[JSON] Todo
 :<|>
      "todo" :> QueryParam "completed" Completed :> Get '[JSON] [Todo]
 :<|>
      "todo" :> ReqBody '[JSON] NewTodo :> Post '[JSON] Todo
 :<|>
      "todo" :> Capture "id" TodoId :> Delete '[JSON] ()
 :<|>
      "todo" :> Capture "id" TodoId
             :> ReqBody '[JSON] NewTodo
             :> Put '[JSON] Todo

   -- GET    /todo/:id
   -- GET    /todo?completed=false
   -- POST   /todo
   -- DELETE /todo/:id
   -- PUT    /todo/:id
```

API Specification Promotion

```haskell
type API path capture requestBody returnType contentTypes =
   path :>
     ( capture :> Get contentTypes returnType
  :<|> Get contentTypes [returnType]
  :<|> capture :> requestBody :> Put contentTypes returnType
  :<|> capture :> Delete contentTypes ()
  :<|> requestBody :> Post contentTypes returnType
     )

type TodoAPI =
    API “todo" (Capture "id" TodoId) (ReqBody '[JSON] Description)
       Todo ‘[JSON, HTML]
```

Type-safe web handlers

```haskell
---------------------------------------------------------------------------
-- | Todo API
type TodoAPI =
      "todo" :> Capture "id" TodoId :> Get '[JSON] (Maybe Todo)
 :<|> "todo" :> QueryParam "completed" Completed :> Get '[JSON] [Todo]
 :<|> "todo" :> ReqBody '[JSON] NewTodo :> Post '[JSON] Todo
 :<|> "todo" :> Capture "id" TodoId :> Delete '[JSON] ()
 :<|> "todo" :> Capture "id" TodoId :> ReqBody '[JSON] NewTodo
             :> Put '[JSON] (Maybe Todo)
-------------------------------------------------------------------------------
todoEndpoints :: Server TodoAPI
todoEndpoints = todoGet :<|> todoGetAll :<|> todoCreate :<|>
                todoDelete :<|> todoUpdate
-------------------------------------------------------------------------------
todoGet    :: TodoId -> EitherT ServantErr IO (Maybe Todo)
todoGetAll :: Maybe Completed -> EitherT ServantErr IO [Todo]
todoCreate :: NewTodo -> EitherT ServantErr IO Todo
todoDelete :: TodoId -> EitherT ServantErr IO ()
todoUpdate :: TodoId -> NewTodo -> EitherT ServantErr IO (Maybe Todo)
```

What if I want my own monad?

```haskell
newtype m :~> n = Nat { unNat :: forall a. m a -> n a}
```

Natural Transformation

```haskell
-- | Core Todo Type
newtype TodoApp a = TodoApp {
    runTodo :: ReaderT Config (EitherT TodoError IO) a
  } deriving ( MonadIO, MonadReader Config
             , Applicative, Monad, Functor, MonadError TodoError )

-- | Application
app :: Config -> Application
app cfg = serve (Proxy :: Proxy TodoAPI) server
  where
    server :: Server TodoAPI
    server = enter todoToEither todoEndpoints

    todoToEither :: TodoApp :~> EitherT ServantErr IO
    todoToEither = Nat $ flip bimapEitherT id errorToServantErr
                       . flip runReaderT cfg . runTodo

    errorToServantErr :: TodoError -> ServantErr
    errorToServantErr = const err500
```

Your custom monad stack

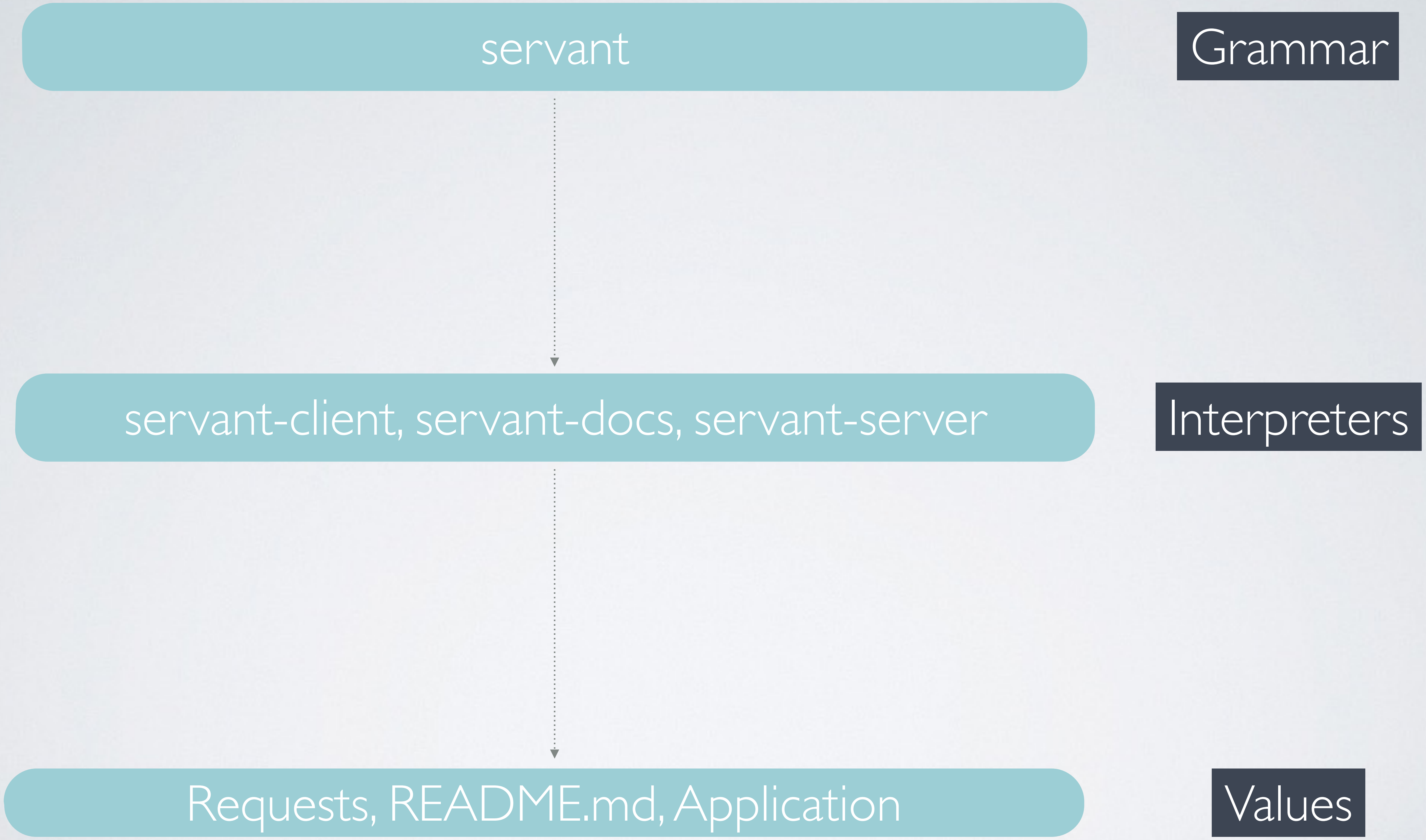
```haskell
todoAPI :: ServerT TodoAPI TodoApp
todoAPI = todoGetAll :<|> todoGet :<|> todoDelete :<|>
          todoUpdate :<|> todoCreate
---------------------------------------------------------------
todoGetAll  :: Maybe OrderBy -> Maybe Completed -> TodoApp [Todo]
todoGet     :: TodoId -> TodoApp (Maybe Todo)
todoDelete :: TodoId -> TodoApp ()
todoUpdate  :: TodoId -> NewTodo -> TodoApp (Maybe Todo)
todoCreate  :: NewTodo -> TodoApp Todo
```

Extensible API

```haskell
data AuthToken

type TodoAPI = "todo" :> AuthToken :> Capture "todoid" TodoId
                      :> Get '[JSON] Todo

instance HasServer api => HasServer (AuthToken :> api) where
instance HasClient api => HasClient (AuthToken :> api) where
instance HasDocs   api => HasDocs   (AuthToken :> api) where

-- Web handler
getTodo :: AuthToken -> TodoId -> TodoApp Todo
-- Client function
getTodo :: AuthToken -> TodoId -> EitherT ServantError IO Todo
```

Serve it

```haskell
{-# LANGUAGE RecordWildCards   #-}
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE OverloadedStrings #-}
module Main ( main ) where

import Todo.App                             ( app, getConfig )
import Network.Wai.Handler.Warp             ( run )

-- | Application Entry Point
main :: IO ()
main = do
 config@Config { port = port } <- getConfig
 putStrLn $ "Running server on " ++ show port ++ "..."
 run port (app config)
```


servant
Grammar
servant-client, servant-docs, servant-server
Interpreters
Requests, README.md, Application
Values

servant-jQuery

```
function gettodo(id, onSuccess, onError)
{
  $.ajax(
    { url: '/todo/' + encodeURIComponent(id) + ''
    , success: onSuccess
    , error: onError
    , type: 'GET'
    });
}

function puttodo(id, body, onSuccess, onError)
{
  $.ajax(
    { url: '/todo/' + encodeURIComponent(id) + ''
    , success: onSuccess
    , data: JSON.stringify(body)
    , contentType: 'application/json'
    , error: onError
    , type: 'PUT'
    });
}
```

```
function gettodo(completed, onSuccess, onError)
{
  $.ajax(
    { url: '/todo' + '?completed=' + encodeURIComponent(completed)
    , success: onSuccess
    , error: onError
    , type: 'GET'
    });
}

function posttodo(body, onSuccess, onError)
{
  $.ajax(
    { url: '/todo'
    , success: onSuccess
    , data: JSON.stringify(body)
    , contentType: 'application/json'
    , error: onError
    , type: 'POST'
    });
}

function deletetodo(id, onSuccess, onError)
{
  $.ajax(
    { url: '/todo/' + encodeURIComponent(id) + ''
    , success: onSuccess
    , error: onError
    , type: 'DELETE'
    });
}
```

lackey - ruby functions

```ruby
def get_todo_id(excon, id)
  excon.request(
    method: :get,
    path: "/todo/#{id}",
    headers: {},
    body: nil
  )
end

def get_todo_completed(excon, completed: nil)
  excon.request(
    method: :get,
    path: "/todo?&completed=#{completed}",
    headers: {},
    body: nil
  )
end
```

```ruby
def post_todo(excon, body)
  excon.request(
    method: :post,
    path: "/todo",
    headers: {},
    body: body
  )
end

def delete_todo_id(excon, id)
  excon.request(
    method: :delete,
    path: "/todo/#{id}",
    headers: {},
    body: nil
  )
end

def put_todo_id(excon, id, body)
  excon.request(
    method: :put,
    path: "/todo/#{id}",
    headers: {},
    body: body
  )
end
```

servant-docs

```
## GET /todo

#### GET Parameters:

- completed
     - **Values**: *true, false*
     - **Description**: filter todos by completed status


#### Response:

- Status code 200
- Headers: []

- Supported content types are:

    - `application/json`

- No response body

## POST /todo

#### Request:

- Supported content types are:

    - `application/json`

#### Response:

- Status code 201
- Headers: []
```

```
## DELETE /todo/:id

#### Captures:

- *id*: Id of Todo

#### Response:

- Status code 200
- Headers: []

- Supported content types are:

    - `application/json`

- No response body

## GET /todo/:id

#### Captures:

- *id*: Id of Todo

#### Response:

- Status code 200
- Headers: []

- Supported content types are:

    - `application/json`

- No response body

## PUT /todo/:id

#### Captures:

- *id*: Id of Todo
```

ghcjs-servant-client

```haskell
-- Type
type API = "todo" :> Get [Todo] -- GET /todos
      :<|> "todo" :> ReqBody NewTodo :> Post Todo -- POST /books

-- Functions
getAllTodos :: BaseUrl -> EitherT String IO [Todo]
createTodo :: TodoId -> NewTodo -> EitherT String IO Todo
(getAllTodos :<|> createTodo) = client (Proxy :: Proxy API)
```

servant-client

```haskell
------------------------------------------------------------------------------
-- | Client Handlers
createUser
  :<|> todoGetAll
  :<|> todoGet
  :<|> todoDelete
  :<|> todoUpdate
  :<|> todoCount
  :<|> todoCreate = client (Proxy :: Proxy API) (BaseUrl Http "localhost" 8000)
------------------------------------------------------------------------------
clientRequest :: IO ()
clientRequest =
  print =<< do runEitherT $ todoCreate token (NewTodo "walk dog")
```

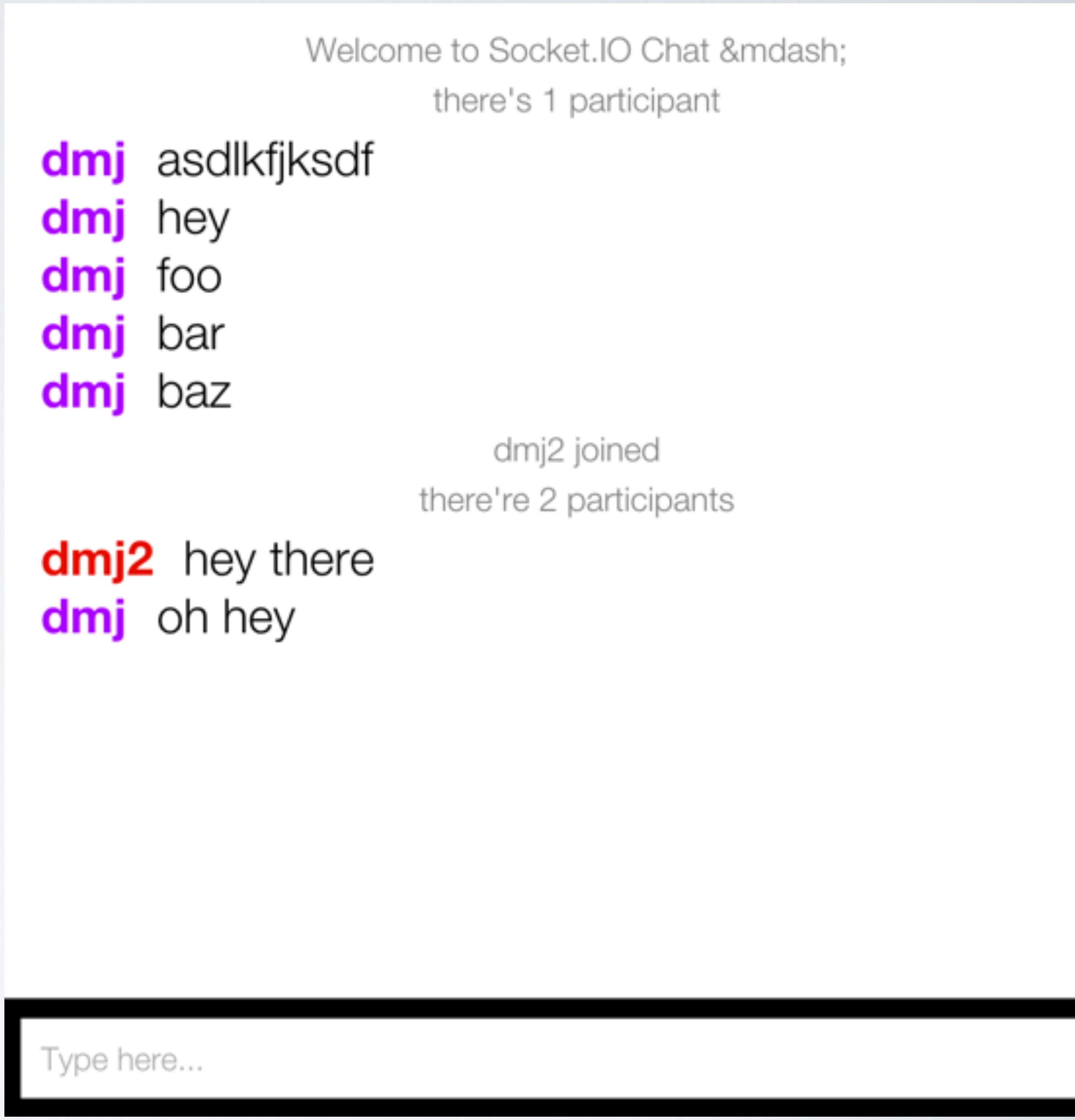
servant-mocks

```haskell
data User = User {
    name :: String
  , age  :: Int
  } deriving (Show, Generic)

instance ToJSON User

instance Arbitrary User where
  arbitrary = liftM2 User arbitrary arbitrary

type API = Get '[JSON] [User]

api :: Proxy API
api = Proxy

main :: IO ()
main = run 8000 (serve api $ mock api)
```

```
> Davids-MacBook-Pro-2: curl localhost:8000
[{"age":-27,"name":"\u0000\\Â¬Ãµ>\u001f8\u0007\u0014750Â¬â‰ Â¬ÄVâˆšÃ«m\u001b|
âˆšÃ®âˆšÃ¨Â¬Ã³âˆšÃ«Â¬ÃœeÂ¬Ü1\u000cÂ¬Ã»!"},{"age":27,"name":"\u0004e`)"},{"age":-2,"name":"Â¬Ä
\u000cÂ¬Ã‡âˆšÃ¯0$"},{"age":15,"name":"M'Â¬Âµ\u001ehFI&âˆšÜ\"LT2oâˆšâˆ'4\u0000\u0011\u0003P{RâˆšÖ"},
{"age":1,"name":"âˆšßVÂ¬Ã˜GfbÂ¬Ã®\u0012âˆšá"},{"age":5,"name":"HM,5âˆšÃ®IÂ¬Úgâˆšâˆ'[`âˆšÃ£
\u0008*Â¬Ù\u0006âˆšÃ¤\u000f"},{"age":-20,"name":"Â¬Ã¦CÂ¬Ã–Â¬Ã£_âˆšâˆžÂ¬ÃœâˆšÃ¨\u001dhÂ¬ÙY}\\âˆšÃ¹\
\@\u0001{5\u0015Â¬â‰ z"},{"age":-14,"name":"j/âˆšÃ¨8>\u0018\u0007gâˆšÂ¥Y4\\iH|\u000c\u001eI"},
{"age":-4,"name":"\"h\u0004GâˆšÃ®"},{"age":19,"name":"hâˆšógJ!y$yÂ¬ÒâˆšÒ\u0019\")G\u0011âˆšÃ¹
\u001dâˆšÜÂ¬Ã²\\\u0014\u0004âˆšòN]\u0015HH"},{"age":-18,"name":"\u0014qâˆšòÂ¬Ã©=Â¬Ã²
\u0007\n,.Â¬âˆ's+~G\u00135Â¬Ã‰~âˆšÖÂ¬èâˆšÓ\u0001Â¬Ã˜Zl\u000e|k"},{"age":14,"name":"Â¬Ã‡"},
{"age":-4,"name":"[Â¬Ã¦pf3\u000fw-âˆšÃ»Â¬Ã«84Â¬Ã‡*|"},{"age":-11,"name":"âˆšé
\u00067\u001cK0\u0010qÂ¬Ã†Â¬Ã¦\u0006Â¬ó:âˆšÃ¹f%q"},{"age":24,"name":"rE\tâˆšÃ»v.:\u000beUâˆšÃœc
\u000e?\u0014[aâˆšÃ§âˆšÃªE*!)k\tÂ¬â€¢\u0006\u000e&["}]%
```

websockets via Engine-IO

```haskell
type API = "socket.io" :> Raw :<|> Raw

api :: Proxy API
api = Proxy

server :: WaiMonad () -> Server API
server sHandler = socketIOHandler
             :<|> serveDirectory "resources"
  where
    socketIOHandler req respond =
       toWaiApplication sHandler req respond

app :: WaiMonad () -> Application
app sHandler = serve api $ server sHandler

port :: Int
port = 3001

main :: IO ()
main = do
    state <- ServerState <$> STM.newTVarIO 0
    sHandler <- SocketIO.initialize
        waiAPI (eioServer state)
    putStrLn $ "Running on " <> show port
    run port $ app sHandler
```



Welcome to Socket.IO Chat &mdash;

there's 1 participant

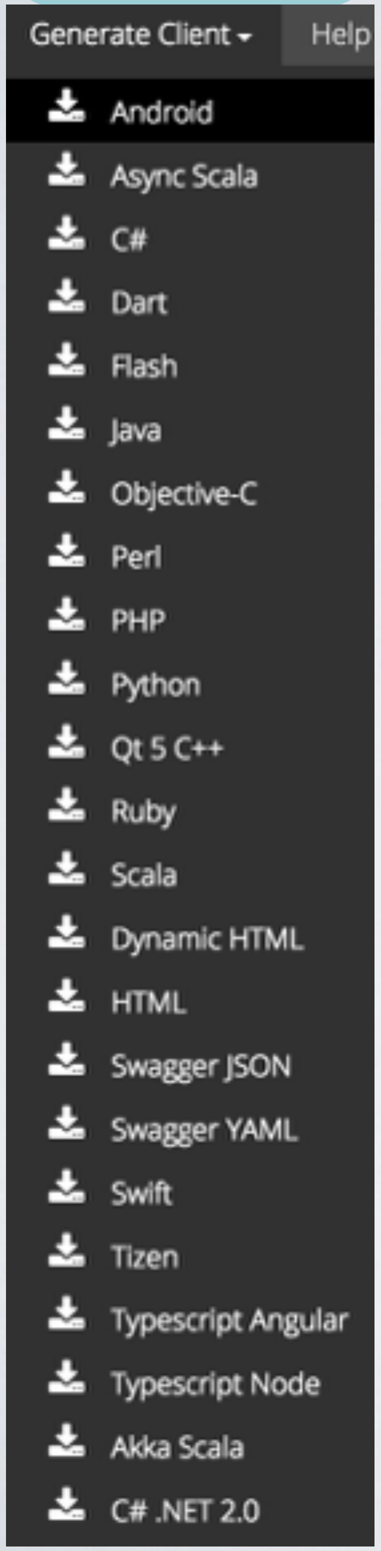
dmj asdlkfjksdf
dmj hey
dmj foo
dmj bar
dmj baz

dmj2 joined

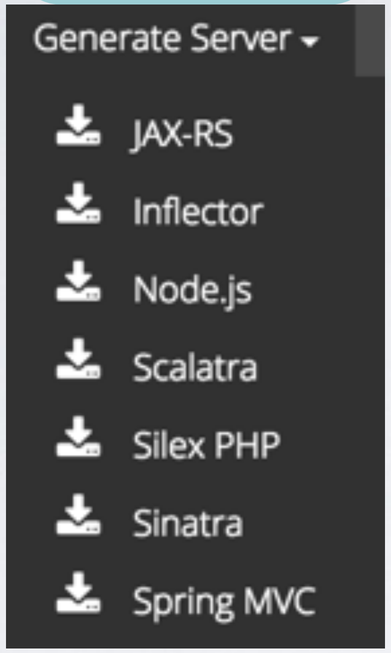
there're 2 participants

dmj2 hey there
dmj oh hey

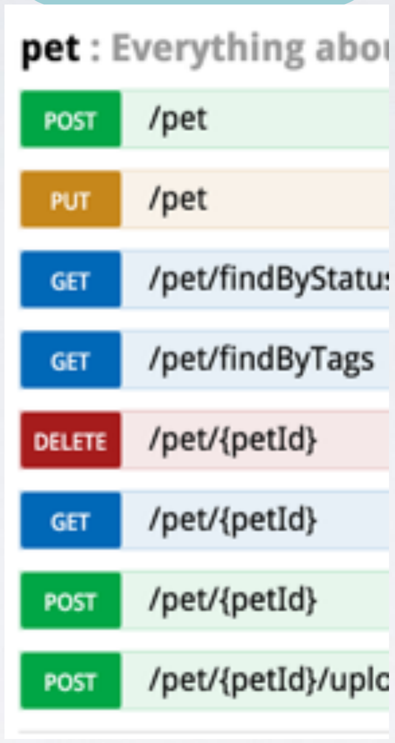
Client



Server

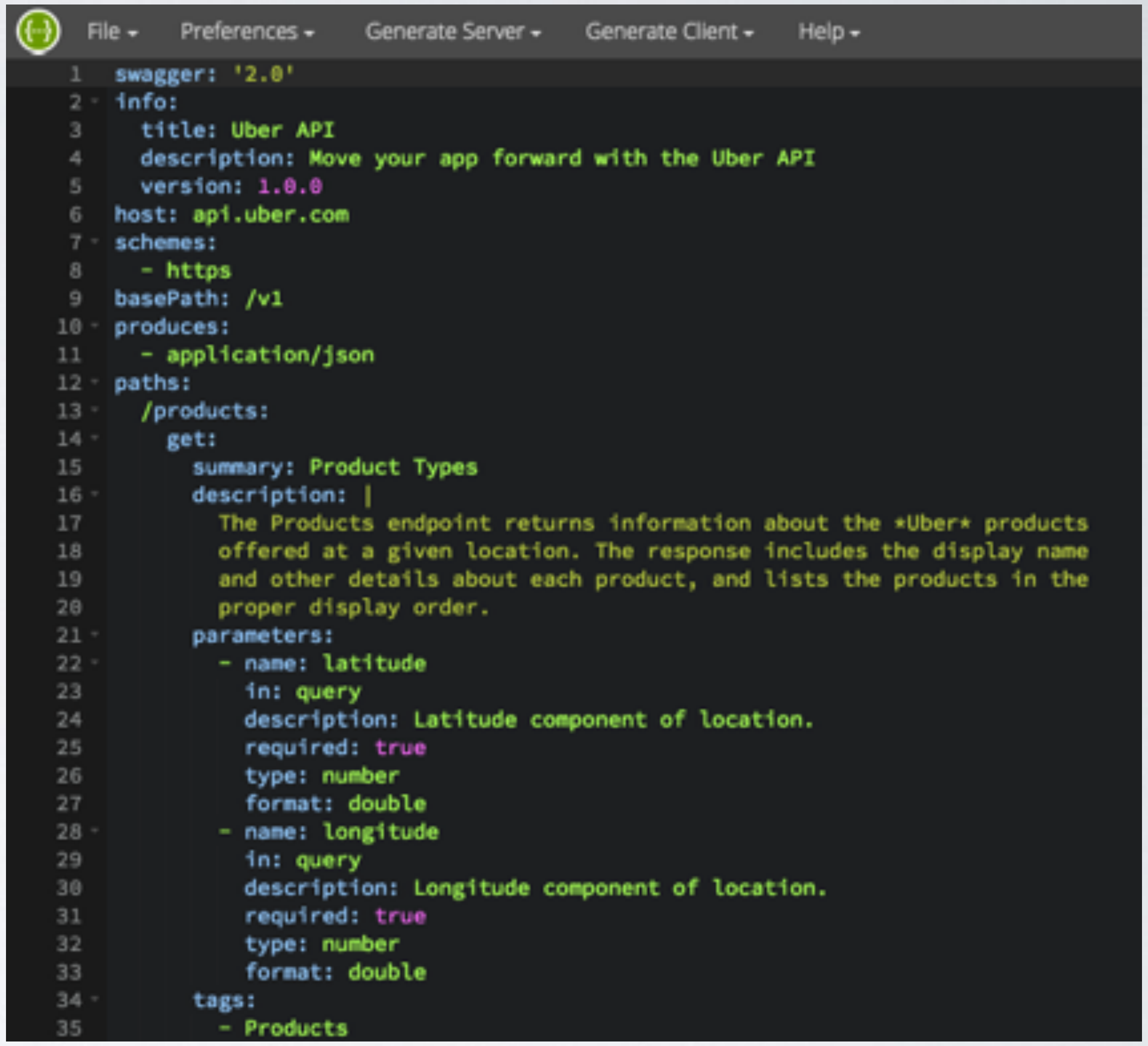


Docs



SWAGGER

The World's Most Popular Framework for APIs.

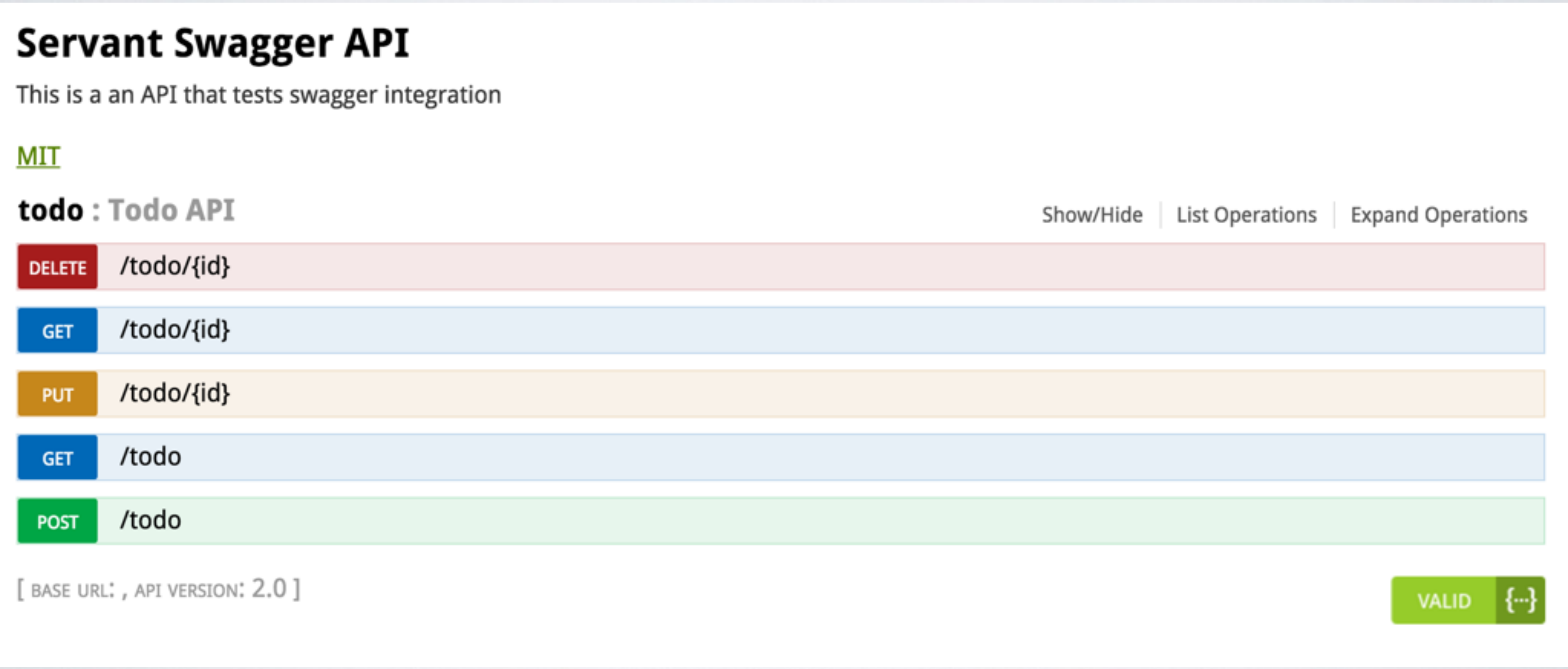


```
swagger: '2.0'
info:
  title: Uber API
  description: Move your app forward with the Uber API
  version: 1.0.0
host: api.uber.com
schemes:
  - https
basePath: /v1
produces:
  - application/json
paths:
  /products:
    get:
      summary: Product Types
      description: |
        The Products endpoint returns information about the *Uber* products
        offered at a given location. The response includes the display name
        and other details about each product, and lists the products in the
        proper display order.
      parameters:
        - name: latitude
          in: query
          description: Latitude component of location.
          required: true
          type: number
          format: double
        - name: longitude
          in: query
          description: Longitude component of location.
          required: true
          type: number
          format: double
      tags:
        - Products
```

JSON/YAML

servant-swagger

(Coming to a hackage near you)



Servant Swagger API

This is a an API that tests swagger integration

MIT

todo : Todo API

DELETE /todo/{id}

GET /todo/{id}

PUT /todo/{id}

GET /todo

POST /todo

[BASE URL: , API VERSION: 2.0]

Hello World servant-server

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DataKinds     #-}

import Data.Aeson
import Servant
import Servant.Server
import Control.Monad.Trans.Either
import Network.Wai.Handler.Warp
import Data.Text    (Text)
-- Types
data Hello = Hello

-- Serialization
instance ToJSON Hello where
  toJSON = const $ object [ "message" .= ("hello world" :: Text) ]

-- API specification
type API = "hello" :> Get '[JSON] Hello

-- API implementation
endpoints :: Server API
endpoints = helloWorld
  where
    helloWorld :: EitherT ServantErr IO Hello
    helloWorld = pure Hello

main :: IO ()
main = do
  putStrLn "Running on 8000"
  run 8000 $ serve (Proxy :: Proxy API) endpoints

> curl localhost:8000/hello   — { “message” : “hello world” }
```

What?

- Who put strings in my types?
- Why are there lists w/ backticks?
- What is (:>) and (:<|>) ?

```
"hello" :> Get '[JSON] Hello
```

- How do we get from a type to a web server?
- What is Proxy ?

```
run 8000 $ serve (Proxy :: Proxy API) endpoints
```

Review

```
> :type 1
— 1 :: Num a => a
> :kind Int
— Int :: *
- Maybe :: * -> *
- StateT :: * -> (* -> *) -> * -> *
```

- Values have Types
- Types have Kinds

Data Kinds & Kind Signatures

```
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE DataKinds      #-}

import GHC.TypeLits

data Response = Response

type TypeTuple = ‘((200 :: Nat), ("OK" :: Symbol), ('Response :: Response))
type TypeList  = '["a", "b"]
type TypeMap   = '[ '("a", 0), '("b", 1) ]

type ContentTypes = ‘[JSON, HTML]
```

- Values promoted to Types
- Types promoted to kinds
- Access to type level literals (Nat, Symbols)
- Access to type level lists, tuples, maps
- Types can be annotated with Kinds

Type Operators

```haskell
{-# LANGUAGE TypeOperators #-}

data path :> a
data l :<|> r = l :<|> r
data (:<|>) l r = l :<|> r
```

- Operator symbols in types can be written infix

Poly Kinds

```
data a :> b
type API = ("user" :: Symbol) :> (Get '[JSON] User :: *)

-- Main.hs:9:12-16: The first argument of ‘:>’ should have kind ‘*’,
--         but “api" has kind GHC.TypeLits.Symbol
--       In the type "api" :> Get '[JSON] User
--       In the type declaration for ‘API’
-- Compilation failed.
```

Why?

```
> :kind (:>)
(:>) :: * -> * -> *
```

Solution 1 - Wrong

```
data (a :: Symbol) :> b
```

```
> :kind (:>)
(:>) :: Symbol -> * -> *
```

Solution 2 - Enable PolyKinds

```
{-# LANGUAGE PolyKinds       #-}
> :kind (:>)
(:>) :: k -> k1 -> *
```

```
data (a :: k) :> b
```

Data.Proxy

```haskell
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE PolyKinds      #-}

data Proxy (t :: k) = Proxy

api :: Proxy API
api = Proxy

:kind Proxy
– Proxy :: k -> *
```

- A way to pass types as arguments to functions
- Access to types at runtime

Proxy Magic

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DataKinds     #-}

import Data.Proxy
import GHC.TypeLits
import Data.Type.Equality
import Data.Type.Bool

type TypeStatement = If (1 == 1) "true" "false"

main :: IO ()
main = do
  print $ symbolVal (Proxy :: Proxy TypeStatement)
  print $ symbolVal (Proxy :: Proxy "foo")

-- "true"
-- "foo"
```

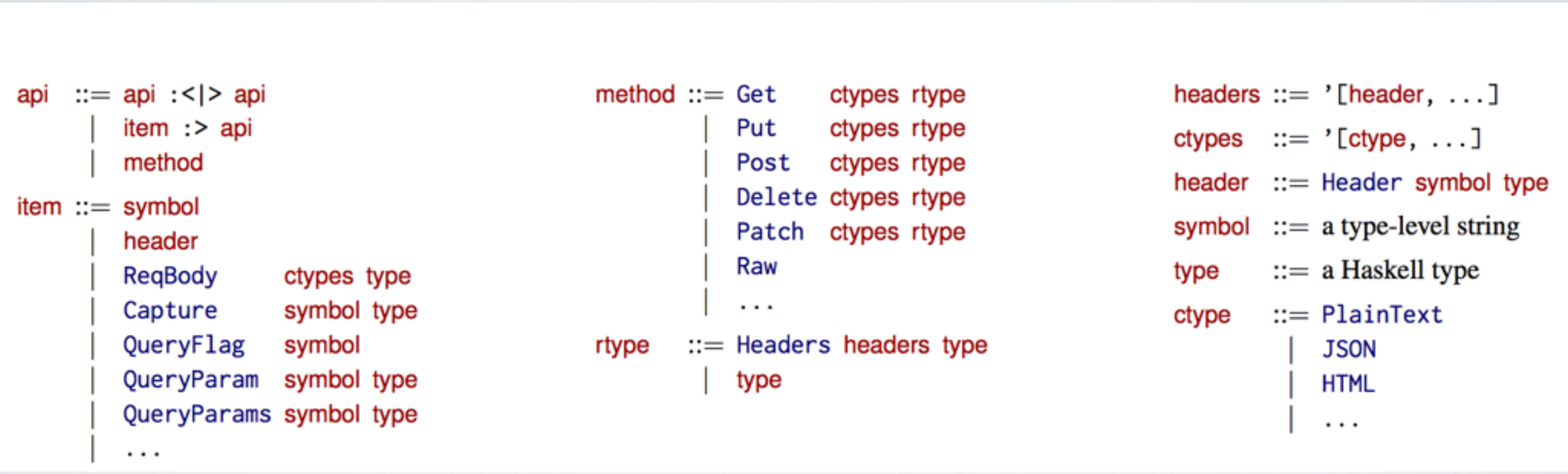
Deconstructing Type Level lists

```haskell
import Data.Proxy

-- Grammar
data JSON
data HTML

-- Values
data MimeType = Json | Html deriving Show

-- Single MimeType Interpreter
class ToMimeType a where toMimeType :: Proxy a -> MimeType

-- Single MimeType Instance
instance ToMimeType JSON where toMimeType Proxy = Json
instance ToMimeType HTML where toMimeType Proxy = Html

-- Multiple MimeTypes
class ToMimeTypes a where toMimeTypes :: Proxy a -> [MimeType]

-- Base case
instance ToMimeTypes '[] where toMimeTypes Proxy = []

-- Inductive step, type-level pattern matching
instance (ToMimeType x, ToMimeTypes xs) => ToMimeTypes (x ': xs) where
  toMimeTypes Proxy =
    toMimeType (Proxy :: Proxy x) : toMimeTypes (Proxy :: Proxy xs)

main :: IO ()
main = print $ toMimeTypes (Proxy :: Proxy '[JSON, HTML])
-- > [Json,Html]
```

Type level grammar

```haskell
{-# LANGUAGE PolyKinds      #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE DataKinds      #-}
{-# LANGUAGE TypeOperators  #-}

import GHC.TypeLits

-- Content Types
data HTML; data JSON

-- Verbs
data Put    (mimeTypes :: [*]) a
data Delete (mimeTypes :: [*]) a
data Get    (mimeTypes :: [*]) a
data Post   (mimeTypes :: [*]) a

-- Routing combinators
data (path :: k) :> rest
data left :<|> right

-- URL options
data Capture (name :: Symbol) typ
data QueryParam (name :: Symbol) typ

-- Request Body
data ReqBody (mimeTypes :: [*]) a

-- Custom Types
data TodoId
data Description
data Todo
```

Type Level Grammar



```
api    ::= api :<|> api
         | item :> api
         | method
item   ::= symbol
         | header
         | ReqBody     ctypes type
         | Capture     symbol type
         | QueryFlag   symbol
         | QueryParam  symbol type
         | QueryParams symbol type
         | ...

method ::= Get    ctypes rtype
         | Put    ctypes rtype
         | Post   ctypes rtype
         | Delete ctypes rtype
         | Patch  ctypes rtype
         | Raw
         | ...
rtype  ::= Headers headers type
         | type

headers ::= '[header, ...]
ctypes  ::= '[ctype, ...]
header  ::= Header symbol type
symbol  ::= a type-level string
type    ::= a Haskell type
ctype   ::= PlainText
          | JSON
          | HTML
          | ...
```

Typeclasses !

```
class HasDocs layout where
  docsFor :: Proxy layout -> (Endpoint, Action) -> API

class HasServer layout where
  type ServerT layout (m :: * -> *) :: *
  route :: Proxy layout -> Server layout -> RoutingApplication

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

Instances as interpretations

```
(KnownSymbol path, HasServer sublayout) => HasServer (path :> sublayout)
(HasServer a, HasServer b) => HasServer (a :<|> b)

HasServer (Get ctypes ())
HasServer (Put ctypes ())
HasServer (Post ctypes ())
HasServer (Delete ctypes ())
HasServer (Options ctypes ())

HasServer (Header sym a :> sublayout)
HasServer (Capture capture a :> sublayout)
```

Type Families

```haskell
{-# LANGUAGE TypeFamilies #-}
class HasStripe a where
  type Stripe a :: *

instance HasStripe Customer where
  type Stripe Customer = CustomerResponse

data Customer = Customer
data CustomerResponse

submitStripe
  :: ( MonadIO m
     , ToFormURLEncoded request
     , FromJSON response
     , response ~ Stripe request
     )
  => request -> m (Either String response)

-- :t submitStripe Customer:: MonadIO m => m (Either String CustomerResponse)
```

Example type-level traversal

```haskell
data a :> b; data a :<|> b
data Done

type API = "api" :> "user" :> Done
      :<|> "api" :> "todo" :> Done

class HasRoutes routes where
  toRoutes :: Proxy routes -> String -> [String]

-- Base Case
instance HasRoutes Done where
  toRoutes Proxy xs = [ xs ]

-- Recursive Step
instance (HasRoutes rest, KnownSymbol path)=> HasRoutes (path :> rest) where
  toRoutes Proxy xs = toRoutes (Proxy :: Proxy rest) newRoute
    where
      newRoute = xs ++ "/" ++ symbolVal (Proxy :: Proxy path)

-- Alternative
instance (HasRoutes a, HasRoutes b) => HasRoutes (a :<|> b) where
  toRoutes Proxy xs = toRoutes (Proxy :: Proxy a) xs <>
                      toRoutes (Proxy :: Proxy b) xs

getRoutes :: HasRoutes r => Proxy r -> [String]
getRoutes p = toRoutes p []

main :: IO ()
main = print $ getRoutes (Proxy :: Proxy API)
-- > main
-- > [“/api/user","/api/todo"] :: [String]
```

Example type-level traversal w/ type-family

```haskell
type API = "user" :> Get '[JSON] [User]                          -- GET /user
      :<|> "user" :> Capture "user" UserId :> Get '[JSON] User  -- GET /user/:userid

-- Class
class HasRoute a where
  type Route a
  toRoute :: Proxy a -> String -> Route a

-- Base Case
instance HasRoute (Get xs a) where
  type Route (Get xs a) = String
  toRoute Proxy str     = str

-- Recursive Step
instance (HasRoute rest, Show typ) => HasRoute (Capture name typ :> rest) where
  type Route (Capture name typ :> rest) = typ -> Route rest
  toRoute Proxy path typ = toRoute (Proxy :: Proxy rest) newPath
     where newPath = path ++ "/" ++ show typ

-- Recursive Step
instance (HasRoute rest, KnownSymbol path) => HasRoute (path :> rest) where
  type Route (path :> rest) = Route rest
  toRoute Proxy path = toRoute (Proxy :: Proxy rest) newPath
    where newPath = path ++ "/" ++ symbolVal (Proxy :: Proxy path)

-- Alternate
instance (HasRoute a, HasRoute b) => HasRoute (a :<|> b) where
  type Route (pathA :<|> pathB) = Route pathA :<|> Route pathB
  toRoute Proxy path = toRoute (Proxy :: Proxy a) path :<|> toRoute (Proxy :: Proxy b) path

-- Definition
(a :: String) :<|> (b :: UserId -> String) = toRoute (Proxy :: Proxy API) []

main :: IO ()
main = do print a -- "/user"
          print $ b (UserId 3) -- "/user/3"
```

Extending servant, JWT combinator

```haskell
instance HasServer api => HasServer (AuthToken :> api) where
  type ServerT (AuthToken :> api) m = UserId -> ServerT api m
  route Proxy subServer req@Request{..} resp =
    case getKey req of
      Nothing -> the401
      Just userid -> route (Proxy :: Proxy api) (subServer userid) req resp
   where
     the401 = resp . succeedWith $ responseLBS status401 [] "Invalid or missing Token"
     getKey :: Request -> Maybe UserId
     getKey Request{..} = do
       key <- lookup "X-Access-Token" requestHeaders
       sub <- JWT.sub . JWT.claims <$>
                 JWT.decodeAndVerifySignature (JWT.secret "secret") (T.decodeUtf8 key)
       fromText =<< JWT.stringOrURIToText <$> sub
```

```haskell
type TodoAPI = AuthToken :> "todo" :> Capture "id" TodoId :> Get '[JSON] (Maybe Todo)
todoGet :: UserId -> TodoId -> TodoApp (Maybe Todo)
```

Todo App

```
Web Tests
  Should create a user
  Should return 0 on initial todo count
  Should return an empty list with no todos
  Should create a todo
  Should update a todo
  Should delete a todo
  Should get a todo

Finished in 0.2142 seconds
7 examples, 0 failures
```

http://github.com/chicagohaskell/servant-presentation

Additional Content Types

- servant-blaze
- servant-lucid
- servant-ede
- servant-JuicyPixels

Coming soon

- Servant 0.5.0
  - faster routing
  - auth
  - no matrix-*
  - ExceptT

Questions