Type-Level Web APIs with Servant

An Exercise in Domain-Specific Generic Programming

WGP 2015, Vancouver

Alp Mestanogullari, Sönke Hahn, Julian K. Arni, Andres Löh

30 August 2015

A Servant web API

```
type Counter  =  Get '[JSON] Int
            :<|> "step" :> Post '[] ()
```

A Haskell datatype.

A Servant web API

```
type Counter   =   Get '[JSON] Int
               :<|> "step" :> Post '[] ()
```

```
GET   /      obtain the current value
POST  /step  increment counter
```

A Servant web API

```
type GetCounter  = Get '[JSON] Int
type StepCounter = "step" :> Post '[] ()
type Counter = GetCounter :<|> StepCounter
```

```
GET   /      obtain the current value
POST  /step  increment counter
```

A Servant web API

```haskell
type GetCounter  = Get '[JSON] CounterVal
type StepCounter = "step" :> Post '[] ()

type Counter = GetCounter :<|> StepCounter

newtype CounterVal = CounterVal Int
  deriving (Show, Num, FromJSON, ToJSON)
```

```
GET   /      obtain the current value
POST  /step  increment counter
```

What can we do with an API type?

Many things, but in particular:

- generate documentation,
- implement a server,
- obtain a client,
- describe links into the API.

What can we do with an API type?

Many things, but in particular:

- generate documentation,
- implement a server,
- obtain a client,
- describe links into the API.

The above are supported by Servant out of the box, but additional “interpretations” can be implemented.

Documentation

Generating documentation

A way to refer to the API type as a value:

```
counterAPI :: Proxy Counter
counterAPI = Proxy
```

Generating documentation

A way to refer to the API type as a value:

```
counterAPI :: Proxy Counter
counterAPI = Proxy
```

Generating Markdown documentation for the API:

```
counterDocs :: String
counterDocs = markdown (docs counterAPI)
```

Generating documentation

A way to refer to the API type as a value:

```
counterAPI :: Proxy Counter
counterAPI = Proxy
```

Generating Markdown documentation for the API:

```
counterDocs :: String
counterDocs = markdown (docs counterAPI)
```

```
markdown :: API -> String
docs     :: HasDocs api => Proxy api -> API
```

Fails at compile time because we lack necessary information!

Sample values

Input types are required to be an instance of `ToSample`:

```
instance ToSample CounterVal CounterVal where
  toSample _ = Just (CounterVal 42)
```

Documentation example

GET /

Response:

- Status code 200
- Headers: []
- Supported content types are:
  - `application/json`
- Response body as below.

```
42
```

Server

Implementing a server

What we can expect just by having the API type:

- decisions on whether a request is valid or not,
- routing the incoming requests to the right handlers.

What we have to provide ourselves:

- handlers that actually turn inputs into outputs.

The types of handlers

```
type GetCounter  = Get '[JSON] CounterVal
type StepCounter = "step" :> Post '[] ()
type Counter = GetCounter :<|> StepCounter
```

The types of handlers

```
type GetCounter  = Get '[JSON] CounterVal
type StepCounter = "step" :> Post '[] ()

type Counter = GetCounter :<|> StepCounter
```

Handler should be “something that produces a `CounterVal`”:

```
Server GetCounter ~ EitherT ServantErr IO CounterVal
```

The types of handlers

```
type GetCounter  = Get '[JSON] CounterVal
type StepCounter = "step" :> Post '[] ()

type Counter = GetCounter :<|> StepCounter
```

Handler should be “something that produces a `CounterVal`”:

```
Server GetCounter ~ EitherT ServantErr IO CounterVal
```

```
handleGetCounter :: TVar CounterVal
                 -> Server GetCounter
handleGetCounter ctr = liftIO (readTVarIO ctr)
```

The types of handlers

```haskell
type GetCounter  = Get '[JSON] CounterVal
type StepCounter = "step" :> Post '[] ()
type Counter = GetCounter :<|> StepCounter
```

Handler should be “something that produces a `()`”:

```haskell
Server StepCounter ~ EitherT ServantErr IO ()
```

```haskell
handleStepCounter :: TVar CounterVal
                  -> Server StepCounter
handleStepCounter ctr =
  liftIO $ atomically $ modifyTVar ctr (+ 1)
```

The types of handlers

```haskell
type GetCounter  = Get '[JSON] CounterVal
type StepCounter = "step" :> Post '[] ()
type Counter = GetCounter :<|> StepCounter
```

Handler should be a combination of two handlers:

```haskell
Server Counter ~
  (Server GetCounter :<|> Server StepCounter)
```

```haskell
handleCounter :: TVar CounterVal
              -> Server Counter
handleCounter ctr =     handleGetCounter  ctr
                  :<|> handleStepCounter ctr
```

Running the server

```
serve :: HasServer api
      => Proxy api -> Server api -> Application
```

Running the server

```
serve :: HasServer api
      => Proxy api -> Server api -> Application
```

```
start :: IO ()
start = do
  initCtr <- newTVarIO 0
  run 8000
    (serve counterAPI (handleCounter initCtr))
```

Running the server

```haskell
serve :: HasServer api
      => Proxy api -> Server api -> Application
```

```haskell
start :: IO ()
start = do
  initCtr <- newTVarIO 0
  run 8000
    (serve counterAPI (handleCounter initCtr))
```

Statically checks:

- the presence of handlers for every API part,
- the types of the handlers.

Complete server code

API with Proxy

```
type GetCounter  = Get '[JSON] CounterVal
type StepCounter = "step" :> Post '[] ()
type Counter = GetCounter :<|> StepCounter

newtype CounterVal = CounterVal Int
  deriving (Show, Num, FromJSON, ToJSON)

counterAPI :: Proxy Counter
counterAPI = Proxy
```

Handler(s)

```
handleGetCounter    ::  TVar CounterVal -> Server GetCounter
handleGetCounter ctr = liftIO (readTVarIO ctr)

handleStepCounter   ::  TVar CounterVal -> Server StepCounter
handleStepCounter ctr = liftIO $ atomically $ modifyTVar ctr (+ 1)

handleCounter       ::  TVar CounterVal -> Server Counter
handleCounter ctr    =  handleGetCounter  ctr
                   :<|> handleStepCounter ctr
```

Driver

```
start :: IO ()
start = do
  initCtr <- newTVarIO 0
  run 8000 (serve counterAPI (handleCounter initCtr))
```

Client

Obtaining a client

```
client :: HasClient api
       => Proxy api -> BaseUrl -> Client api
```

```
getCounter :<|> stepCounter =
  client counterAPI (BaseUrl Http "localhost" 8000)
```

Obtaining a client

```
client :: HasClient api
       => Proxy api -> BaseUrl -> Client api
```

```
getCounter :<|> stepCounter =
  client counterAPI (BaseUrl Http "localhost" 8000)
```

Yields:

```
getCounter  :: EitherT ServantError IO CounterVal
stepCounter :: EitherT ServantError IO ()
```

Interacting with the client

```
GHCi> runEitherT getCounter
Right (CounterVal 0)
GHCi> runEitherT (stepCounter >> getCounter)
Right (CounterVal 1)
```

Modifying the API

Making a change

```
type SetCounter  =  ReqBody '[JSON] CounterVal
                 :> Put '[] ()

type Counter   =   GetCounter
              :<|> StepCounter
              :<|> SetCounter
```

Making a change

```
type SetCounter  =  ReqBody '[JSON] CounterVal
                 :> Put '[] ()
type Counter   =   GetCounter
              :<|> StepCounter
              :<|> SetCounter
```

Server and client become type-incorrect.

Adapting the server

```
Server SetCounter ~
  CounterVal -> EitherT ServantErr IO ()
```

Adapting the server

```
Server SetCounter ~
  CounterVal -> EitherT ServantErr IO ()
```

```
handleSetCounter :: TVar CounterVal
                 -> Server SetCounter
handleSetCounter ctr newValue =
  liftIO $ atomically $ writeTVar ctr newValue
```

```
handleCounter :: TVar CounterVal -> Server Counter
handleCounter ctr   =   handleGetCounter  ctr
                  :<|> handleStepCounter ctr
                  :<|> handleSetCounter  ctr
```

Intermediate summary

Servant consists of:

- a type-level DSL for web APIs.
- a number of interpretations (documentation, server, client, links, ...) of the DSL.

Intermediate summary

Servant consists of:

- a type-level DSL for web APIs.
- a number of interpretations (documentation, server, client, links, ...) of the DSL.

Servant is extensible:

- with respect to the DSL constructs,
- and with respect to the interpretations.

Type-level DSL

What is a web API?

- Requests and responses.
- Requests can be constrained in various ways:
  - path,
  - parameters,
  - headers,
  - body,
  - method.
- Responses can also be constrained:
  - status code,
  - headers,
  - body.

The grammar

```
api ::= api :<|> api
     |  item :> api
     |  method
```

The grammar

```
api ::= api :<|> api
      | item :> api
      | method
```

```
item ::= symbol
       | header
       | ReqBody     ctypes type
       | Capture     symbol type
       | QueryFlag   symbol
       | QueryParam  symbol type
       | QueryParams symbol type
       | ...
```

The grammar

```
api ::= api :<|> api
      |  item :> api
      |  method
```

```
method ::= Get    ctypes rtype
         | Put    ctypes rtype
         | Post   ctypes rtype
         | Delete ctypes rtype
         | Patch  ctypes rtype
         | Raw
         | ...

ctypes  ::= '[ctype, ...]

ctype   ::= PlainText | JSON | HTML | ...
```

The grammar

```
api ::= api :<|> api
     |  item :> api
     |  method
```

Mapping to Haskell types:

```haskell
data api1 :<|> api2
infixr 8 :<|>

data (item :: k) :> api
infixr 9 :>

data ReqBody (ctypes :: [*]) (t :: *)
data Capture (symbol :: Symbol) (t :: *)

data Get (ctypes :: [*]) (t :: *)
data Post (ctypes :: [*]) (t :: *)

data JSON
```

Defining interpretations

Each interpretation is a class:

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

From the grammar to an interpretation

How to deal with the different syntactic categories?

From the grammar to an interpretation

How to deal with the different syntactic categories?

Option 1: use different classes for each category

```
class HasServerAPI api
class HasServerItem item
class HasServerMethod method
...
```

From the grammar to an interpretation

How to deal with the different syntactic categories?

Option 2: inline productions and do (almost) everything with api

```
api ::= api :<|> api
      | symbol :> api
      | Header symbol type :> api
      | ReqBody ctypes type :> api
      | ...
      | Get ctypes (Headers headers type)
      | Get ctypes type
      | ...
```

Structure of a typical interpretation

```
instance ... => HasServer (api1 :<|> api2)
instance ... => HasServer ((sym :: Symbol) :> api)
instance ... => HasServer (Header sym t :> api)
instance ... => HasServer (ReqBody ctype t)
...
instance ... => HasServer (Get ctypes (Headers hs t))
instance ... => HasServer (Get ctypes t)
```

Content-type handling

```
Get '[JSON, HTML, PlainText] MyType
```

Content-type handling

```
Get '[JSON, HTML, PlainText] MyType
```

```
class Accept ctype where
  contentType :: Proxy ctype -> MediaType
```

Every content-type descriptor is an instance of `Accept`.

Rendering and unrendering

```
class Accept ctype => MimeRender ctype t where
  mimeRender ::
    Proxy ctype -> t -> ByteString
class Accept ctype => MimeUnrender ctype t where
  mimeUnrender ::
    Proxy ctype -> ByteString -> Maybe t
```

Type-level computation is everywhere

In particular:

- the server interpretation computes the type of the handler,
- the client interpretation computes the types of the request functions,
- the link interpretation computes whether one API fragment is contained in another,
- the documentation interpretation uses a similar mechanism to attach additional documentation to specific parts of the API.

Example: the type of the handler

```
type Server (api1 :<|> api2) =
       Server api1 :<|> Server api2
type Server ((sym :: Symbol) :> api) =
       Server api
type Server (ReqBody ctypes t :> api) =
       t -> Server api
```

Extending the language

Adding a new interpretation:

- define a new class
- provide the instances for each of the syntactic constructs

Adding a new language construct:

- define a new datatype
- provide instances for all the interpretations

Current and future work

- Efficient routing.
- Error messages.
- Error handling.
- Authentication.
- CSV content type.
- Javascript client functions.
- Mock server generation.
- Swagger or other API description languages.
- …

Domain-specific generic programming

Datatype-generic programming

Aim: describe many or all datatypes.

Often uses structural combinators to describe the syntax of datatypes on the type-level:

- sums,
- products.

Generic functions interpret the structure and work for many datatypes and are robust to change.

Domain-specific generic programming

Aim: describe the syntax of a specific domain on the type-level.

Also uses structural combinators:

- sums (`:<|>`),
- products (`:>`).

Generic functions interpret the structure and work for many inhabitants of the domain and are robust to change.

Other ideas and examples

- database schemas,
- configuration options,
- general context-free grammars,
- …

Conclusion

By putting so much into the type safety, Servant programs

- save a lot of boilerplate,
- are much more robust, concise, and easy to refactor.

Conclusion

By putting so much into the type safety, Servant programs

- save a lot of boilerplate,
- are much more robust, concise, and easy to refactor.

Haskell (with all its current extensions) is mostly up to the task.
Essential:

- type classes,
- type families,
- data kinds, type-level strings, kind polymorphism.

Conclusion

By putting so much into the type safety, Servant programs

- save a lot of boilerplate,
- are much more robust, concise, and easy to refactor.

Haskell (with all its current extensions) is mostly up to the task.
Essential:

- type classes,
- type families,
- data kinds, type-level strings, kind polymorphism.

Domain-specific generic programming can be applied to other domains as well.

https://haskell-servant.github.io

Julian and Sönke offer a CUFP tutorial on Friday.