Servant - Typesafe DSL for describing & deriving Servers and Clients

Servant is a type-level DSL written in the functional programming language Haskell. It has a focus on encoding as much as possible about each endpoint of a server on the type-level, thus enabling automatic derivation of server-side handler type signatures and – more impressively – fully functional client-side accessors.

Because your API is statically known, it is also incredibly simple to generate documentation for it – be it a custom layout, or via a defined standard like Swagger/OpenAPI.

At the same time, Servant is designed to be incredibly extensible and pluggable. Almost all aspects of its system can be extended ad-hoc – a new return type? Define (or more realistically, derive a new typeclass instance) so it can convert your value and send it in a Response.

Want to require a whole section of your server require authentication? No need to declare that separately from the API definition itself – just include the combinator in the API. Not content with the existing Auth combinator not being able to verify login tokens via your database? Just define a new combinator that sits on top and that queries your database, deciding to terminate the routing process before it even reaches your handler.

A good start to Servant is given through its "Read the Docs" page at https://docs.servant.dev/en/stable/tutorial/index.html. The Tutorial gives a nice overview over the core batteries-included functionalities in Servant, while the Cookbook takes a more broad look at the ecosystem of available pluggable libraries.

As mentioned, Servant is written in Haskell, a pure, lazy, functional programming language. If one is mainly acquainted with languages like C++, Java, C#, Python, JavaScript and the like, Haskell might provide an interesting challenge to tackle and to remold how one thinks about programming. As a pure language, Haskell does not directly offer any mutation – all is handled via constructing updated copies of basic datastructures. To enable efficiency while also encapsulating the side-effects of mutability, or interaction with outside concepts like the filesystem, such operations are constrained into a specific context called `IO`. But it is possible to define more specific "effects" as well – while `IO` is a blunt hammer that just screams "I'm Impure!", `State` can be used to pass along a mutable state in your application, `Except` can model throwing and catching exceptions, while `STM` – short for "Software Transactional Memory" allows for deadlock- and race condition-free concurrent programming.

Servant itself only concerns the design and implementation of your servers API – what URLs with which request bodies, headers, authentications etc. can the server handle? If you need to use a database – and most realistically, you do – you simply choose a library for it, construct the connection at server startup and then make it available by tweaking the type of your handlers once – making the connection available as a `Reader(T)` context. From then on, you can simply use the connection with the libraries functions as if they were a built-in concept. One popular such library is `persistent`, which also functions as an ORM both to SQL and NoSQL-databases.

And just like Servant, it is completely typesafe. For more advanced queries, the SQL-specific library `esqueleto` offers a DSL for writing queries.

And last but not least, the obligatory small example: A small counter server keeping track of the last modification.

```haskell
{-# LANGUAGE DataKinds, DeriveGeneric, TypeOperators, BlockArguments #-}
{-# LANGUAGE DeriveAnyClass, RecordWildCards, DerivingStrategies #-}
module Main where

import Control.Monad.IO.Class
import Data.Aeson
import Data.IORef
import Data.Proxy
import GHC.Generics
import Network.Wai.Handler.Warp
import Servant.API
import Servant.Server

data Counter = Counter { count :: Int, lastModification :: Int }
  deriving stock (Eq, Show, Generic)
  deriving anyclass (FromJSON, ToJSON)

type API =
  -- GET /count, returns { "count": 20, "lastModification": 0 }
  "count" :> Get '[JSON] Counter
  -- POST /add with Int as JSON, returns new Counter
  :<|> "add" :> ReqBody '[JSON] Int :> Post '[JSON] Counter
  -- POST /subtract with Int as JSON, returns nothing
  :<|> "subtract" :> ReqBody '[JSON] Int :> NoContentVerb 'POST

getCount :: IORef Counter -> Handler Counter
getCount ref = liftIO $ readIORef ref

addCount :: IORef Counter -> Int -> Handler Counter
addCount ref i = do
  Counter {..} <- liftIO $ readIORef ref
  let counter = Counter { count = count + i, lastModification = i }
  liftIO $ writeIORef ref counter
  return counter

subCount :: IORef Counter -> Int -> Handler NoContent
subCount ref i = do
  liftIO $ modifyIORef ref \Counter {..} -> Counter {count = count-i, lastModification = (-i)}
  return NoContent

server :: IORef Counter -> Server API
server ref = getCount ref :<|> addCount ref :<|> subCount ref

main = do
  ref <- newIORef $ Counter 0 0
  run 8080 $ serve (Proxy :: Proxy API) $ server ref
```