Servant

Describing Your API as a Type

Writing APIs

- Unpackaging Requests
- Parsing
- Handling Errors
- Repackaging Result
- Repeat Process

Fundamentals

- Define API as a **Type**
- Manipulate **Business** Types, not **Network** Types
- Don’t Repeat Yourself!
- Once API is defined, reuse structure!

Organization

- Endpoint Construction and Server Handlers
- More Endpoint Combinators
- Client Functions
- Authentication
- Documentation

Organization

- **Endpoint Construction and Server Handlers**
- More Endpoint Combinators
- Client Functions
- Authentication
- Documentation

Business Type

```
data User =
  { userName :: String
  , userEmail :: String
  , userAge :: Int }

user1 :: User
user1 = User "John Doe" "john.doe@gmail.com" 24

user2 :: User
user2 = User "Jane Doe" "jane.doe@gmail.com" 25
```

Business Type

```
userMap :: M.Map Int User
userMap = M.fromList [(1,user1), (2,user2)]

allUsers :: [User]
allUsers = M.elems userMap
```

All Users

```
GET /api/users :

[
  {
    "userName" : "John Doe",
    "userEmail" : "john.doe@gmail.com",
    "userAge" : 24
  },
  {
    "userName" : "Jane Doe",
    "userEmail" : "jane.doe@gmail.com",
    "userAge" : 25
  }
]
```

Single User

```
GET /api/users/1 :
{
  “userName” : “John Doe”,
  “userEmail” : “john.doe@gmail.com”,
  “userAge” : 24
}
```

Type Operators

- `:<|>`
  - Combine Endpoints
  - “e-plus”
- `:>`
  - Combine combinators
  - “c-plus”

Servant Endpoints

```
type UsersAPI = "api" :> "users" :> Get '[JSON] [User]
  :<|> "api" :> "users" :> Capture "userid" Int :> Get '[JSON] User
```

Servant Endpoints

```
type UsersAPI =
  "api" :> "users" :> Get '[JSON] [User]
    :<|>
  "api" :> "users" :> Capture "userid" Int :> Get '[JSON] User
```

Servant Endpoints

```
type UsersAPI = "api" :> "users" :> Get '[JSON] [User]
  :<|> "api" :> "users" :> Capture "userid" Int :> Get '[JSON] User
```

Servant Endpoints

```
-- GET /api/users/all
type UsersAPI = "api" :> "users" :> "all" :> Get '[JSON] [User]
```

Servant Endpoints

```
type UsersAPI = "api" :> "users" :> Get '[JSON] [User]
  :<|> "api" :> "users" :> Capture "userid" Int :> Get '[JSON] User
```

Servant Endpoints

```
type UsersAPI = "api" :> "users" :> Get '[JSON] [User]
  :<|> "api" :> "users" :> Capture "userid" Int :> Get '[JSON] User
```

Content Types

- Built-In
  - JSON
  - Plain Text
  - Octet Stream
  - Form URL Encoded
- Can make your own
  - HTML, like Lucid or Blaze

Refactoring Endpoints

```
type UsersAPI = "api" :> "users" :>
  ( Get '[JSON] [User]
   :<|> Capture "userid" Int :> GET '[JSON] User )
```

Handler Monad

- Handler Monad
  - `type Handler a = ExceptT ServantErr IO a`
- Can use your own monad

Handler Functions

```
allUsersHandler :: Handler [User]
```

```
singleUserHandler :: Int -> Handler User
```

Handler Functions

```
allUsersHandler :: Handler [User]
allUsersHandler = return allUsers

singleUserHandler :: Int -> Handler User
singleUserHandler uid = case M.lookup uid userMap of
  Nothing -> throwE $ err401 {errBody = "Could not find user with ID" }
  Just u -> return u
```

Serving the Server!

```
usersServer :: Server UsersAPI
usersServer = allUsersHandler :<|> singleUserHandler
```

Serving the Server!

```
usersServer :: Server UsersAPI
usersServer = allUsersHandler :<|> singleUserHandler

usersAPI :: Proxy UsersAPI
usersAPI = Proxy :: Proxy UsersAPI
```

Serving the Server!

```
usersServer :: Server UsersAPI
usersServer = allUsersHandler :<|> singleUserHandler

usersAPI :: Proxy UsersAPI
usersAPI = Proxy :: Proxy UsersAPI

main :: IO ()
main = run 8000 (serve usersAPI usersServer)
```

Organization

- Endpoint Construction and Server Handlers
- **More Endpoint Combinators**
- Client Functions
- Authentication
- Documentation

Adding Complexity

- More Endpoint Pieces
- Query Parameters
- Request Bodies
- Headers
- Etc.

Query Parameters

```
GET /api/users/filter_age?age_less_than=25
```

Query Parameters

```
GET /api/users/filter_age?age_less_than=25
```

```
"api" :> "users" :> "filter_age" :> QueryParam "age_less_than" Int :> Get '[JSON] [User]
```

Query Param Handlers

```
ageFilterHandler :: Maybe Int -> Handler [User]
ageFilterHandler Nothing = return allUsers
ageFilterHandler (Just maxAge) = return $ filter (\u -> userAge u < maxAge) allUsers
```

Parameter List

```
GET /api/users/filter_name?name=John&Jane
```

```
"api" :> "users" :> "filter_name" :> QueryParams "name" String :> Get '[JSON] [User]
```

Parameter List Handler

```
nameFilter :: [String] -> Handler [User]
nameFilter names = return $ filter filterByName allUsers
  where
    filterByName u = userFirstName u `elem` names
```

Parameter List

```
GET /api/users/filter_flag?is_old
```

```
"api" :> "users" :> "filter_flag" :> QueryFlag "is_old" :> Get '[JSON] [User]
```

Parameter Flag Handler

```
flagFilterHandler :: Bool -> Handler [User]
flagFilterHandler isOld = if isOld
  then return $ filter (\u -> userAge u > 24) allUsers
  else return allUsers
```

Request Body

```
PUT /api/users/:uid

"api" :> "users" :> Capture "userid" Int :> ReqBody '[JSON] User :> Put '[JSON] User
```

Request Body Handler

```
updateUserHandler :: Int -> User -> Handler User
updateUserHandler uid newUser = case M.lookup uid userMap of
  Nothing -> throwE $ err401 { errBody = "Couldn't find user." }
  Just _ -> do
    let newMap = M.insert uid newUser
    return newUser
```

Other Combinators

- Headers
  - `"api" :> "users" :> Header "api-token" Text :> Get '[JSON] [User]`
- Raw
  - `"api" :> "static" :> Raw`
- Authentication (later)
  - `"api" :> "users" :> BasicAuth "admin" User :> Get '[JSON] [User]`
  - `"api" :> "users" :> AuthProtect "admin" :> Get '[JSON] [User]`

Full API So Far

```
type UsersAPI = "api" :> "users" :>
  (Get '[JSON] [User]
  :<|> Capture "userid" Int :> Get '[JSON] User
  :<|> "filter_age" :> QueryParam "age_less_than" Int :> Get '[JSON] [User]
  :<|> "filter_name" :> QueryParams "name" String :> Get '[JSON] [User]
  :<|> "filter_flag" :> QueryFlag "is_old" :> Get '[JSON] [User]
  :<|> Capture "userid" Int :> ReqBody '[JSON] User :> Put '[JSON] User)
```

Organization

- Endpoint Construction and Server Handlers
- More Endpoint Combinators
- **Client Functions**
- Authentication
- Documentation

Client Functions

- Calling from our client
- Preferably with normal business types
- Don’t Repeat Yourself!
- “servant-client” library

Client Functions

```
import MyServer (UsersAPI, usersAPI)
Import Servant

...

allUsersClient :: ClientM [User]
singleUserClient :: Int -> ClientM User
ageFilterClient :: Maybe Int -> ClientM [User]
nameFilterClient :: [String] -> ClientM [User]
flagFilterClient :: Bool -> ClientM [User]
updateUserClient :: Int -> User -> ClientM User
```

Pattern Match

```
( allUsersClient :<|>
  singleUserClient :<|>
  ageFilterClient :<|>
  nameFilterClient :<|>
  flagFilterClient :<|>
  updateUserClient) = client usersAPI
```

ClientM Monad

```
env :: IO ClientEnv
env = do
  manager <- newManager tlsManagerSettings
  url <- parseBaseUrl "host=127.0.0.1 port=8080"
  return $ ClientEnv manager url
```

Calling Our API

```
fetchAllUsers :: IO ()
fetchAllUsers = do
  environment <- env
  result <- runClientM allUsersClient environment
  print result
```

Calling Our API

```
fetchAllUsers :: IO ()
fetchAllUsers = do
  environment <- env
  result <- runClientM allUsersClient environment
  print result

RESULT:
Right [User {userName = "John Doe", userEmail = "john.doe@gmail.com", userAge = 24},User
{userName = "Jane Doe", userEmail = "jane.doe@gmail.com", userAge = 25}]
```

Organization

- Endpoint Construction and Server Handlers
- More Endpoint Combinators
- Client Functions
- **Authentication**
- Documentation

But wait...

- No **Authentication** Yet!
- Basic Authentication
  - Just supply username/password on every request
  - Can also work with JWT
- Generalized Authentication
  - Allows arbitrary actions on authentication

BasicAuth Combinator

```
type UsersAPI = "api" :> "users" :>
  (BasicAuth "admin" User :> Get '[JSON] [User]

…

allUsersHandler :: User -> Handler [User]
allUsersHandler user = return allUsers
```

Auth Check

```
authCheck :: BasicAuthCheck User
authCheck = BasicAuthCheck check
  where
    check :: BasicAuthData -> IO AuthResult
    check (BasicAuthData username password) =
      if username == "James" && password == "admin"
        then return (Authorized $ User "James" "james@mondaymorninghaskell.com" 24)
        else return Unauthorized
```

Add Context and Serve!

```
authContext :: Context (BasicAuthCheck User ': '[])
authContext = authCheck .: EmptyContext

main :: IO ()
main = run 8000 (serveWithContext usersAPI authContext usersServer)
```

Client Side Basic Auth

```
allUsersClient :: BasicAuthData -> ClientM [User]

fetchAllUsers :: IO ()
fetchAllUsers = do
  environment <- env
  let authInfo = BasicAuthData "James" "admin"
  result <- runClientM (allUsersClient authInfo) environment
  print result
```

Client Side Basic Auth

```
allUsersClient :: BasicAuthData -> ClientM [User]

fetchAllUsers :: IO ()
fetchAllUsers = do
  environment <- env
  let authInfo = BasicAuthData “James” “admin”
  result <- runClientM (allUsersClient authInfo) environment
  print result

RESULT:
Right [User {userName = "John Doe", userEmail = "john.doe@gmail.com", userAge = 24},User
{userName = "Jane Doe", userEmail = "jane.doe@gmail.com", userAge = 25}]
```

Client Side Basic Auth

```
allUsersClient :: BasicAuthData -> ClientM [User]

fetchAllUsers :: IO ()
fetchAllUsers = do
  environment <- env
  let authInfo = BasicAuthData “Bad Name” “Bad Password”
  result <- runClientM (allUsersClient authInfo) environment
  print result

RESULT:
Left (FailureResponse {responseStatus = Status {statusCode = 403, statusMessage = "Forbidden"},
responseContentType = application/octet-stream, responseBody = ""})
```

Generalized Authentication

- Sometimes basic auth is not enough
- Request may need more information
- Server Response might be more complex

AuthProtect Combinator

```
type UsersAPI = "api" :> "users" :>
  (AuthProtect "admin" :> Get '[JSON] [User]
```

AuthServerData Type Family

```
type instance AuthServerData (AuthProtect “admin”) = Int

...

allUsersHandler :: Int -> Handler [User]
```

Generalized Handler

```
authHandler :: AuthHandler Request Int
authHandler = mkAuthHandler handler
  where
    handler request = ...
```

Context and Run!

```
authContext :: Context (AuthHandler Request Int ': '[])
authContext = authHandler :. EmptyContext

main :: IO ()
main = run 8000 (serveWithContext usersAPI authContext usersServer)
```

Client Side

```
type instance AuthClientData (AuthProtect “admin”) = JWToken

allUsersClient :: AuthenticateReq (AuthProtect “admin”) -> ClientM [User]
```

Inserting Token

```
addJWTHeader :: JWToken -> AuthenticateReq (AuthProtect "admin")
addJWTHeader jwt = mkAuthenticateReq jwt insertToken

insertToken :: JWToken -> Req -> Req
insertToken jwt req = req { headers = newPair : oldHeaders }
  where
    oldHeaders = headers req
    newPair = ("auth-token", jwt)
```

Calling the Client Function

```
fetchAllUsers :: IO ()
fetchAllUsers = do
  environment <- env
  result <- runClientM (allUsersClient (addJWTHeader "my authtoken")) environment
  print result
```

Organization

- Endpoint Construction and Server Handlers
- More Endpoint Combinators
- Client Functions
- Authentication
- **Documentation**

Generating Documentation

- Still Don’t Repeat Yourself!
- Have to add some business logic descriptions
- But otherwise use “docs” like “client”

Instances

- ToCapture
- ToParam
- ToSample

ToCapture Instance

```
instance ToCapture (Capture "userid" Int) where
  toCapture _ = DocCapture "userid" "The ID for the particular user"
```

ToParam Instance

```
instance ToParam (QueryParam "age_less_than" Int) where
  toParam _ = DocQueryParam
    "age_less_then" ["18", "24", "30"] "The upper bound of the age for returned users" Normal

instance ToParam (QueryParams "name" String) where
  toParam _ = DocQueryParam
    "name" ["John Doe", "Jane Doe"] "The names of users you are querying for." List

instance ToParam (QueryFlag "is_old") where
  toParam _ = DocQueryParam
    "is_old" [] "A flag filtering if the user is older than 24 years of age." Flag
```

ToSample Instance

```
instance ToSample User where
  toSamples _ = singleSample user1
```

Extending our API

```
type FullAPI = UsersAPI :<|> Raw

fullAPI :: Proxy FullAPI
fullAPI = Proxy :: Proxy FullAPI
```

Serving the Full API

```
docsBS :: ByteString
docsBS = encodeUtf8 . pack . markdown $ docs usersAPI

fullServer :: Server FullAPI
fullServer = usersServer :<|> serveDocs
  where
    plain = ("Content-Type", "text/plain")
    serveDocs _ response = response $ responseLBS ok200 [plain] docsBS

main :: IO ()
main = run 8000 (serve fullAPI fullServer)
```

Markdown Docs!

```
## GET /api/users

#### Response:

- Status code 200
- Headers: []

- Supported content types are:

    - `application/json`

...
```

Summary

- Make a type our of our API
- Construct Endpoints
- Write Handlers
- Write Client Functions
- Always use (:<|>)!
- Authentication
- Documentation

Extra Features

- Make Javascript client functions!
- Hook up your API with Reflex FRP!
- Use any documentation format!

Monday Morning Haskell

- Weekly Haskell Blog
- Mini Course on starting with Haskell
- Subscribe at:
  - mondaymorninghaskell.com/bayhac
- Get slides and code!

Acknowledgments

BayHac Organizers, esp. **Dan Burton, Erica Waichman**

TAKT™

Questions?