servant Documentation

Release

Servant Contributors

July 05, 2018

Contents

servant is a set of Haskell libraries for writing *type-safe* web applications but also *deriving* clients (in Haskell and other languages) or generating documentation for them, and more.

This is achieved by taking as input a description of the web API as a Haskell type. Servant is then able to check that your server-side request handlers indeed implement your web API faithfully, or to automatically derive Haskell functions that can hit a web application that implements this API, generate a Swagger description or code for client functions in some other languages directly.

If you would like to learn more, click the tutorial link below.

CHAPTER 1

Tutorial

This is an introductory tutorial to **servant**. Whilst browsing is fine, it makes more sense if you read the sections in order, or at least read the first section before anything else.

Any comments, issues or feedback about the tutorial can be submitted to servant's issue tracker.

In fact, the whole tutorial is a cabal project and can be built and played with locally as follows:

```
$ git clone https://github.com/haskell-servant/servant.git
$ cd servant
# build
$ cabal new-build tutorial
# load in ghci to play with it
$ cabal new-repl tutorial
```

The code can be found in the **.lhs* files under *doc/tutorial/* in the repository. Feel free to edit it while you're reading this documentation and see the effect of your changes.

Nix users should feel free to take a look at the *nix/shell.nix* file in the repository and use it to provision a suitable environment to build and run the examples.

A web API as a type

The source for this tutorial section is a literate haskell file, so first we need to have some language extensions and imports:

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

module ApiType where

import Data.Text
import Data.Time (UTCTime)
import Servant.API
```

Consider the following informal specification of an API:

> The endpoint at `/users` expects a GET request with query string parameter `sortby` whose value can be one of `age` or `name` and returns a list/array of JSON objects describing users, with fields `age`, `name`, `email`, `registration_date`".

You *should* be able to formalize that. And then use the formalized version to get you much of the way towards writing a web app. And all the way towards getting some client libraries, and documentation, and more.

How would we describe it with **servant**? An endpoint description is a good old Haskell **type**:

```
type UserAPI = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]

data SortBy = Age | Name

data User = User {
  name :: String,
  age :: Int,
  email :: String,
  registration_date :: UTCTime
}
```

Let's break that down:

- `"users"` says that our endpoint will be accessible under `/users`;
- `QueryParam "sortby" SortBy`, where `SortBy` is defined by `data SortBy = Age | Name`, says that the endpoint has a query string parameter named `sortby` whose value will be extracted as a value of type `SortBy`.
- `Get '[JSON] [User]` says that the endpoint will be accessible through HTTP GET requests, returning a list of users encoded as JSON. You will see later how you can make use of this to make your data available under different formats, the choice being made depending on the Accept header specified in the client's request.
- The `:>` operator that separates the various "combinators" just lets you sequence static path fragments, URL captures and other combinators. The ordering only matters for static path fragments and URL captures. `"users" :> "list-all" :> Get '[JSON] [User]`, equivalent to `/users/list-all`, is obviously not the same as `"list-all" :> "users" :> Get '[JSON] [User]`, which is equivalent to `/list-all/users`. This means that sometimes `:>` is somehow equivalent to `/`, but sometimes it just lets you chain another combinator.

Tip: If your endpoint responds to `/` (the root path), just omit any combinators that introduce path segments. E.g. the following api has only one endpoint on `/`:

```
type RootEndpoint =
  Get '[JSON] User
```

We can also describe APIs with multiple endpoints by using the `:<|>` combinators. Here's an example:

```
type UserAPI2 = "users" :> "list-all" :> Get '[JSON] [User]
           :<|> "list-all" :> "users" :> Get '[JSON] [User]
```

servant provides a fair amount of combinators out-of-the-box, but you can always write your own when you need it. Here's a quick overview of the most often needed combinators that **servant** comes with.

Combinators

Static strings

As you've already seen, you can use type-level strings (enabled with the `DataKinds` language extension) for static path fragments. Chaining them amounts to `/`-separating them in a URL.

```
type UserAPI3 = "users" :> "list-all" :> "now" :> Get '[JSON] [User]
              -- describes an endpoint reachable at:
              -- /users/list-all/now
```

`Delete`, `Get`, `Patch`, `Post` and `Put`

The `Get` combinator is defined in terms of the more general `Verb`:

```
data Verb method (statusCode :: Nat) (contentType :: [*]) a
type Get = Verb 'GET 200
```

There are other predefined type synonyms for other common HTTP methods, such as e.g.:

```
type Delete = Verb 'DELETE 200
type Patch  = Verb 'PATCH 200
type Post   = Verb 'POST 200
type Put    = Verb 'PUT 200
```

There are also variants that do not return a 200 status code, such as for example:

```
type PostCreated  = Verb 'POST 201
type PostAccepted = Verb 'POST 202
```

An endpoint always ends with a variant of the `Verb` combinator (unless you write your own combinators). Examples:

```
type UserAPI4 = "users" :> Get '[JSON] [User]
           :<|> "admins" :> Get '[JSON] [User]
```

`StreamGet` and `StreamPost`

The `StreamGet` and `StreamPost` combinators are defined in terms of the more general `Stream`

```
data Stream (method :: k1) (framing :: *) (contentType :: *) a
type StreamGet  = Stream 'GET
type StreamPost = Stream 'POST
```

These describe endpoints that return a stream of values rather than just a single value. They not only take a single content type as a parameter, but also a framing strategy – this specifies how the individual results are delineated from one another in the stream. The three standard strategies given with Servant are `NewlineFraming`, `NetstringFraming` and `NoFraming`, but others can be written to match other protocols.

`Capture`

URL captures are segments of the path of a URL that are variable and whose actual value is captured and passed to the request handlers. In many web frameworks, you'll see it written as in `/users/:userid`, with that leading `:` denoting that `userid` is just some kind of variable name or placeholder. For instance, if `userid` is supposed to range over all integers greater or equal to 1, our endpoint will match requests made to `/users/1`, `/users/143` and so on.

The `Capture` combinator in **servant** takes a (type-level) string representing the "name of the variable" and a type, which indicates the type we want to decode the "captured value" to.

```
data Capture (s :: Symbol) a
-- s :: Symbol just says that 's' must be a type-level string.
```

In some web frameworks, you use regexes for captures. We use a `FromHttpApiData` class, which the captured value must be an instance of.

Examples:

```
type UserAPI5 = "user" :> Capture "userid" Integer :> Get '[JSON] User
                -- equivalent to 'GET /user/:userid'
                -- except that we explicitly say that "userid"
                -- must be an integer

           :<|> "user" :> Capture "userid" Integer :> DeleteNoContent '[JSON] NoContent
                -- equivalent to 'DELETE /user/:userid'
```

In the second case, `DeleteNoContent` specifies a 204 response code, `JSON` specifies the content types on which the handler will match, and `NoContent` says that the response will always be empty.

QueryParam, QueryParams, QueryFlag

`QueryParam`, `QueryParams` and `QueryFlag` are about parameters in the query string, i.e., those parameters that come after the question mark (`?`) in URLs, like `sortby` in `/users?sortby=age`, whose value is set to `age`. `QueryParams` lets you specify that the query parameter is actually a list of values, which can be specified using `?param=value1&param=value2`. This represents a list of values composed of `value1` and `value2`. `QueryFlag` lets you specify a boolean-like query parameter where a client isn't forced to specify a value. The absence or presence of the parameter's name in the query string determines whether the parameter is considered to have the value `True` or `False`. For instance, `/users?active` would list only active users whereas `/users` would list them all.

Here are the corresponding data type declarations:

```
data QueryParam (sym :: Symbol) a
data QueryParams (sym :: Symbol) a
data QueryFlag (sym :: Symbol)
```

Examples:

```
type UserAPI6 = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]
                -- equivalent to 'GET /users?sortby={age, name}'
```

Again, your handlers don't have to deserialize these things (into, for example, a `SortBy`). **servant** takes care of it.

ReqBody

Each HTTP request can carry some additional data that the server can use in its *body*, and this data can be encoded in any format – as long as the server understands it. This can be used for example for an endpoint for creating new users: instead of passing each field of the user as a separate query string parameter or something dirty like that, we can group all the data into a JSON object. This has the advantage of supporting nested objects.

servant's `ReqBody` combinator takes a list of content types in which the data encoded in the request body can be represented and the type of that data. And, as you might have guessed, you don't have to check the content type header, and do the deserialization yourself. We do it for you. And return `Bad Request` or `Unsupported Content Type` as appropriate.

Here's the data type declaration for it:

```
data ReqBody (contentTypes :: [*]) a
```

Examples:

```
type UserAPI7 = "users" :> ReqBody '[JSON] User :> Post '[JSON] User
                -- - equivalent to 'POST /users' with a JSON object
                --   describing a User in the request body
                -- - returns a User encoded in JSON
```

```
          :<|> "users" :> Capture "userid" Integer
                       :> ReqBody '[JSON] User
                       :> Put '[JSON] User
               -- - equivalent to 'PUT /users/:userid' with a JSON
               --   object describing a User in the request body
               -- - returns a User encoded in JSON
```

Request `Headers`

Request headers are used for various purposes, from caching to carrying auth-related data. They consist of a header name and an associated value. An example would be `Accept: application/json`.

The `Header` combinator in **servant** takes a type-level string for the header name and the type to which we want to decode the header's value (from some textual representation), as illustrated below:

```
data Header (sym :: Symbol) a
```

Here's an example where we declare that an endpoint makes use of the `User-Agent` header which specifies the name of the software/library used by the client to send the request.

```
type UserAPI8 = "users" :> Header "User-Agent" Text :> Get '[JSON] [User]
```

Content types

So far, whenever we have used a combinator that carries a list of content types, we've always specified `'[JSON]`. However, **servant** lets you use several content types, and also lets you define your own content types.

Four content types are provided out-of-the-box by the core **servant** package: `JSON`, `PlainText`, `FormUrlEncoded` and `OctetStream`. If for some obscure reason you wanted one of your endpoints to make your user data available under those 4 formats, you would write the API type as below:

```
type UserAPI9 = "users" :> Get '[JSON, PlainText, FormUrlEncoded, OctetStream] [User]
```

(There are other packages that provide other content types. For example **servant-lucid** and **servant-blaze** allow to generate html pages (using **lucid** and **blaze-html**) and both come with a content type for html.)

We will further explain how these content types and your data types can play together in the section about serving an API.

Response `Headers`

Just like an HTTP request, the response generated by a webserver can carry headers too. **servant** provides a `Headers` combinator that carries a list of `Header` types and can be used by simply wrapping the "return type" of an endpoint with it.

```
data Headers (ls :: [*]) a
```

If you want to describe an endpoint that returns a "User-Count" header in each response, you could write it as below:

```
type UserAPI10 = "users" :> Get '[JSON] (Headers '[Header "User-Count" Integer] [User])
```

Basic Authentication

Once you've established the basic routes and semantics of your API, it's time to consider protecting parts of it. Authentication and authorization are broad and nuanced topics; as servant began to explore this space we started small with one of HTTP's earliest authentication schemes: Basic Authentication.

When protecting endpoints with basic authentication, we need to specify two items:

1. The **realm** of authentication as per the Basic Authentication spec.
2. The datatype returned by the server after authentication is verified. This is usually a `User` or `Customer` type datatype.

With those two items in mind, *servant* provides the following combinator:

```
data BasicAuth (realm :: Symbol) (userData :: *)
```

Which is used like so:

```
type ProtectedAPI11
     = UserAPI                                  -- this is public
 :<|> BasicAuth "my-realm" User :> UserAPI2 -- this is protected by auth
```

Empty APIs

Sometimes it is useful to be able to generalise an API over the type of some part of it:

```
type UserAPI12 innerAPI
     = UserAPI              -- this is the fixed bit of the API
 :<|> "inner" :> innerAPI   -- this lets us put various other APIs under /inner
```

If there is a case where you do not have anything extra to serve, you can use the `EmptyAPI` combinator to indicate this:

```
type UserAPI12Alone = UserAPI12 EmptyAPI
```

This also works well as a placeholder for unfinished parts of an API while it is under development, for when you know that there should be *something* there but you don't yet know what. Think of it as similar to the unit type `()`.

Interoperability with `wai`: `Raw`

Finally, we also include a combinator named `Raw` that provides an escape hatch to the underlying low-level web library `wai`. It can be used when you want to plug a wai `Application` into your webservice:

```
type UserAPI13 = "users" :> Get '[JSON] [User]
                 -- a /users endpoint

            :<|> Raw
                 -- requests to anything else than /users
                 -- go here, where the server will try to
                 -- find a file with the right name
                 -- at the right path
```

One example for this is if you want to serve a directory of static files along with the rest of your API. But you can plug in everything that is an `Application`, e.g. a whole web application written in any of the web frameworks that support `wai`.

Serving an API

Enough chit-chat about type-level combinators and representing an API as a type. Can we have a webservice already?

A first example

Equipped with some basic knowledge about the way we represent APIs, let's now write our first webservice.

The source for this tutorial section is a literate haskell file, so first we need to have some language extensions and imports:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeOperators #-}

module Server where

import Prelude ()
import Prelude.Compat

import Control.Monad.Except
import Control.Monad.Reader
import Data.Aeson.Compat
import Data.Aeson.Types
import Data.Attoparsec.ByteString
import Data.ByteString (ByteString)
import Data.List
import Data.Maybe
import Data.String.Conversions
import Data.Time.Calendar
import GHC.Generics
import Lucid
import Network.HTTP.Media ((//), (/:))
import Network.Wai
import Network.Wai.Handler.Warp
import Servant
import System.Directory
import Text.Blaze
import Text.Blaze.Html.Renderer.Utf8
import qualified Data.Aeson.Parser
import qualified Text.Blaze.Html
```

Important: the `Servant` module comes from the **servant-server** package, the one that lets us run webservers that implement a particular API type. It reexports all the types from the **servant** package that let you declare API types as well as everything you need to turn your request handlers into a fully-fledged webserver. This means that in your applications, you can just add **servant-server** as a dependency, import `Servant` and not worry about anything else.

We will write a server that will serve the following API.

```
type UserAPI1 = "users" :> Get '[JSON] [User]
```

Here's what we would like to see when making a GET request to `/users`.

```
[ {"name": "Isaac Newton", "age": 372, "email": "isaac@newton.co.uk", "registration_date": "1683-03-0
, {"name": "Albert Einstein", "age": 136, "email": "ae@mc2.org", "registration_date": "1905-12-01"}
]
```

Now let's define our `User` data type and write some instances for it.

```
data User = User
  { name :: String
  , age :: Int
  , email :: String
  , registration_date :: Day
  } deriving (Eq, Show, Generic)

instance ToJSON User
```

Nothing funny going on here. But we now can define our list of two users.

```
users1 :: [User]
users1 =
  [ User "Isaac Newton"    372 "isaac@newton.co.uk" (fromGregorian 1683  3 1)
  , User "Albert Einstein" 136 "ae@mc2.org"         (fromGregorian 1905 12 1)
  ]
```

We can now take care of writing the actual webservice that will handle requests to such an API. This one will be very simple, being reduced to just a single endpoint. The type of the web application is determined by the API type, through a *type family* named `Server`. (Type families are just functions that take types as input and return types.) The `Server` type family will compute the right type that a bunch of request handlers should have just from the corresponding API type.

The first thing to know about the `Server` type family is that behind the scenes it will drive the routing, letting you focus only on the business logic. The second thing to know is that for each endpoint, your handlers will by default run in the `Handler` monad. This is overridable very easily, as explained near the end of this guide. Third thing, the type of the value returned in that monad must be the same as the second argument of the HTTP method combinator used for the corresponding endpoint. In our case, it means we must provide a handler of type `Handler [User]`. Well, we have a monad, let's just `return` our list:

```
server1 :: Server UserAPI1
server1 = return users1
```

That's it. Now we can turn `server` into an actual webserver using wai and warp:

```
userAPI :: Proxy UserAPI1
userAPI = Proxy

-- 'serve' comes from servant and hands you a WAI Application,
-- which you can think of as an "abstract" web application,
-- not yet a webserver.
app1 :: Application
app1 = serve userAPI server1
```

The `userAPI` bit is, alas, boilerplate (we need it to guide type inference). But that's about as much boilerplate as you get.

And we're done! Let's run our webservice on the port 8081.

```
main :: IO ()
main = run 8081 app1
```

You can put this all into a file or just grab servant's repo and look at the *doc/tutorial* directory. This code (the source of this web page) is in *doc/tutorial/Server.lhs*.

If you run it, you can go to `http://localhost:8081/users` in your browser or query it with curl and you see:

```
$ curl http://localhost:8081/users
[{"email":"isaac@newton.co.uk","registration_date":"1683-03-01","age":372,"name":"Isaac Newton"},{"er
```

More endpoints

What if we want more than one endpoint? Let's add `/albert` and `/isaac` to view the corresponding users encoded in JSON.

```
type UserAPI2 = "users" :> Get '[JSON] [User]
           :<|> "albert" :> Get '[JSON] User
           :<|> "isaac" :> Get '[JSON] User
```

And let's adapt our code a bit.

```
isaac :: User
isaac = User "Isaac Newton" 372 "isaac@newton.co.uk" (fromGregorian 1683 3 1)

albert :: User
albert = User "Albert Einstein" 136 "ae@mc2.org" (fromGregorian 1905 12 1)

users2 :: [User]
users2 = [isaac, albert]
```

Now, just like we separate the various endpoints in `UserAPI` with `:<|>`, we are going to separate the handlers with `:<|>` too! They must be provided in the same order as in in the API type.

```
server2 :: Server UserAPI2
server2 = return users2
     :<|> return albert
     :<|> return isaac
```

And that's it! You can run this example in the same way that we showed for `server1` and check out the data available at `/users`, `/albert` and `/isaac`.

From combinators to handler arguments

Fine, we can write trivial webservices easily, but none of the two above use any "fancy" combinator from servant. Let's address this and use `QueryParam`, `Capture` and `ReqBody` right away. You'll see how each occurrence of these combinators in an endpoint makes the corresponding handler receive an argument of the appropriate type automatically. You don't have to worry about manually looking up URL captures or query string parameters, or decoding/encoding data from/to JSON. Never.

We are going to use the following data types and functions to implement a server for `API`.

```
type API = "position" :> Capture "x" Int :> Capture "y" Int :> Get '[JSON] Position
      :<|> "hello" :> QueryParam "name" String :> Get '[JSON] HelloMessage
      :<|> "marketing" :> ReqBody '[JSON] ClientInfo :> Post '[JSON] Email

data Position = Position
  { xCoord :: Int
  , yCoord :: Int
  } deriving Generic

instance ToJSON Position
```

```
newtype HelloMessage = HelloMessage { msg :: String }
  deriving Generic

instance ToJSON HelloMessage

data ClientInfo = ClientInfo
  { clientName :: String
  , clientEmail :: String
  , clientAge :: Int
  , clientInterestedIn :: [String]
  } deriving Generic

instance FromJSON ClientInfo
instance ToJSON ClientInfo

data Email = Email
  { from :: String
  , to :: String
  , subject :: String
  , body :: String
  } deriving Generic

instance ToJSON Email

emailForClient :: ClientInfo -> Email
emailForClient c = Email from' to' subject' body'

  where from'    = "great@company.com"
        to'      = clientEmail c
        subject' = "Hey " ++ clientName c ++ ", we miss you!"
        body'    = "Hi " ++ clientName c ++ ",\n\n"
                 ++ "Since you've recently turned " ++ show (clientAge c)
                 ++ ", have you checked out our latest "
                 ++ intercalate ", " (clientInterestedIn c)
                 ++ " products? Give us a visit!"
```

We can implement handlers for the three endpoints:

```
server3 :: Server API
server3 = position
     :<|> hello
     :<|> marketing

  where position :: Int -> Int -> Handler Position
        position x y = return (Position x y)

        hello :: Maybe String -> Handler HelloMessage
        hello mname = return . HelloMessage $ case mname of
          Nothing -> "Hello, anonymous coward"
          Just n  -> "Hello, " ++ n

        marketing :: ClientInfo -> Handler Email
        marketing clientinfo = return (emailForClient clientinfo)
```

Did you see that? The types for your handlers changed to be just what we needed! In particular:

- a `Capture "something" a` becomes an argument of type `a` (for `position`);
- a `QueryParam "something" a` becomes an argument of type `Maybe a` (because an endpoint can tech-

nically be accessed without specifying any query string parameter, we decided to "force" handlers to be aware that the parameter might not always be there);

- a `ReqBody contentTypeList a` becomes an argument of type `a`;

And that's it. Here's the example in action:

```
$ curl http://localhost:8081/position/1/2
{"xCoord":1,"yCoord":2}
$ curl http://localhost:8081/hello
{"msg":"Hello, anonymous coward"}
$ curl http://localhost:8081/hello?name=Alp
{"msg":"Hello, Alp"}
$ curl -X POST -d '{"clientName":"Alp Mestanogullari", "clientEmail" : "alp@foo.com", "clientAge": 25
{"subject":"Hey Alp Mestanogullari, we miss you!","body":"Hi Alp Mestanogullari,\n\nSince you've rece
```

For reference, here's a list of some combinators from **servant**:

- `Delete`, `Get`, `Patch`, `Post`, `Put`: these do not become arguments. They provide the return type of handlers, which usually is `Handler <something>`.
- `Capture "something" a` becomes an argument of type `a`.
- `QueryParam "something" a`, `Header "something" a` all become arguments of type `Maybe a`, because there might be no value at all specified by the client for these.
- `QueryFlag "something"` gets turned into an argument of type `Bool`.
- `QueryParams "something" a` gets turned into an argument of type `[a]`.
- `ReqBody contentTypes a` gets turned into an argument of type `a`.

The `FromHttpApiData`/`ToHttpApiData` classes

Wait... How does **servant** know how to decode the `Int`s from the URL? Or how to decode a `ClientInfo` value from the request body? This is what this and the following two sections address.

`Captures` and `QueryParams` are represented by some textual value in URLs. `Headers` are similarly represented by a pair of a header name and a corresponding (textual) value in the request's "metadata". How types are decoded from headers, captures, and query params is expressed in a class `FromHttpApiData` (from the package **http-api-data**):

```
class FromHttpApiData a where
  {-# MINIMAL parseUrlPiece | parseQueryParam #-}
  -- | Parse URL path piece.
  parseUrlPiece :: Text -> Either Text a
  parseUrlPiece = parseQueryParam

  -- | Parse HTTP header value.
  parseHeader :: ByteString -> Either Text a
  parseHeader = parseUrlPiece . decodeUtf8

  -- | Parse query param value.
  parseQueryParam :: Text -> Either Text a
  parseQueryParam = parseUrlPiece
```

As you can see, as long as you provide either `parseUrlPiece` (for `Captures`) or `parseQueryParam` (for `QueryParams`), the other methods will be defined in terms of this.

http-api-data provides a decent number of instances, helpers for defining new ones, and wonderful documentation.

There's not much else to say about these classes. You will need instances for them when using `Capture`, `QueryParam`, `QueryParams`, and `Header` with your types. You will need `FromHttpApiData` instances for server-side request handlers and `ToHttpApiData` instances only when using **servant-client**, as described in the section about deriving haskell functions to query an API.

Using content-types with your data types

The same principle was operating when decoding request bodies from JSON, and responses *into* JSON. (JSON is just the running example - you can do this with any content-type.)

This section introduces a couple of typeclasses provided by **servant** that make all of this work.

The truth behind `JSON`

What exactly is `JSON` (the type as used in `Get '[JSON] User`)? Like the 3 other content-types provided out of the box by **servant**, it's a really dumb data type.

```
data JSON
data PlainText
data FormUrlEncoded
data OctetStream
```

Obviously, this is not all there is to `JSON`, otherwise it would be quite pointless. Like most of the data types in **servant**, `JSON` is mostly there as a special *symbol* that's associated with encoding (resp. decoding) to (resp. from) the *JSON* format. The way this association is performed can be decomposed into two steps.

The first step is to provide a proper `MediaType` (from **http-media**) representation for `JSON`, or for your own content-types. If you look at the haddocks from this link, you can see that we just have to specify `application/json` using the appropriate functions. In our case, we can just use `(//) :: ByteString -> ByteString -> MediaType`. The precise way to specify the `MediaType` is to write an instance for the `Accept` class:

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

instance Accept JSON where
    contentType _ = "application" // "json"
```

The second step is centered around the `MimeRender` and `MimeUnrender` classes. These classes just let you specify a way to encode and decode values into or from your content-type's representation.

```
class Accept ctype => MimeRender ctype a where
    mimeRender :: Proxy ctype -> a -> ByteString
    -- alternatively readable as:
    mimeRender :: Proxy ctype -> (a -> ByteString)
```

Given a content-type and some user type, `MimeRender` provides a function that encodes values of type `a` to lazy `ByteStrings`.

In the case of `JSON`, this is easily dealt with! For any type `a` with a `ToJSON` instance, we can render values of that type to JSON using `Data.Aeson.encode`.

```
instance ToJSON a => MimeRender JSON a where
  mimeRender _ = encode
```

And now the `MimeUnrender` class, which lets us extract values from lazy `ByteStrings`, alternatively failing with an error string.

```
class Accept ctype => MimeUnrender ctype a where
    mimeUnrender :: Proxy ctype -> ByteString -> Either String a
```

We don't have much work to do there either, `Data.Aeson.eitherDecode` is precisely what we need. However, it only allows arrays and objects as toplevel JSON values and this has proven to get in our way more than help us so we wrote our own little function around **aeson** and **attoparsec** that allows any type of JSON value at the toplevel of a "JSON document". Here's the definition in case you are curious.

```
eitherDecodeLenient :: FromJSON a => ByteString -> Either String a
eitherDecodeLenient input = do
    v :: Value <- parseOnly (Data.Aeson.Parser.value <* endOfInput) (cs input)
    parseEither parseJSON v
```

This function is exactly what we need for our `MimeUnrender` instance.

```
instance FromJSON a => MimeUnrender JSON a where
    mimeUnrender _ = eitherDecodeLenient
```

And this is all the code that lets you use `JSON` with `ReqBody`, `Get`, `Post` and friends. We can check our understanding by implementing support for an `HTML` content-type, so that users of your webservice can access an HTML representation of the data they want, ready to be included in any HTML document, e.g. using jQuery's `load` function, simply by adding `Accept: text/html` to their request headers.

Case-studies: servant-blaze and servant-lucid

These days, most of the haskellers who write their HTML UIs directly from Haskell use either **blaze-html** or **lucid**. The best option for **servant** is obviously to support both (and hopefully other templating solutions!). We're first going to look at **lucid**:

```
data HTMLLucid
```

Once again, the data type is just there as a symbol for the encoding/decoding functions, except that this time we will only worry about encoding since **lucid** doesn't provide a way to extract data from HTML.

```
instance Accept HTMLLucid where
    contentType _ = "text" // "html" /: ("charset", "utf-8")
```

Note that this instance uses the `(/:)` operator from **http-media** which lets us specify additional information about a content-type, like the charset here.

The rendering instances call similar functions that take types with an appropriate instance to an "abstract" HTML representation and then write that to a `ByteString`.

```
instance ToHtml a => MimeRender HTMLLucid a where
    mimeRender _ = renderBS . toHtml

-- let's also provide an instance for lucid's
-- 'Html' wrapper.
instance MimeRender HTMLLucid (Html a) where
    mimeRender _ = renderBS
```

For **blaze-html** everything works very similarly:

```
-- For this tutorial to compile 'HTMLLucid' and 'HTMLBlaze' have to be
-- distinct. Usually you would stick to one html rendering library and then
-- you can go with one 'HTML' type.
data HTMLBlaze
```

```
instance Accept HTMLBlaze where
    contentType _ = "text" // "html" /: ("charset", "utf-8")

instance ToMarkup a => MimeRender HTMLBlaze a where
    mimeRender _ = renderHtml . Text.Blaze.Html.toHtml

-- while we're at it, just like for lucid we can
-- provide an instance for rendering blaze's 'Html' type
instance MimeRender HTMLBlaze Text.Blaze.Html.Html where
    mimeRender _ = renderHtml
```

Both **servant-blaze** and **servant-lucid** let you use `HTMLLucid` and `HTMLBlaze` in any content-type list as long as you provide an instance of the appropriate class (`ToMarkup` for **blaze-html**, `ToHtml` for **lucid**).

We can now write a webservice that uses **servant-lucid** to show the `HTMLLucid` content-type in action. We will be serving the following API:

```
type PersonAPI = "persons" :> Get '[JSON, HTMLLucid] [Person]
```

where `Person` is defined as follows:

```
data Person = Person
  { firstName :: String
  , lastName  :: String
  } deriving Generic -- for the JSON instance

instance ToJSON Person
```

Now, let's teach **lucid** how to render a `Person` as a row in a table, and then a list of `Persons` as a table with a row per person.

```
-- HTML serialization of a single person
instance ToHtml Person where
  toHtml person =
    tr_ $ do
      td_ (toHtml $ firstName person)
      td_ (toHtml $ lastName person)

  -- do not worry too much about this
  toHtmlRaw = toHtml

-- HTML serialization of a list of persons
instance ToHtml [Person] where
  toHtml persons = table_ $ do
    tr_ $ do
      th_ "first name"
      th_ "last name"

    -- this just calls toHtml on each person of the list
    -- and concatenates the resulting pieces of HTML together
    foldMap toHtml persons

  toHtmlRaw = toHtml
```

We create some `Person` values and serve them as a list:

```
people :: [Person]
people =
  [ Person "Isaac"  "Newton"
  , Person "Albert" "Einstein"
```

```
  ]

personAPI :: Proxy PersonAPI
personAPI = Proxy

server4 :: Server PersonAPI
server4 = return people

app2 :: Application
app2 = serve personAPI server4
```

And we're good to go:

```
$ curl http://localhost:8081/persons
[{"lastName":"Newton","firstName":"Isaac"},{"lastName":"Einstein","firstName":"Albert"}]
$ curl -H 'Accept: text/html' http://localhost:8081/persons
<table><tr><td>first name</td><td>last name</td></tr><tr><td>Isaac</td><td>Newton</td></tr><tr><td>Al
# or just point your browser to http://localhost:8081/persons
```

The `Handler` monad

At the heart of the handlers is the monad they run in, namely a newtype `Handler` around `ExceptT ServantErr IO` (haddock documentation for `ExceptT`). One might wonder: why this monad? The answer is that it is the simplest monad with the following properties:

- it lets us both return a successful result (using `return`) or "fail" with a descriptive error (using `throwError`);
- it lets us perform IO, which is absolutely vital since most webservices exist as interfaces to databases that we interact with in `IO`.

Let's recall some definitions.

```
-- from the 'mtl' package at
newtype ExceptT e m a = ExceptT (m (Either e a))
```

In short, this means that a handler of type `Handler a` is simply equivalent to a computation of type `IO (Either ServantErr a)`, that is, an IO action that either returns an error or a result.

The module `Control.Monad.Except` from which `ExceptT` comes is worth looking at. Perhaps most importantly, `ExceptT` and `Handler` are an instances of `MonadError`, so `throwError` can be used to return an error from your handler (whereas `return` is enough to return a success).

Most of what you'll be doing in your handlers is running some IO and, depending on the result, you might sometimes want to throw an error of some kind and abort early. The next two sections cover how to do just that.

Performing IO

Another important instances from the list above are `MonadIO m => MonadIO (ExceptT e m)`, and therefore also `MonadIO Handler` as there is `MonadIO IO` instance.. `MonadIO` is a class from the **transformers** package defined as:

```
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
```

So if you want to run any kind of IO computation in your handlers, just use `liftIO`:

```
type IOAPI1 = "myfile.txt" :> Get '[JSON] FileContent

newtype FileContent = FileContent
  { content :: String }
  deriving Generic

instance ToJSON FileContent

server5 :: Server IOAPI1
server5 = do
  filecontent <- liftIO (readFile "myfile.txt")
  return (FileContent filecontent)
```

Failing, through `ServantErr`

If you want to explicitly fail at providing the result promised by an endpoint using the appropriate HTTP status code (not found, unauthorized, etc) and some error message, all you have to do is use the `throwError` function mentioned above and provide it with the appropriate value of type `ServantErr`, which is defined as:

```
data ServantErr = ServantErr
    { errHTTPCode     :: Int
    , errReasonPhrase :: String
    , errBody         :: ByteString -- lazy bytestring
    , errHeaders      :: [Header]
    }
```

Many standard values are provided out of the box by the `Servant.Server` module. If you want to use these values but add a body or some headers, just use record update syntax:

```
failingHandler :: Handler ()
failingHandler = throwError myerr

  where myerr :: ServantErr
        myerr = err503 { errBody = "Sorry dear user." }
```

Here's an example where we return a customised 404-Not-Found error message in the response body if "myfile.txt" isn't there:

```
server6 :: Server IOAPI1
server6 = do
  exists <- liftIO (doesFileExist "myfile.txt")
  if exists
    then liftIO (readFile "myfile.txt") >>= return . FileContent
    else throwError custom404Err

  where custom404Err = err404 { errBody = "myfile.txt just isn't there, please leave this server alo
```

Here's how that server looks in action:

```
$ curl --verbose http://localhost:8081/myfile.txt
[snip]
* Connected to localhost (127.0.0.1) port 8081 (#0)
> GET /myfile.txt HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8081
> Accept: */*
>
< HTTP/1.1 404 Not Found
```

```
[snip]
myfile.txt just isnt there, please leave this server alone.

$ echo Hello > myfile.txt

$ curl --verbose http://localhost:8081/myfile.txt
[snip]
* Connected to localhost (127.0.0.1) port 8081 (#0)
> GET /myfile.txt HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8081
> Accept: */*
>
< HTTP/1.1 200 OK
[snip]
< Content-Type: application/json
[snip]
{"content":"Hello\n"}
```

Response headers

To add headers to your response, use addHeader. Note that this changes the type of your API, as we can see in the following example:

```
type MyHandler = Get '[JSON] (Headers '[Header "X-An-Int" Int] User)

myHandler :: Server MyHandler
myHandler = return $ addHeader 1797 albert
```

Note that the type of `addHeader header x` is different than the type of `x`! And if you add more headers, more headers will appear in the header list:

```
type MyHeadfulHandler = Get '[JSON] (Headers '[Header "X-A-Bool" Bool, Header "X-An-Int" Int] User)

myHeadfulHandler :: Server MyHeadfulHandler
myHeadfulHandler = return $ addHeader True $ addHeader 1797 albert
```

But what if your handler only *sometimes* adds a header? If you declare that your handler adds headers, and you don't add one, the return type of your handler will be different than expected. To solve this, you have to explicitly *not* add a header by using `noHeader`:

```
type MyMaybeHeaderHandler
  = Capture "withHeader" Bool :> Get '[JSON] (Headers '[Header "X-An-Int" Int] User)

myMaybeHeaderHandler :: Server MyMaybeHeaderHandler
myMaybeHeaderHandler x = return $ if x then addHeader 1797 albert
                                       else noHeader albert
```

Serving static files

servant-server also provides a way to just serve the content of a directory under some path in your web API. As mentioned earlier in this document, the `Raw` combinator can be used in your APIs to mean "plug here any WAI application". Well, **servant-server** provides a function to get a file and directory serving WAI application, namely:

```
-- exported by Servant and Servant.Server
serveDirectoryWebApp :: FilePath -> Server Raw
```

`serveDirectoryWebApp`'s argument must be a path to a valid directory.

Here's an example API that will serve some static files:

```
type StaticAPI = "static" :> Raw
```

And the server:

```
staticAPI :: Proxy StaticAPI
staticAPI = Proxy
```

```
server7 :: Server StaticAPI
server7 = serveDirectoryWebApp "static-files"

app3 :: Application
app3 = serve staticAPI server7
```

This server will match any request whose path starts with `/static` and will look for a file at the path described by the rest of the request path, inside the *static-files/* directory of the path you run the program from.

In other words: If a client requests `/static/foo.txt`, the server will look for a file at `./static-files/foo.txt`. If that file exists it'll succeed and serve the file. If it doesn't exist, the handler will fail with a `404` status code.

`serveDirectoryWebApp` uses some standard settings that fit the use case of serving static files for most web apps. You can find out about the other options in the documentation of the `Servant.Utils.StaticFiles` module.

Nested APIs

Let's see how you can define APIs in a modular way, while avoiding repetition. Consider this simple example:

```
type UserAPI3 = -- view the user with given userid, in JSON
                Capture "userid" Int :> Get '[JSON] User

           :<|> -- delete the user with given userid. empty response
                Capture "userid" Int :> DeleteNoContent '[JSON] NoContent
```

We can instead factor out the `userid`:

```
type UserAPI4 = Capture "userid" Int :>
  (    Get '[JSON] User
  :<|> DeleteNoContent '[JSON] NoContent
  )
```

However, you have to be aware that this has an effect on the type of the corresponding `Server`:

```
Server UserAPI3 = (Int -> Handler User)
             :<|> (Int -> Handler NoContent)

Server UserAPI4 = Int -> (    Handler User
                         :<|> Handler NoContent
                         )
```

In the first case, each handler receives the *userid* argument. In the latter, the whole `Server` takes the *userid* and has handlers that are just computations in `Handler`, with no arguments. In other words:

```
server8 :: Server UserAPI3
server8 = getUser :<|> deleteUser

  where getUser :: Int -> Handler User
        getUser _userid = error "..."

        deleteUser :: Int -> Handler NoContent
        deleteUser _userid = error "..."

-- notice how getUser and deleteUser
-- have a different type! no argument anymore,
-- the argument directly goes to the whole Server
server9 :: Server UserAPI4
server9 userid = getUser userid :<|> deleteUser userid

  where getUser :: Int -> Handler User
        getUser = error "..."

        deleteUser :: Int -> Handler NoContent
        deleteUser = error "..."
```

Note that there's nothing special about `Capture` that lets you "factor it out": this can be done with any combinator. Here are a few examples of APIs with a combinator factored out for which we can write a perfectly valid `Server`.

```
-- we just factor out the "users" path fragment
type API1 = "users" :>
  (    Get '[JSON] [User] -- user listing
  :<|> Capture "userid" Int :> Get '[JSON] User -- view a particular user
  )

-- we factor out the Request Body
type API2 = ReqBody '[JSON] User :>
  (    Get '[JSON] User -- just display the same user back, don't register it
  :<|> PostNoContent '[JSON] NoContent  -- register the user. empty response
  )

-- we factor out a Header
type API3 = Header "Authorization" Token :>
  (    Get '[JSON] SecretData -- get some secret data, if authorized
  :<|> ReqBody '[JSON] SecretData :> PostNoContent '[JSON] NoContent -- add some secret data, if aut
  )

newtype Token = Token ByteString
newtype SecretData = SecretData ByteString
```

This approach lets you define APIs modularly and assemble them all into one big API type only at the end.

```
type UsersAPI =
       Get '[JSON] [User] -- list users
  :<|> ReqBody '[JSON] User :> PostNoContent '[JSON] NoContent -- add a user
  :<|> Capture "userid" Int :>
         ( Get '[JSON] User -- view a user
      :<|> ReqBody '[JSON] User :> PutNoContent '[JSON] NoContent -- update a user
      :<|> DeleteNoContent '[JSON] NoContent -- delete a user
         )

usersServer :: Server UsersAPI
usersServer = getUsers :<|> newUser :<|> userOperations
```

```
  where getUsers :: Handler [User]
        getUsers = error "..."

        newUser :: User -> Handler NoContent
        newUser = error "..."

        userOperations userid =
          viewUser userid :<|> updateUser userid :<|> deleteUser userid

          where
            viewUser :: Int -> Handler User
            viewUser = error "..."

            updateUser :: Int -> User -> Handler NoContent
            updateUser = error "..."

            deleteUser :: Int -> Handler NoContent
            deleteUser = error "..."
```

```
type ProductsAPI =
       Get '[JSON] [Product] -- list products
  :<|> ReqBody '[JSON] Product :> PostNoContent '[JSON] NoContent -- add a product
  :<|> Capture "productid" Int :>
         ( Get '[JSON] Product -- view a product
      :<|> ReqBody '[JSON] Product :> PutNoContent '[JSON] NoContent -- update a product
      :<|> DeleteNoContent '[JSON] NoContent -- delete a product
         )

data Product = Product { productId :: Int }

productsServer :: Server ProductsAPI
productsServer = getProducts :<|> newProduct :<|> productOperations

  where getProducts :: Handler [Product]
        getProducts = error "..."

        newProduct :: Product -> Handler NoContent
        newProduct = error "..."

        productOperations productid =
          viewProduct productid :<|> updateProduct productid :<|> deleteProduct productid

          where
            viewProduct :: Int -> Handler Product
            viewProduct = error "..."

            updateProduct :: Int -> Product -> Handler NoContent
            updateProduct = error "..."

            deleteProduct :: Int -> Handler NoContent
            deleteProduct = error "..."
```

```
type CombinedAPI = "users" :> UsersAPI
              :<|> "products" :> ProductsAPI

server10 :: Server CombinedAPI
server10 = usersServer :<|> productsServer
```

Finally, we can realize the user and product APIs are quite similar and abstract that away:

```
-- API for values of type 'a'
-- indexed by values of type 'i'
type APIFor a i =
       Get '[JSON] [a] -- list 'a's
  :<|> ReqBody '[JSON] a :> PostNoContent '[JSON] NoContent -- add an 'a'
  :<|> Capture "id" i :>
         ( Get '[JSON] a -- view an 'a' given its "identifier" of type 'i'
      :<|> ReqBody '[JSON] a :> PutNoContent '[JSON] NoContent -- update an 'a'
      :<|> DeleteNoContent '[JSON] NoContent -- delete an 'a'
         )

-- Build the appropriate 'Server'
-- given the handlers of the right type.
serverFor :: Handler [a] -- handler for listing of 'a's
          -> (a -> Handler NoContent) -- handler for adding an 'a'
          -> (i -> Handler a) -- handler for viewing an 'a' given its identifier of type 'i'
          -> (i -> a -> Handler NoContent) -- updating an 'a' with given id
          -> (i -> Handler NoContent) -- deleting an 'a' given its id
          -> Server (APIFor a i)
serverFor = error "..."
-- implementation left as an exercise. contact us on IRC
-- or the mailing list if you get stuck!
```

When your API contains the `EmptyAPI` combinator, you'll want to use `emptyServer` in the corresponding slot for your server, which will simply fail with 404 whenever a request reaches it:

```
type CombinedAPI2 = API :<|> "empty" :> EmptyAPI

server11 :: Server CombinedAPI2
server11 = server3 :<|> emptyServer
```

Using another monad for your handlers

Remember how `Server` turns combinators for HTTP methods into `Handler`? Well, actually, there's more to that. `Server` is actually a simple type synonym.

```
type Server api = ServerT api Handler
```

`ServerT` is the actual type family that computes the required types for the handlers that's part of the `HasServer` class. It's like `Server` except that it takes another parameter which is the monad you want your handlers to run in, or more generally the return types of your handlers. This third parameter is used for specifying the return type of the handler for an endpoint, e.g when computing `ServerT (Get '[JSON] Person) SomeMonad`. The result would be `SomeMonad Person`.

The first and main question one might have then is: how do we write handlers that run in another monad? How can we "bring back" the value from a given monad into something **servant** can understand?

Natural transformations

If we have a function that gets us from an `m a` to an `n a`, for any `a`, what do we have?

```
type (~>) m n = forall a. m a -> n a
```

For example:

```
listToMaybe' :: [] ~> Maybe
listToMaybe' = listToMaybe -- from Data.Maybe
```

Note that `servant` doesn't declare the ~> type-alias, as the unfolded variant isn't much longer to write, as we'll see shortly.

So if you want to write handlers using another monad/type than `Handler`, say the `Reader String` monad, the first thing you have to prepare is a function:

```
readerToHandler :: Reader String a -> Handler a
```

We obviously have to run the `Reader` computation by supplying it with a `String`, like `"hi"`. We get an `a` out from that and can then just `return` it into `Handler`.

```
readerToHandler :: Reader String a -> Handler a
readerToHandler r = return (runReader r "hi")
```

We can write some simple webservice with the handlers running in `Reader String`.

```
type ReaderAPI = "a" :> Get '[JSON] Int
            :<|> "b" :> ReqBody '[JSON] Double :> Get '[JSON] Bool

readerAPI :: Proxy ReaderAPI
readerAPI = Proxy

readerServerT :: ServerT ReaderAPI (Reader String)
readerServerT = a :<|> b where
    a :: Reader String Int
    a = return 1797

    b :: Double -> Reader String Bool
    b _ = asks (== "hi")
```

We unfortunately can't use `readerServerT` as an argument of `serve`, because `serve` wants a `Server ReaderAPI`, i.e., with handlers running in `Handler`. But there's a simple solution to this.

Welcome `hoistServer`

That's right. We have just written `readerToHandler`, which is exactly what we would need to apply to all handlers to make the handlers have the right type for `serve`. Being cumbersome to do by hand, we provide a function `hoistServer` which takes a natural transformation between two parameterized types `m` and `n` and a `ServerT someapi m`, and returns a `ServerT someapi n`.

In our case, we can wrap up our little webservice by using `hoistServer readerAPI readerToHandler` on our handlers.

```
readerServer :: Server ReaderAPI
readerServer = hoistServer readerAPI readerToHandler readerServerT

app4 :: Application
app4 = serve readerAPI readerServer
```

This is the webservice in action:

```
$ curl http://localhost:8081/a
1797
$ curl http://localhost:8081/b
"hi"
```

An arrow is a reader too.

In previous versions of `servant` we had an `enter` to do what `hoistServer` does now. `enter` had a ambitious design goals, but was problematic in practice.

One problematic situation was when the source monad was `(->) r`, yet it's handy in practice, because `(->) r` is isomorphic to `Reader r`.

We can rewrite the previous example without `Reader`:

```
funServerT :: ServerT ReaderAPI ((->) String)
funServerT = a :<|> b where
    a :: String -> Int
    a _ = 1797

    -- unfortunately, we cannot make `String` the first argument.
    b :: Double -> String -> Bool
    b _ s = s == "hi"

funToHandler :: (String -> a) -> Handler a
funToHandler f = return (f "hi")

app5 :: Application
app5 = serve readerAPI (hoistServer readerAPI funToHandler funServerT)
```

Streaming endpoints

We can create endpoints that don't just give back a single result, but give back a *stream* of results, served one at a time. Stream endpoints only provide a single content type, and also specify what framing strategy is used to delineate the results. To serve these results, we need to give back a stream producer. Adapters can be written to `Pipes`, `Conduit` and the like, or written directly as `StreamGenerators`. StreamGenerators are IO-based continuations that are handed two functions – the first to write the first result back, and the second to write all subsequent results back. (This is to allow handling of situations where the entire stream is prefixed by a header, or where a boundary is written between elements, but not prior to the first element). The API of a streaming endpoint needs to explicitly specify which sort of generator it produces. Note that the generator itself is returned by a `Handler` action, so that additional IO may be done in the creation of one.

```
type StreamAPI = "userStream" :> StreamGet NewlineFraming JSON (StreamGenerator User)
streamAPI :: Proxy StreamAPI
streamAPI = Proxy

streamUsers :: StreamGenerator User
streamUsers = StreamGenerator $ \sendFirst sendRest -> do
                        sendFirst isaac
                        sendRest  albert
                        sendRest  albert

app6 :: Application
app6 = serve streamAPI (return streamUsers)
```

This simple application returns a stream of `User` values encoded in JSON format, with each value separated by a newline. In this case, the stream will consist of the value of `isaac`, followed by the value of `albert`, then the value of `albert` a third time. Importantly, the stream is written back as results are produced, rather than all at once. This means first that results are delivered when they are available, and second, that if an exception interrupts production of the full stream, nonetheless partial results have already been written back.

Conclusion

You're now equipped to write webservices/web-applications using **servant**. The rest of this document focuses on **servant-client**, **servant-js** and **servant-docs**.

Querying an API

While defining handlers that serve an API has a lot to it, querying an API is simpler: we do not care about what happens inside the webserver, we just need to know how to talk to it and get a response back. That said, we usually have to write the querying functions by hand because the structure of the API isn't a first class citizen and can't be inspected to generate the client-side functions.

servant however has a way to inspect APIs, because APIs are just Haskell types and (GHC) Haskell lets us do quite a few things with types. In the same way that we look at an API type to deduce the types the handlers should have, we can inspect the structure of the API to *derive* Haskell functions that take one argument for each occurrence of `Capture`, `ReqBody`, `QueryParam` and friends (see the tutorial introduction for an overview). By *derive*, we mean that there's no code generation involved - the functions are defined just by the structure of the API type.

The source for this tutorial section is a literate Haskell file, so first we need to have some language extensions and imports:

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

module Client where

import Data.Aeson
import Data.Proxy
import GHC.Generics
import Network.HTTP.Client (newManager, defaultManagerSettings)
import Servant.API
import Servant.Client
```

Also, we need examples for some domain specific data types:

```
data Position = Position
  { xCoord :: Int
  , yCoord :: Int
  } deriving (Show, Generic)

instance FromJSON Position

newtype HelloMessage = HelloMessage { msg :: String }
  deriving (Show, Generic)

instance FromJSON HelloMessage

data ClientInfo = ClientInfo
  { clientName :: String
  , clientEmail :: String
  , clientAge :: Int
  , clientInterestedIn :: [String]
  } deriving Generic

instance ToJSON ClientInfo
```

```
data Email = Email
  { from :: String
  , to :: String
  , subject :: String
  , body :: String
  } deriving (Show, Generic)

instance FromJSON Email
```

Enough chitchat, let's see an example. Consider the following API type from the previous section:

```
type API = "position" :> Capture "x" Int :> Capture "y" Int :> Get '[JSON] Position
      :<|> "hello" :> QueryParam "name" String :> Get '[JSON] HelloMessage
      :<|> "marketing" :> ReqBody '[JSON] ClientInfo :> Post '[JSON] Email
```

What we are going to get with **servant-client** here is three functions, one to query each endpoint:

```
position :: Int -- ^ value for "x"
         -> Int -- ^ value for "y"
         -> ClientM Position

hello :: Maybe String -- ^ an optional value for "name"
      -> ClientM HelloMessage

marketing :: ClientInfo -- ^ value for the request body
          -> ClientM Email
```

Each function makes available as an argument any value that the response may depend on, as evidenced in the API type. How do we get these functions? By calling the function `client`. It takes one argument:

- a `Proxy` to your API,

```
api :: Proxy API
api = Proxy

position :<|> hello :<|> marketing = client api
```

`client api` returns client functions for our *entire* API, combined with `:<|>`, which we can pattern match on as above. You could say `client` "calculates" the correct type and number of client functions for the API type it is given (via a `Proxy`), as well as their implementations.

If you have an `EmptyAPI` in your API, servant-client will hand you a value of type `EmptyClient` in the corresponding slot, where `data EmptyClient = EmptyClient`, as a way to indicate that you can't do anything useful with it.

```
type API' = API :<|> EmptyAPI

api' :: Proxy API'
api' = Proxy

(position' :<|> hello' :<|> marketing') :<|> EmptyClient = client api'
```

```
-- | URI scheme to use
data Scheme =
    Http  -- ^ http://
  | Https -- ^ https://
  deriving

-- | Simple data type to represent the target of HTTP requests
```

```
--   for servant's automatically-generated clients.
data BaseUrl = BaseUrl
  { baseUrlScheme :: Scheme -- ^ URI scheme to use
  , baseUrlHost :: String   -- ^ host (eg "haskell.org")
  , baseUrlPort :: Int      -- ^ port (eg 80)
  , baseUrlPath :: String   -- ^ path (eg "/a/b/c")
  }
```

That's it. Let's now write some code that uses our client functions.

```
queries :: ClientM (Position, HelloMessage, Email)
queries = do
  pos <- position 10 10
  message <- hello (Just "servant")
  em  <- marketing (ClientInfo "Alp" "alp@foo.com" 26 ["haskell", "mathematics"])
  return (pos, message, em)

run :: IO ()
run = do
  manager' <- newManager defaultManagerSettings
  res <- runClientM queries (mkClientEnv manager' (BaseUrl Http "localhost" 8081 ""))
  case res of
    Left err -> putStrLn $ "Error: " ++ show err
    Right (pos, message, em) -> do
      print pos
      print message
      print em
```

Here's the output of the above code running against the appropriate server:

```
Position {xCoord = 10, yCoord = 10}
HelloMessage {msg = "Hello, servant"}
Email {from = "great@company.com", to = "alp@foo.com", subject = "Hey Alp, we miss you!", body = "Hi
```

The types of the arguments for the functions are the same as for (server-side) request handlers.

Changing the monad the client functions live in

Just like `hoistServer` allows us to change the monad in which request handlers of a web application live in, we also have `hoistClient` for changing the monad in which *client functions* live. Consider the following trivial API:

```
type HoistClientAPI = Get '[JSON] Int :<|> Capture "n" Int :> Post '[JSON] Int

hoistClientAPI :: Proxy HoistClientAPI
hoistClientAPI = Proxy
```

We already know how to derive client functions for this API, and as we have seen above they all return results in the `ClientM` monad when using `servant-client`. However, `ClientM` rarely (or never) is the actual monad we need to use the client functions in. Sometimes we need to run them in IO, sometimes in a custom monad stack. `hoistClient` is a very simple solution to the problem of "changing" the monad the clients run in.

```
hoistClient
  :: HasClient ClientM api   -- we need a valid API
  => Proxy api               -- a Proxy to the API type
  -> (forall a. m a -> n a)  -- a "monad conversion function" (natural transformation)
  -> Client m api            -- clients in the source monad
  -> Client n api            -- result: clients in the target monad
```

The "conversion function" argument above, just like the ones given to `hoistServer`, must be able to turn an `m a` into an `n a` for any choice of type `a`.

Let's see this in action on our example. We first derive our client functions as usual, with all of them returning a result in `ClientM`.

```
getIntClientM :: ClientM Int
postIntClientM :: Int -> ClientM Int
getIntClientM :<|> postIntClientM = client hoistClientAPI
```

And we finally decide that we want the handlers to run in IO instead, by "post-applying" `runClientM` to a fixed client environment.

```
-- our conversion function has type: forall a. ClientM a -> IO a
-- the result has type:
-- Client IO HoistClientAPI = IO Int :<|> (Int -> IO Int)
getClients :: ClientEnv -> Client IO HoistClientAPI
getClients clientEnv
  = hoistClient hoistClientAPI
                ( fmap (either (error . show) id)
                . flip runClientM clientEnv
                )
                (client hoistClientAPI)
```

Querying Streaming APIs.

Consider the following streaming API type:

```
type StreamAPI = "positionStream" :> StreamGet NewlineFraming JSON (ResultStream Position)
```

Note that when we declared an API to serve, we specified a `StreamGenerator` as a producer of streams. Now we specify our result type as a `ResultStream`. With types that can be used both ways, if appropriate adaptors are written (in the form of `ToStreamGenerator` and `BuildFromStream` instances), then this asymmetry isn't necessary. Otherwise, if you want to share the same API across clients and servers, you can parameterize it like so:

```
type StreamAPI f = "positionStream" :> StreamGet NewlineFraming JSON (f Position)
type ServerStreamAPI = StreamAPI StreamGenerator
type ClientStreamAPI = StreamAPI ResultStream
```

In any case, here's how we write a function to query our API:

```
streamAPI :: Proxy StreamAPI
streamAPI = Proxy

posStream :: ClientM (ResultStream Position)

posStream = client streamAPI
```

And here's how to just print out all elements from a `ResultStream`, to give some idea of how to work with them.

```
printResultStream :: Show a => ResultStream a -> IO ()
printResultStream (ResultStream k) = k $ \getResult ->
       let loop = do
            r <- getResult
            case r of
                Nothing -> return ()
                Just x -> print x >> loop
       in loop
```

The stream is parsed and provided incrementally. So the above loop prints out each result as soon as it is received on the stream, rather than waiting until they are all available to print them at once.

You now know how to use **servant-client**!

Generating Javascript functions to query an API

We will now see how **servant** lets you turn an API type into javascript functions that you can call to query a webservice.

For this, we will consider a simple page divided in two parts. At the top, we will have a search box that lets us search in a list of Haskell books by author/title with a list of results that gets updated every time we enter or remove a character, while at the bottom we will be able to see the classical probabilistic method to approximate pi, using a webservice to get random points. Finally, we will serve an HTML file along with a couple of Javascript files, among which one that's automatically generated from the API type and which will provide ready-to-use functions to query your API.

The source for this tutorial section is a literate haskell file, so first we need to have some language extensions and imports:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeOperators #-}

module Javascript where

import Control.Monad.IO.Class
import Data.Aeson
import Data.Proxy
import Data.Text as T (Text)
import Data.Text.IO as T (writeFile, readFile)
import GHC.Generics
import Language.Javascript.JQuery
import Network.Wai
import Network.Wai.Handler.Warp
import qualified Data.Text as T
import Servant
import Servant.JS
import System.Random
```

Now let's have the API type(s) and the accompanying datatypes.

```
type API = "point" :> Get '[JSON] Point
      :<|> "books" :> QueryParam "q" Text :> Get '[JSON] (Search Book)

type API' = API :<|> Raw

data Point = Point
  { x :: Double
  , y :: Double
  } deriving Generic

instance ToJSON Point

data Search a = Search
  { query   :: Text
  , results :: [a]
  } deriving Generic
```

```
mkSearch :: Text -> [a] -> Search a
mkSearch = Search

instance ToJSON a => ToJSON (Search a)

data Book = Book
  { author :: Text
  , title  :: Text
  , year   :: Int
  } deriving Generic

instance ToJSON Book

book :: Text -> Text -> Int -> Book
book = Book
```

We need a "book database". For the purpose of this guide, let's restrict ourselves to the following books.

```
books :: [Book]
books =
  [ book "Paul Hudak" "The Haskell School of Expression: Learning Functional Programming through Mult
  , book "Bryan O'Sullivan, Don Stewart, and John Goerzen" "Real World Haskell" 2008
  , book "Miran Lipovača" "Learn You a Haskell for Great Good!" 2011
  , book "Graham Hutton" "Programming in Haskell" 2007
  , book "Simon Marlow" "Parallel and Concurrent Programming in Haskell" 2013
  , book "Richard Bird" "Introduction to Functional Programming using Haskell" 1998
  ]
```

Now, given an optional search string `q`, we want to perform a case insensitive search in that list of books. We're obviously not going to try and implement the best possible algorithm, this is out of scope for this tutorial. The following simple linear scan will do, given how small our list is.

```
searchBook :: Monad m => Maybe Text -> m (Search Book)
searchBook Nothing  = return (mkSearch "" books)
searchBook (Just q) = return (mkSearch q books')

  where books' = filter (\b -> q' `T.isInfixOf` T.toLower (author b)
                            || q' `T.isInfixOf` T.toLower (title b)
                        )
                        books
        q' = T.toLower q
```

We also need an endpoint that generates random points `(x, y)` with `-1 <= x,y <= 1`. The code below uses random's `System.Random`.

```
randomPoint :: MonadIO m => m Point
randomPoint = liftIO . getStdRandom $ \g ->
  let (rx, g')  = randomR (-1, 1) g
      (ry, g'') = randomR (-1, 1) g'
  in (Point rx ry, g'')
```

If we add static file serving, our server is now complete.

```
api :: Proxy API
api = Proxy

api' :: Proxy API'
api' = Proxy
```

```
server :: Server API
server = randomPoint
    :<|> searchBook

server' :: Server API'
server' = server
     :<|> serveDirectoryFileServer "static"

app :: Application
app = serve api' server'

main :: IO ()
main = run 8000 app
```

Why two different API types, proxies and servers though? Simply because we don't want to generate javascript functions for the `Raw` part of our API type, so we need a `Proxy` for our API type `API'` without its `Raw` endpoint.

The `EmptyAPI` combinator needs no special treatment as it generates no Javascript functions: an empty API has no endpoints to access.

Very similarly to how one can derive haskell functions, we can derive the javascript with just a simple function call to `jsForAPI` from `Servant.JS`.

```
apiJS1 :: Text
apiJS1 = jsForAPI api jquery
```

This `Text` contains 2 Javascript functions, 'getPoint' and 'getBooks':

```
var getPoint = function(onSuccess, onError)
{
  $.ajax(
    { url: '/point'
    , success: onSuccess
    , error: onError
    , type: 'GET'
    });
}

var getBooks = function(q, onSuccess, onError)
{
  $.ajax(
    { url: '/books' + '?q=' + encodeURIComponent(q)
    , success: onSuccess
    , error: onError
    , type: 'GET'
    });
}
```

We created a directory `static` that contains two static files: `index.html`, which is the entrypoint to our little web application; and `ui.js`, which contains some hand-written javascript. This javascript code assumes the two generated functions `getPoint` and `getBooks` in scope. Therefore we need to write the generated javascript into a file:

```
writeJSFiles :: IO ()
writeJSFiles = do
  T.writeFile "static/api.js" apiJS1
  jq <- T.readFile =<< Language.Javascript.JQuery.file
  T.writeFile "static/jq.js" jq
```

(We're also writing the jquery library into a file, as it's also used by `ui.js`.) `static/api.js` will be included in `index.html` and the two generated functions will therefore be available in `ui.js`.

And we're good to go. You can start the `main` function of this file and go to `http://localhost:8000/`. Start typing in the name of one of the authors in our database or part of a book title, and check out how long it takes to approximate pi using the method mentioned above.

Customizations

Instead of calling `jquery`, you can call its variant `jqueryWith`. Here are the type definitions

```
jquery :: JavaScriptGenerator
jqueryWith :: CommonGeneratorOptions -> JavaScriptGenerator
```

The `CommonGeneratorOptions` will let you define different behaviors to change how functions are generated. Here is the definition of currently available options:

```
data CommonGeneratorOptions = CommonGeneratorOptions
  {
    -- | function generating function names
    functionNameBuilder :: FunctionName -> Text
    -- | name used when a user want to send the request body (to let you redefine it)
  , requestBody :: Text
    -- | name of the callback parameter when the request was successful
  , successCallback :: Text
    -- | name of the callback parameter when the request reported an error
  , errorCallback :: Text
    -- | namespace on which we define the js function (empty mean local var)
  , moduleName :: Text
    -- | a prefix that should be prepended to the URL in the generated JS
  , urlPrefix :: Text
  }
```

This pattern is available with all supported backends, and default values are provided.

Vanilla support

If you don't use JQuery for your application, you can reduce your dependencies to simply use the `XMLHttpRequest` object from the standard API.

Use the same code as before but simply replace the previous `apiJS` with the following one:

```
apiJS2 :: Text
apiJS2 = jsForAPI api vanillaJS
```

The rest is *completely* unchanged.

The output file is a bit different, but it has the same parameters,

```
var getPoint = function(onSuccess, onError)
{
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/point', true);
  xhr.setRequestHeader(\"Accept\",\"application/json\");
  xhr.onreadystatechange = function (e) {
    if (xhr.readyState == 4) {
      if (xhr.status == 204 || xhr.status == 205) {
        onSuccess();
      } else if (xhr.status >= 200 && xhr.status < 300) {
```

```
        var value = JSON.parse(xhr.responseText);
        onSuccess(value);
      } else {
        var value = JSON.parse(xhr.responseText);
        onError(value);
      }
    }
  }
  xhr.send(null);
}

var getBooks = function(q, onSuccess, onError)
{
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/books' + '?q=' + encodeURIComponent(q), true);
  xhr.setRequestHeader(\"Accept\",\"application/json\");
  xhr.onreadystatechange = function (e) {
    if (xhr.readyState == 4) {
      if (xhr.status == 204 || xhr.status == 205) {
        onSuccess();
      } else if (xhr.status >= 200 && xhr.status < 300) {
        var value = JSON.parse(xhr.responseText);
        onSuccess(value);
      } else {
        var value = JSON.parse(xhr.responseText);
        onError(value);
      }
    }
  }
  xhr.send(null);
}
```

And that's all, your web service can of course be accessible from those two clients at the same time!

Axios support

Simple usage

If you use Axios library for your application, we support that too!

Use the same code as before but simply replace the previous `apiJS` with the following one:

```
apiJS3 :: Text
apiJS3 = jsForAPI api $ axios defAxiosOptions
```

The rest is *completely* unchanged.

The output file is a bit different,

```
var getPoint = function()
{
  return axios({ url: '/point'
    , method: 'get'
    });
}
```

```
var getBooks = function(q)
{
  return axios({ url: '/books' + '?q=' + encodeURIComponent(q)
    , method: 'get'
    });
}
```

Caution: In order to support the promise style of the API, there are no onSuccess nor onError callback functions.

Defining Axios configuration

Axios lets you define a 'configuration' to determine the behavior of the program when the AJAX request is sent.

We mapped this into a configuration

```
data AxiosOptions = AxiosOptions
  { -- | indicates whether or not cross-site Access-Control requests
    -- should be made using credentials
    withCredentials :: !Bool
    -- | the name of the cookie to use as a value for xsrf token
  , xsrfCookieName :: !(Maybe Text)
    -- | the name of the header to use as a value for xsrf token
  , xsrfHeaderName :: !(Maybe Text)
  }
```

Angular support

Simple usage

You can apply the same procedure as with `vanillaJS` and `jquery`, and generate top level functions.

The difference is that `angular` Generator always takes an argument.

```
apiJS4 :: Text
apiJS4 = jsForAPI api $ angular defAngularOptions
```

The generated code will be a bit different than previous generators. An extra argument `$http` will be added to let Angular magical Dependency Injector operate.

Caution: In order to support the promise style of the API, there are no onSuccess nor onError callback functions.

```
var getPoint = function($http)
{
  return $http(
    { url: '/point'
    , method: 'GET'
    });
}



var getBooks = function($http, q)
{
  return $http(
    { url: '/books' + '?q=' + encodeURIComponent(q)
```

```
    , method: 'GET'
    });
}
```

You can then build your controllers easily

```
app.controller("MyController", function($http) {
  this.getPoint = getPoint($http)
    .success(/* Do something */)
    .error(/* Report error */);

  this.getPoint = getBooks($http, q)
    .success(/* Do something */)
    .error(/* Report error */);
});
```

Service generator

You can also generate automatically a service to wrap the whole API as a single Angular service:

```
app.service('MyService', function($http) {
  return ({
  postCounter: function()
  {
   return $http(
     { url: '/counter'
     , method: 'POST'
      });
  },
  getCounter: function()
  {
   return $http(
     { url: '/books' + '?q=' + encodeURIComponent(q), true);
     , method: 'GET'
      });
  }
  });
});
```

To do so, you just have to use an alternate generator.

```
apiJS5 :: Text
apiJS5 = jsForAPI api $ angularService defAngularOptions
```

Again, it is possible to customize some portions with the options.

```
data AngularOptions = AngularOptions
  { -- | When generating code with wrapInService, name of the service to generate, default is 'app'
    serviceName :: Text
  , -- | beginning of the service definition
    prologue :: Text -> Text -> Text
  , -- | end of the service definition
    epilogue :: Text
  }
```

Custom function name builder

Servant comes with three name builders included:

- camelCase (the default)
- concatCase
- snakeCase

Keeping the JQuery as an example, let's see the impact:

```
apiJS6 :: Text
apiJS6 = jsForAPI api $ jqueryWith defCommonGeneratorOptions { functionNameBuilder= snakeCase }
```

This `Text` contains 2 Javascript functions:

```
var get_point = function(onSuccess, onError)
{
  $.ajax(
    { url: '/point'
    , success: onSuccess
    , error: onError
    , type: 'GET'
    });
}

var get_books = function(q, onSuccess, onError)
{
  $.ajax(
    { url: '/books' + '?q=' + encodeURIComponent(q)
    , success: onSuccess
    , error: onError
    , type: 'GET'
    });
}
```

Documenting an API

The source for this tutorial section is a literate haskell file, so first we need to have some language extensions and imports:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeOperators #-}
{-# OPTIONS_GHC -fno-warn-orphans #-}

module Docs where

import Data.ByteString.Lazy (ByteString)
import Data.Proxy
import Data.Text.Lazy.Encoding (encodeUtf8)
import Data.Text.Lazy (pack)
import Network.HTTP.Types
import Network.Wai
```

```
import Servant.API
import Servant.Docs
import Servant.Server
import Web.FormUrlEncoded(FromForm(..), ToForm(..))
```

And we'll import some things from one of our earlier modules (Serving an API):

```
import Server (Email(..), ClientInfo(..), Position(..), HelloMessage(..),
  server3, emailForClient)
```

Like client function generation, documentation generation amounts to inspecting the API type and extracting all the data we need to then present it in some format to users of your API.

This time however, we have to assist **servant**. While it is able to deduce a lot of things about our API, it can't magically come up with descriptions of the various pieces of our APIs that are human-friendly and explain what's going on "at the business-logic level". A good example to study for documentation generation is our webservice with the `/position`, `/hello` and `/marketing` endpoints from earlier:

```
type ExampleAPI = "position" :> Capture "x" Int :> Capture "y" Int :> Get '[JSON] Position
      :<|> "hello" :> QueryParam "name" String :> Get '[JSON] HelloMessage
      :<|> "marketing" :> ReqBody '[JSON] ClientInfo :> Post '[JSON] Email

exampleAPI :: Proxy ExampleAPI
exampleAPI = Proxy
```

While **servant** can see e.g. that there are 3 endpoints and that the response bodies will be in JSON, it doesn't know what influence the captures, parameters, request bodies and other combinators have on the webservice. This is where some manual work is required.

For every capture, request body, response body, query param, we have to give some explanations about how it influences the response, what values are possible and the likes. Here's how it looks like for the parameters we have above.

```
instance ToCapture (Capture "x" Int) where
  toCapture _ =
    DocCapture "x"                                  -- name
               "(integer) position on the x axis" -- description

instance ToCapture (Capture "y" Int) where
  toCapture _ =
    DocCapture "y"                                  -- name
               "(integer) position on the y axis" -- description

instance ToSample Position where
  toSamples _ = singleSample (Position 3 14) -- example of output

instance ToParam (QueryParam "name" String) where
  toParam _ =
    DocQueryParam "name"                        -- name
                  ["Alp", "John Doe", "..."] -- example of values (not necessarily exhaustive)
                  "Name of the person to say hello to." -- description
                  Normal -- Normal, List or Flag

instance ToSample HelloMessage where
  toSamples _ =
    [ ("When a value is provided for 'name'", HelloMessage "Hello, Alp")
    , ("When 'name' is not specified", HelloMessage "Hello, anonymous coward")
    ]
    -- mutliple examples to display this time
```

```
ci :: ClientInfo
ci = ClientInfo "Alp" "alp@foo.com" 26 ["haskell", "mathematics"]

instance ToSample ClientInfo where
  toSamples _ = singleSample ci

instance ToSample Email where
  toSamples _ = singleSample (emailForClient ci)
```

Types that are used as request or response bodies have to instantiate the `ToSample` typeclass which lets you specify one or more examples of values. `Captures` and `QueryParams` have to instantiate their respective `ToCapture` and `ToParam` classes and provide a name and some information about the concrete meaning of that argument, as illustrated in the code above. The `EmptyAPI` combinator needs no special treatment as it generates no documentation: an empty API has no endpoints to document.

With all of this, we can derive docs for our API.

```
apiDocs :: API
apiDocs = docs exampleAPI
```

`API` is a type provided by **servant-docs** that stores all the information one needs about a web API in order to generate documentation in some format. Out of the box, **servant-docs** only provides a pretty documentation printer that outputs Markdown, but the **servant-pandoc** package can be used to target many useful formats.

servant's markdown pretty printer is a function named `markdown`.

```
markdown :: API -> String
```

That lets us see what our API docs look down in markdown, by looking at `markdown apiDocs`.

```
## GET /hello

#### GET Parameters:

- name
    - **Values**: *Alp, John Doe, ...*
    - **Description**: Name of the person to say hello to.


#### Response:

- Status code 200
- Headers: []

- Supported content types are:

    - `application/json;charset=utf-8`
    - `application/json`

- When a value is provided for 'name' (`application/json;charset=utf-8`, `application/json`):

    ```javascript
{"msg":"Hello, Alp"}
    ```

- When 'name' is not specified (`application/json;charset=utf-8`, `application/json`):

    ```javascript
{"msg":"Hello, anonymous coward"}
```
```

```
    ```

POST /marketing

Request:

- Supported content types are:

    - `application/json;charset=utf-8`
    - `application/json`

- Example (`application/json;charset=utf-8`, `application/json`):

    ```javascript
{"clientAge":26,"clientEmail":"alp@foo.com","clientName":"Alp","clientInterestedIn":["haskell","mathe
    ```

Response:

- Status code 200
- Headers: []

- Supported content types are:

    - `application/json;charset=utf-8`
    - `application/json`

- Example (`application/json;charset=utf-8`, `application/json`):

    ```javascript
{"subject":"Hey Alp, we miss you!","body":"Hi Alp,\n\nSince you've recently turned 26, have you check
    ```

GET /position/:x/:y

Captures:

- *x*: (integer) position on the x axis
- *y*: (integer) position on the y axis

Response:

- Status code 200
- Headers: []

- Supported content types are:

    - `application/json;charset=utf-8`
    - `application/json`

- Example (`application/json;charset=utf-8`, `application/json`):

    ```javascript
{"yCoord":14,"xCoord":3}
    ````
```

However, we can also add one or more introduction sections to the document. We just need to tweak the way we generate `apiDocs`. We will also convert the content to a lazy `ByteString` since this is what **wai** expects for `Raw`
```

endpoints.

```
docsBS :: ByteString
docsBS = encodeUtf8
       . pack
       . markdown
       $ docsWithIntros [intro] exampleAPI

  where intro = DocIntro "Welcome" ["This is our super webservice's API.", "Enjoy!"]
```

`docsWithIntros` just takes an additional parameter, a list of `DocIntros` that must be displayed before any endpoint docs.

More customisation can be done with the `markdownWith` function, which allows customising some of the parameters used when generating Markdown. The most obvious of these is how to handle when a request or response body has multiple content types. For example, if we make a slight change to the `/marketing` endpoint of our API so that the request body can also be encoded as a form:

```
type ExampleAPI2 = "position" :> Capture "x" Int :> Capture "y" Int :> Get '[JSON] Position
      :<|> "hello" :> QueryParam "name" String :> Get '[JSON] HelloMessage
      :<|> "marketing" :> ReqBody '[JSON, FormUrlEncoded] ClientInfo :> Post '[JSON] Email

instance ToForm ClientInfo
instance FromForm ClientInfo

exampleAPI2 :: Proxy ExampleAPI2
exampleAPI2 = Proxy

api2Docs :: API
api2Docs = docs exampleAPI2
```

The relevant output of `markdown api2Docs` is now:

````
#### Request:

- Supported content types are:

    - `application/json;charset=utf-8`
    - `application/json`
    - `application/x-www-form-urlencoded`

- Example (`application/json;charset=utf-8`, `application/json`):

    ```javascript
{"clientAge":26,"clientEmail":"alp@foo.com","clientName":"Alp","clientInterestedIn":["haskell","mathe
    ```

- Example (`application/x-www-form-urlencoded`):

    ```
clientAge=26&clientEmail=alp%40foo.com&clientName=Alp&clientInterestedIn=haskell&clientInterestedIn=m
    ```
````

If, however, you don't want the extra example encoding shown, then you can use `markdownWith (defRenderingOptions & requestExamples .~ FirstContentType)` to get behaviour identical to `markdown apiDocs`.

We can now serve the API *and* the API docs with a simple server.

```
type DocsAPI = ExampleAPI :<|> Raw

api :: Proxy DocsAPI
api = Proxy

server :: Server DocsAPI
server = Server.server3 :<|> Tagged serveDocs where
    serveDocs _ respond =
        respond $ responseLBS ok200 [plain] docsBS
    plain = ("Content-Type", "text/plain")

app :: Application
app = serve api server
```

And if you spin up this server and request anything else than `/position`, `/hello` and `/marketing`, you will see the API docs in markdown. This is because `serveDocs` is attempted if the 3 other endpoints don't match and systematically succeeds since its definition is to just return some fixed bytestring with the `text/plain` content type.

Authentication in Servant

Once you've established the basic routes and semantics of your API, it's time to consider protecting parts of it. Authentication and authorization are broad and nuanced topics; as servant began to explore this space we started small with one of HTTP's earliest authentication schemes: Basic Authentication.

Servant `0.5` shipped with out-of-the-box support for Basic Authentication. However, we recognize that every web application is its own beautiful snowflake and are offering experimental support for generalized or ad-hoc authentication.

In this tutorial we'll build two APIs. One protecting certain routes with Basic Authentication and another protecting the same routes with a custom, in-house authentication scheme.

Basic Authentication

When protecting endpoints with basic authentication, we need to specify two items:

1. The **realm** of authentication as per the Basic Authentication spec.
2. The datatype returned by the server after authentication is verified. This is usually a `User` or `Customer` datatype.

With those two items in mind, *servant* provides the following combinator:

```
data BasicAuth (realm :: Symbol) (userData :: *)
```

You can use this combinator to protect an API as follows:

```
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE DeriveGeneric          #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings      #-}
{-# LANGUAGE ScopedTypeVariables    #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE TypeOperators          #-}
{-# LANGUAGE UndecidableInstances   #-}
```

```
module Authentication where

import Data.Aeson                       (ToJSON)
import Data.ByteString                  (ByteString)
import Data.Map                         (Map, fromList)
import Data.Monoid                      ((<>))
import qualified Data.Map            as Map
import Data.Proxy                       (Proxy (Proxy))
import Data.Text                        (Text)
import GHC.Generics                     (Generic)
import Network.Wai                      (Request, requestHeaders)
import Network.Wai.Handler.Warp         (run)
import Servant.API                      ((:<|>) ((:<|>)), (:>), BasicAuth,
                                         Get, JSON)
import Servant.API.BasicAuth            (BasicAuthData (BasicAuthData))
import Servant.API.Experimental.Auth    (AuthProtect)
import Servant                          (throwError)
import Servant.Server                   (BasicAuthCheck (BasicAuthCheck),
                                         BasicAuthResult( Authorized
                                                        , Unauthorized
                                                        ),
                                         Context ((:.), EmptyContext),
                                         err401, err403, errBody, Server,
                                         serveWithContext, Handler)
import Servant.Server.Experimental.Auth (AuthHandler, AuthServerData,
                                         mkAuthHandler)
import Servant.Server.Experimental.Auth()
import Web.Cookie                       (parseCookies)

-- | private data that needs protection
newtype PrivateData = PrivateData { ssshhh :: Text }
  deriving (Eq, Show, Generic)

instance ToJSON PrivateData

-- | public data that anyone can use.
newtype PublicData = PublicData { somedata :: Text }
  deriving (Eq, Show, Generic)

instance ToJSON PublicData

-- | A user we'll grab from the database when we authenticate someone
newtype User = User { userName :: Text }
  deriving (Eq, Show)

-- | a type to wrap our public api
type PublicAPI = Get '[JSON] [PublicData]

-- | a type to wrap our private api
type PrivateAPI = Get '[JSON] PrivateData

-- | our API
type BasicAPI = "public"  :> PublicAPI
           :<|> "private" :> BasicAuth "foo-realm" User :> PrivateAPI

-- | a value holding a proxy of our API type
basicAuthApi :: Proxy BasicAPI
basicAuthApi = Proxy
```

You can see that we've prefixed our public API with "public" and our private API with "private." Additionally, the private parts of our API use the `BasicAuth` combinator to protect them under a Basic Authentication scheme (the realm for this authentication is `"foo-realm"`).

Unfortunately we're not done. When someone makes a request to our `"private"` API, we're going to need to provide to servant the logic for validifying usernames and passwords. This adds a certain conceptual wrinkle in servant's design that we'll briefly discuss. If you want the **TL;DR**: we supply a lookup function to servant's new `Context` primitive.

Until now, all of servant's API combinators extracted information from a request or dictated the structure of a response (e.g. a `Capture` param is pulled from the request path). Now consider an API resource protected by basic authentication. Once the required `WWW-Authenticate` header is checked, we need to verify the username and password. But how? One solution would be to force an API author to provide a function of type `BasicAuthData -> Handler User` and servant should use this function to authenticate a request. Unfortunately this didn't work prior to `0.5` because all of servant's machinery was engineered around the idea that each combinator can extract information from only the request. We cannot extract the function `BasicAuthData -> Handler User` from a request! Are we doomed?

Servant `0.5` introduced `Context` to handle this. The type machinery is beyond the scope of this tutorial, but the idea is simple: provide some data to the `serve` function, and that data is propagated to the functions that handle each combinator. Using `Context`, we can supply a function of type `BasicAuthData -> Handler User` to the `BasicAuth` combinator handler. This will allow the handler to check authentication and return a `User` to downstream handlers if successful.

In practice we wrap `BasicAuthData -> Handler` into a slightly different function to better capture the semantics of basic authentication:

```
-- | The result of authentication/authorization
data BasicAuthResult usr
  = Unauthorized
  | BadPassword
  | NoSuchUser
  | Authorized usr
  deriving (Eq, Show, Read, Generic, Typeable, Functor)

-- | Datatype wrapping a function used to check authentication.
newtype BasicAuthCheck usr = BasicAuthCheck
  { unBasicAuthCheck :: BasicAuthData
                     -> IO (BasicAuthResult usr)
  }
  deriving (Generic, Typeable, Functor)
```

We now use this datatype to supply servant with a method to authenticate requests. In this simple example the only valid username and password is `"servant"` and `"server"`, respectively, but in a real, production application you might do some database lookup here.

```
-- | 'BasicAuthCheck' holds the handler we'll use to verify a username and password.
authCheck :: BasicAuthCheck User
authCheck =
  let check (BasicAuthData username password) =
        if username == "servant" && password == "server"
        then return (Authorized (User "servant"))
        else return Unauthorized
  in BasicAuthCheck check
```

And now we create the `Context` used by servant to find `BasicAuthCheck`:

```
-- | We need to supply our handlers with the right Context. In this case,
-- Basic Authentication requires a Context Entry with the 'BasicAuthCheck' value
```

```
-- tagged with "foo-tag" This context is then supplied to 'server' and threaded
-- to the BasicAuth HasServer handlers.
basicAuthServerContext :: Context (BasicAuthCheck User ': '[])
basicAuthServerContext = authCheck :. EmptyContext
```

We're now ready to write our `server` method that will tie everything together:

```
-- | an implementation of our server. Here is where we pass all the handlers to our endpoints.
-- In particular, for the BasicAuth protected handler, we need to supply a function
-- that takes 'User' as an argument.
basicAuthServer :: Server BasicAPI
basicAuthServer =
  let publicAPIHandler = return [PublicData "foo", PublicData "bar"]
      privateAPIHandler (user :: User) = return (PrivateData (userName user))
  in publicAPIHandler :<|> privateAPIHandler
```

Finally, our main method and a sample session working with our server:

```
-- | hello, server!
basicAuthMain :: IO ()
basicAuthMain = run 8080 (serveWithContext basicAuthApi
                                           basicAuthServerContext
                                           basicAuthServer
                         )

{- Sample session

$ curl -XGET localhost:8080/public
[{"somedata":"foo"},{"somedata":"bar"}

$ curl -iXGET localhost:8080/private
HTTP/1.1 401 Unauthorized
transfer-encoding: chunked
Date: Thu, 07 Jan 2016 22:36:38 GMT
Server: Warp/3.1.8
WWW-Authenticate: Basic realm="foo-realm"

$ curl -iXGET localhost:8080/private -H "Authorization: Basic c2VydmFudDpzZXJ2ZXI="
HTTP/1.1 200 OK
transfer-encoding: chunked
Date: Thu, 07 Jan 2016 22:37:58 GMT
Server: Warp/3.1.8
Content-Type: application/json
{"ssshhh":"servant"}
-}
```

Generalized Authentication

Sometimes your server's authentication scheme doesn't quite fit with the standards (or perhaps servant hasn't rolled-out support for that new, fancy authentication scheme). For such a scenario, servant `0.5` provides easy and simple experimental support to roll your own authentication.

Why experimental? We worked on the design for authentication for a long time. We really struggled to find a nice, type-safe niche in the design space. In fact, `Context` came out of this work, and while it really fit for schemes like Basic and JWT, it wasn't enough to fully support something like OAuth or HMAC, which have flows, roles, and other fancy ceremonies. Further, we weren't sure *how* people will use auth.

So, in typical startup fashion, we developed an MVP of 'generalized auth' and released it in an experimental module, with the hope of getting feedback from you! So, if you're reading this or using generalized auth support, please give us your feedback!

What is Generalized Authentication?

TL;DR: you throw a tagged `AuthProtect` combinator in front of the endpoints you want protected and then supply a function `Request -> Handler a`, where `a` is the type of your choice representing the data returned by successful authentication - e.g., a `User` or, in our example below, `Account`. This function is run anytime a request matches a protected endpoint. It precisely solves the "I just need to protect these endpoints with a function that does some complicated business logic" and nothing more. Behind the scenes we use a type family instance (`AuthServerData`) and `Context` to accomplish this.

Generalized Authentication in Action

Let's implement a trivial authentication scheme. We will protect our API by looking for a cookie named `"servant-auth-cookie"`. This cookie's value will contain a key from which we can lookup a `Account`.

```
-- | An account type that we "fetch from the database" after
-- performing authentication
newtype Account = Account { unAccount :: Text }

-- | A (pure) database mapping keys to accounts.
database :: Map ByteString Account
database = fromList [ ("key1", Account "Anne Briggs")
                    , ("key2", Account "Bruce Cockburn")
                    , ("key3", Account "Ghédalia Tazartès")
                    ]

-- | A method that, when given a password, will return a Account.
-- This is our bespoke (and bad) authentication logic.
lookupAccount :: ByteString -> Handler Account
lookupAccount key = case Map.lookup key database of
  Nothing -> throwError (err403 { errBody = "Invalid Cookie" })
  Just usr -> return usr
```

For generalized authentication, servant exposes the `AuthHandler` type, which is used to wrap the `Request -> Handler Account` logic. Let's create a value of type `AuthHandler Request Account` using the above `lookupAccount` method (note: we depend upon `cookie`'s `parseCookies` for this):

```
--- | The auth handler wraps a function from Request -> Handler Account.
--- We look for a token in the request headers that we expect to be in the cookie.
--- The token is then passed to our `lookupAccount` function.
authHandler :: AuthHandler Request Account
authHandler = mkAuthHandler handler
  where
  maybeToEither e = maybe (Left e) Right
  throw401 msg = throwError $ err401 { errBody = msg }
  handler req = either throw401 lookupAccount $ do
    cookie <- maybeToEither "Missing cookie header" $ lookup "cookie" $ requestHeaders req
    maybeToEither "Missing token in cookie" $ lookup "servant-auth-cookie" $ parseCookies cookie
```

Let's now protect our API with our new, bespoke authentication scheme. We'll re-use the endpoints from our Basic Authentication example.

```
-- | Our API, with auth-protection
type AuthGenAPI = "private" :> AuthProtect "cookie-auth" :> PrivateAPI
             :<|> "public"  :> PublicAPI

-- | A value holding our type-level API
genAuthAPI :: Proxy AuthGenAPI
genAuthAPI = Proxy
```

Now we need to bring everything together for the server. We have the `AuthHandler Request Account` value and an `AuthProtected` endpoint. To bind these together, we need to provide a Type Family instance that tells the `HasServer` instance that our `Context` will supply a `Account` (via `AuthHandler Request Account`) and that downstream combinators will have access to this `Account` value (or an error will be thrown if authentication fails).

```
-- | We need to specify the data returned after authentication
type instance AuthServerData (AuthProtect "cookie-auth") = Account
```

Note that we specify the type-level tag `"cookie-auth"` when defining the type family instance. This allows us to have multiple authentication schemes protecting a single API.

We now construct the `Context` for our server, allowing us to instantiate a value of type `Server AuthGenAPI`, in addition to the server value:

```
-- | The context that will be made available to request handlers. We supply the
-- "cookie-auth"-tagged request handler defined above, so that the 'HasServer' instance
-- of 'AuthProtect' can extract the handler and run it on the request.
genAuthServerContext :: Context (AuthHandler Request Account ': '[])
genAuthServerContext = authHandler :. EmptyContext

-- | Our API, where we provide all the author-supplied handlers for each end
-- point. Note that 'privateDataFunc' is a function that takes 'Account' as an
-- argument. We dont' worry about the authentication instrumentation here,
-- that is taken care of by supplying context
genAuthServer :: Server AuthGenAPI
genAuthServer =
  let privateDataFunc (Account name) =
          return (PrivateData ("this is a secret: " <> name))
      publicData = return [PublicData "this is a public piece of data"]
  in  privateDataFunc :<|> publicData
```

We're now ready to start our server (and provide a sample session)!

```
-- | run our server
genAuthMain :: IO ()
genAuthMain = run 8080 (serveWithContext genAuthAPI genAuthServerContext genAuthServer)

{- Sample Session:

$ curl -XGET localhost:8080/private
Missing auth header

$ curl -XGET localhost:8080/private -H "servant-auth-cookie: key3"
[{"ssshhh":"this is a secret: Ghédalia Tazartès"}]

$ curl -XGET localhost:8080/private -H "servant-auth-cookie: bad-key"
Invalid Cookie

$ curl -XGET localhost:8080/public
```

```
[{"somedata":"this is a public piece of data"}]
-}
```

Recap

Creating a generalized, ad-hoc authentication scheme was fairly straight forward:

1. use the `AuthProtect` combinator to protect your API.
2. choose a application-specific data type used by your server when authentication is successful (in our case this was `Account`).
3. Create a value of `AuthHandler Request Account` which encapsulates the authentication logic (`Request -> Handler Account`). This function will be executed everytime a request matches a protected route.
4. Provide an instance of the `AuthServerData` type family, specifying your application-specific data type returned when authentication is successful (in our case this was `Account`).

Caveats:

1. The module `Servant.Server.Experimental.Auth` contains an orphan `HasServer` instance for the `AuthProtect` combinator. You may be get orphan instance warnings when using this.
2. Generalized authentication requires the `UndecidableInstances` extension.

Client-side Authentication

Basic Authentication

As of `0.5`, *servant-client* comes with support for basic authentication! Endpoints protected by Basic Authentication will require a value of type `BasicAuthData` to complete the request.

You can find more comprehensive Basic Authentication example in the Cookbook.

Generalized Authentication

Servant `0.5` also shipped with support for generalized authentication. Similar to the server-side support, clients need to supply an instance of the `AuthClientData` type family specifying the datatype the client will use to marshal an unauthenticated request into an authenticated request. Generally, this will look like:

```
import           Servant.Common.Req                 (Req, addHeader)

-- | The datatype we'll use to authenticate a request. If we were wrapping
-- something like OAuth, this might be a Bearer token.
type instance AuthClientData (AuthProtect "cookie-auth") = String

-- | A method to authenticate a request
authenticateReq :: String -> Req -> Req
authenticateReq s req = addHeader "my-bespoke-header" s req
```

Now, if the client method for our protected endpoint was `getProtected`, then we could perform authenticated requests as follows:

```
-- | one could curry this to make it simpler to work with.
result = runExceptT (getProtected (mkAuthenticateReq "secret" authenticateReq))
```

CHAPTER 2

Cookbook

This page is a *collective effort* whose goal is to show how to solve many common problems with servant. If you're interested in contributing examples of your own, feel free to open an issue or a pull request on our github repository or even to just get in touch with us on the **#servant** IRC channel on freenode or on the mailing list.

The scope is very wide. Simple and fancy authentication schemes, file upload, type-safe links, working with CSV, .zip archives, you name it!

Structuring APIs

In this recipe, we will see a few simple ways to structure your APIs by splitting them up into smaller "sub-APIs" or by sharing common structure between different parts. Let's start with the usual throat clearing.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE KindSignatures #-}
import Data.Aeson
import GHC.Generics
import GHC.TypeLits
import Network.Wai.Handler.Warp
import Servant
```

Our application will consist of three different "sub-APIs", with a few endpoints in each of them. Our global API is defined as follows.

```
type API = FactoringAPI
      :<|> SimpleAPI "users" User UserId
      :<|> SimpleAPI "products" Product ProductId
```

We simply join the three different parts with `:<|>`, as if each sub-API was just a simple endpoint. The first part, `FactoringAPI`, shows how we can "factor out" combinators that are common to several endpoints, just like we turn `a * b + a * c` into `a * (b + c)` in algebra.

```
-- Two endpoints:
--   - GET /x/<some 'Int'>[?y=<some 'Int'>]
--   - POST /x/<some 'Int'>
type FactoringAPI =
  "x" :> Capture "x" Int :>
      (    QueryParam "y" Int :> Get '[JSON] Int
      :<|>                       Post '[JSON] Int
      )
```

```
{- this is equivalent to:

type FactoringAPI' =
  "x" :> Capture "x" Int :> QueryParam "y" Int :> Get '[JSON] Int :<|>
  "x" :> Capture "x" Int :> Post '[JSON] Int
-}
```

You can see that both endpoints start with a static path fragment, `/"x"`, then capture some arbitrary `Int` until they finally differ. Now, this also has an effect on the server for such an API, and its type in particular. While the server for `FactoringAPI'` would be made of a function of type `Int -> Maybe Int -> Handler Int` and a function of type `Int -> Handler Int` glued with `:<|>`, a server for `FactoringAPI` (without the `'`) reflects the "factorisation" and therefore, `Server FactoringAPI` is `Int -> (Maybe Int -> Handler Int :<|> Handler Int)`. That is, the server must be a function that takes an `Int` (the `Capture`) and returns two values glued with `:<|>`, one of type `Maybe Int -> Handler Int` and the other of type `Handler Int`. Let's provide such a server implementation, with those "nested types".

Tip: you can load this module in ghci and ask for the concrete type that `Server FactoringAPI` "resolves to" by typing `:kind! Server FactoringAPI`.

```
factoringServer :: Server FactoringAPI
factoringServer x = getXY :<|> postX

  where getXY Nothing  = return x
        getXY (Just y) = return (x + y)

        postX = return (x - 1)
```

If you want to avoid the "nested types" and the need to manually dispatch the arguments (like `x` above) to the different request handlers, and would just like to be able to declare the API type as above but pretending that the `Capture` is not factored out, that every combinator is "distributed" (i.e that all endpoints are specified like `FactoringAPI'` above), then you should look at `flatten` from the servant-flatten package.

Next come the two sub-APIs defined in terms of this `SimpleAPI` type, but with different parameters. That type is just a good old Haskell type synonym that abstracts away a pretty common structure in web services, where you have:

- one endpoint for listing a bunch of entities of some type
- one endpoint for accessing the entity with a given identifier
- one endpoint for creating a new entity

There are many variants on this theme (endpoints for deleting, paginated listings, etc). The simple definition below reproduces such a structure, but instead of picking concrete types for the entities and their identifiers, we simply let the user of the type decide, by making those types parameters of `SimpleAPI`. While we're at it, we'll put all our endpoints under a common prefix that we also take as a parameter.

```
-- Three endpoints:
--    - GET /<name>
--    - GET /<name>/<some 'i'>
--    - POST /<name>
type SimpleAPI (name :: Symbol) a i = name :>
  (                          Get '[JSON] [a]
  :<|> Capture "id" i     :> Get '[JSON] a
  :<|> ReqBody '[JSON] a :> Post '[JSON] NoContent
  )
```

`Symbol` is the kind of type-level strings, which is what servant uses for representing static path fragments. We can even provide a little helper function for creating a server for that API given one handler for each endpoint as arguments.

```
simpleServer
  :: Handler [a]
  -> (i -> Handler a)
  -> (a -> Handler NoContent)
  -> Server (SimpleAPI name a i)
simpleServer listAs getA postA =
  listAs :<|> getA :<|> postA

{- you could alternatively provide such a definition
   but with the handlers running in another monad,
   or even an arbitrary one!

simpleAPIServer
  :: m [a]
  -> (i -> m a)
  -> (a -> m NoContent)
  -> Server (SimpleAPI name a i) m
simpleAPIServer listAs getA postA =
  listAs :<|> getA :<|> postA

   and use 'hoistServer' on the result of `simpleAPIServer`
   applied to your handlers right before you call `serve`.
-}
```

We can use this to define servers for the user and product related sections of the API.

```
userServer :: Server (SimpleAPI "users" User UserId)
userServer = simpleServer
  (return [])
  (\userid -> return $
      if userid == 0
      then User "john" 64
      else User "everybody else" 10
  )
  (\_user -> return NoContent)

productServer :: Server (SimpleAPI "products" Product ProductId)
productServer = simpleServer
  (return [])
  (\_productid -> return $ Product "Great stuff")
  (\_product -> return NoContent)
```

Finally, some dummy types and the serving part.

```
type UserId = Int

data User = User { username :: String, age :: Int }
  deriving Generic

instance FromJSON User
instance ToJSON   User

type ProductId = Int

data Product = Product { productname :: String }
  deriving Generic

instance FromJSON Product
instance ToJSON   Product
```

```
api :: Proxy API
api = Proxy

main :: IO ()
main = run 8080 . serve api $
  factoringServer :<|> userServer :<|> productServer
```

This program is available as a cabal project here.

Using generics

```
{-# LANGUAGE DataKinds     #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE RankNTypes    #-}
{-# LANGUAGE TypeOperators #-}
module Main (main, api, getLink, routesLinks, cliGet) where

import Control.Exception         (throwIO)
import Data.Proxy                (Proxy (..))
import Network.Wai.Handler.Warp  (run)
import System.Environment        (getArgs)

import Servant
import Servant.Client

import Servant.API.Generic
import Servant.Client.Generic
import Servant.Server.Generic
```

The usage is simple, if you only need a collection of routes. First you define a record with field types prefixed by a parameter `route`:

```
data Routes route = Routes
    { _get :: route :- Capture "id" Int :> Get '[JSON] String
    , _put :: route :- ReqBody '[JSON] Int :> Put '[JSON] Bool
    }
  deriving (Generic)
```

Then we'll use this data type to define API, links, server and client.

API

You can get a `Proxy` of the API using `genericApi`:

```
api :: Proxy (ToServantApi Routes)
api = genericApi (Proxy :: Proxy Routes)
```

It's recommented to use `genericApi` function, as then you'll get better error message, for example if you forget to `derive Generic`.

Links

The clear advantage of record-based generics approach, is that we can get safe links very conviently. We don't need to define endpoint types, as field accessors work as proxies:

```
getLink :: Int -> Link
getLink = fieldLink _get
```

We can also get all links at once, as a record:

```
routesLinks :: Routes (AsLink Link)
routesLinks = allFieldLinks
```

Client

Even more power starts to show when we generate a record of client functions. Here we use `genericClientHoist` function, which let us simultaneously hoist the monad, in this case from `ClientM` to `IO`.

```
cliRoutes :: Routes (AsClientT IO)
cliRoutes = genericClientHoist
    (\x -> runClientM x env >>= either throwIO return)
  where
    env = error "undefined environment"

cliGet :: Int -> IO String
cliGet = _get cliRoutes
```

Server

Finally, probably the most handy usage: we can convert record of handlers into the server implementation:

```
record :: Routes AsServer
record = Routes
    { _get = return . show
    , _put = return . odd
    }

app :: Application
app = genericServe record

main :: IO ()
main = do
    args <- getArgs
    case args of
        ("run":_) -> do
            putStrLn "Starting cookbook-generic at http://localhost:8000"
            run 8000 app
        _ -> putStrLn "To run, pass 'run' argument: cabal new-run cookbook-generic run"
```

Serving web applications over HTTPS

This short recipe shows how one can serve a servant application over HTTPS, by simply using `warp-tls` instead of `warp` to provide us a `run` function for running the `Application` that we get by calling `serve`.

As usual, we start by clearing our throat of a few language extensions and imports.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
import Network.Wai
```

```
import Network.Wai.Handler.Warp
import Network.Wai.Handler.WarpTLS
import Servant
```

No need to work with a complicated API here, let's make it as simple as it gets:

```
type API = Get '[JSON] Int

api :: Proxy API
api = Proxy

server :: Server API
server = return 10

app :: Application
app = serve api server
```

It's now time to actually run the `Application`. The `warp-tls` package provides two functions for running an `Application`, called `runTLS` and `runTLSSocket`. We will be using the first one.

It takes two arguments, the TLS settings (certificates, keys, ciphers, etc) and the warp settings (port, logger, etc).

We will be using very simple settings for this example but you are of course invited to read the documentation for those types to find out about all the knobs that you can play with.

```
main :: IO ()
main = runTLS tlsOpts warpOpts app

  where tlsOpts = tlsSettings "cert.pem" "secret-key.pem"
        warpOpts = setPort 8080 defaultSettings
```

This program is available as a cabal project here.

SQLite database

Let's see how we can write a simple web application that uses an SQLite database to store simple textual messages. As usual, we start with a little bit of throat clearing.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeOperators #-}
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad.IO.Class
import Database.SQLite.Simple
import Network.HTTP.Client (newManager, defaultManagerSettings)
import Network.Wai.Handler.Warp
import Servant
import Servant.Client
```

We will only care about a single type here, the messages. We want to be able to add a new one and retrieve them all, using two different endpoints.

```
type Message = String

type API = ReqBody '[PlainText] Message :> Post '[JSON] NoContent
      :<|> Get '[JSON] [Message]
```

```
api :: Proxy API
api = Proxy
```

We proceed with a simple function for creating a table for holding our messages if it doesn't already exist.

```
initDB :: FilePath -> IO ()
initDB dbfile = withConnection dbfile $ \conn ->
  execute_ conn
    "CREATE TABLE IF NOT EXISTS messages (msg text not null)"
```

Next, our server implementation. It will be parametrised (take as an argument) by the name of the file that contains our SQLite database. The handlers are straightforward. One takes care of inserting a new value in the database while the other fetches all messages and returns them. We also provide a function for serving our web app given an SQLite database file, which simply calls servant-server's `serve` function.

```
server :: FilePath -> Server API
server dbfile = postMessage :<|> getMessages

  where postMessage :: Message -> Handler NoContent
        postMessage msg = do
          liftIO . withConnection dbfile $ \conn ->
            execute conn
                    "INSERT INTO messages VALUES (?)"
                    (Only msg)
          return NoContent

        getMessages :: Handler [Message]
        getMessages = fmap (map fromOnly) . liftIO $
         withConnection dbfile $ \conn ->
            query_ conn "SELECT msg FROM messages"

runApp :: FilePath -> IO ()
runApp dbfile = run 8080 (serve api $ server dbfile)
```

Let's also derive some clients for our API and use them to insert two messages and retrieve them in `main`.

```
postMsg :: Message -> ClientM NoContent
getMsgs :: ClientM [Message]
postMsg :<|> getMsgs = client api

main :: IO ()
main = do
  -- you could read this from some configuration file,
  -- environment variable or somewhere else instead.
  let dbfile = "test.db"
  initDB dbfile
  mgr <- newManager defaultManagerSettings
  bracket (forkIO $ runApp dbfile) killThread $ \_ -> do
    ms <- flip runClientM (mkClientEnv mgr (BaseUrl Http "localhost" 8080 "")) $ do
      postMsg "hello"
      postMsg "world"
      getMsgs
    print ms
```

This program prints `Right ["hello","world"]` the first time it is executed, `Right ["hello","world","hello","world"]` the second time and so on.

The entire source for this example is available as a cabal project here.

PostgreSQL connection pool

Let's see how we can write a simple web application that uses a PostgreSQL database to store simple textual messages, just like in the SQLite cookbook recipe. The main difference, besides the database technology, is that in this example we will be using a pool of connections to talk to the database server. The pool abstraction will be provided by the resource-pool library.

As usual, we start with a little bit of throat clearing.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeOperators #-}
import Data.ByteString (ByteString)
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad.IO.Class
import Data.Pool
import Database.PostgreSQL.Simple
import Network.HTTP.Client (newManager, defaultManagerSettings)
import Network.Wai.Handler.Warp
import Servant
import Servant.Client

type DBConnectionString = ByteString
```

We will only care about a single type here, the messages. We want to be able to add a new one and retrieve them all, using two different endpoints.

```
type Message = String

type API = ReqBody '[PlainText] Message :> Post '[JSON] NoContent
      :<|> Get '[JSON] [Message]

api :: Proxy API
api = Proxy
```

We proceed with a simple function for creating a table for holding our messages if it doesn't already exist, given a PostgreSQL connection string.

```
initDB :: DBConnectionString -> IO ()
initDB connstr = bracket (connectPostgreSQL connstr) close $ \conn -> do
  execute_ conn
    "CREATE TABLE IF NOT EXISTS messages (msg text not null)"
  return ()
```

Next, our server implementation. It will be parametrised (take as argument) by the pool of database connections that handlers can use to talk to the PostgreSQL database. The resource pool abstraction allows us to flexibly set up a whole bunch of PostgreSQL connections tailored to our needs and then to forget about it all by simply asking for a connection using `withResource`.

The handlers are straightforward. One takes care of inserting a new value in the database while the other fetches all messages and returns them. We also provide a function for serving our web app given a PostgreSQL connection pool, which simply calls servant-server's `serve` function.

```
server :: Pool Connection -> Server API
server conns = postMessage :<|> getMessages

  where postMessage :: Message -> Handler NoContent
        postMessage msg = do
```

```
          liftIO . withResource conns $ \conn ->
            execute conn
                    "INSERT INTO messages VALUES (?)"
                    (Only msg)
          return NoContent

        getMessages :: Handler [Message]
        getMessages = fmap (map fromOnly) . liftIO $
         withResource conns $ \conn ->
           query_ conn "SELECT msg FROM messages"

runApp :: Pool Connection -> IO ()
runApp conns = run 8080 (serve api $ server conns)
```

We will also need a function for initialising our connection pool. `resource-pool` is quite configurable, feel free to wander in its documentation to gain a better understanding of how it works and what the configuration knobs are. I will be using some dummy values in this example.

```
initConnectionPool :: DBConnectionString -> IO (Pool Connection)
initConnectionPool connStr =
  createPool (connectPostgreSQL connStr)
             close
             2 -- stripes
             60 -- unused connections are kept open for a minute
             10 -- max. 10 connections open per stripe
```

Let's finally derive some clients for our API and use them to insert two messages and retrieve them in `main`, after setting up our pool of database connections.

```
postMsg :: Message -> ClientM NoContent
getMsgs :: ClientM [Message]
postMsg :<|> getMsgs = client api

main :: IO ()
main = do
  -- you could read this from some configuration file,
  -- environment variable or somewhere else instead.
  -- you will need to either change this connection string OR
  -- set some environment variables (see
  -- https://www.postgresql.org/docs/9.5/static/libpq-envars.html)
  -- to point to a running PostgreSQL server for this example to work.
  let connStr = ""
  pool <- initConnectionPool connStr
  initDB connStr
  mgr <- newManager defaultManagerSettings
  bracket (forkIO $ runApp pool) killThread $ \_ -> do
    ms <- flip runClientM (mkClientEnv mgr (BaseUrl Http "localhost" 8080 "")) $ do
      postMsg "hello"
      postMsg "world"
      getMsgs
    print ms
```

This program prints `Right ["hello","world"]` the first time it is executed, `Right ["hello","world","hello","world"]` the second time and so on.

You could alternatively have the handlers live in `ReaderT (Pool Connection)` and access the pool using e.g `ask`, but this would be more complicated than simply taking the pool as argument.

The entire source for this example is available as a cabal project here.

Using a custom monad

In this section we will create and API for a book shelf without any backing DB storage. We will keep state in memory and share it between requests using `Reader` monad and `STM`.

We start with a pretty standard set of imports and definition of the model:

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

import           Control.Concurrent           (forkIO, killThread)
import           Control.Concurrent.STM.TVar  (TVar, newTVar, readTVar,
                                               writeTVar)
import           Control.Exception            (bracket)
import           Control.Monad.IO.Class       (liftIO)
import           Control.Monad.STM            (atomically)
import           Control.Monad.Trans.Reader   (ReaderT, ask, runReaderT)
import           Data.Aeson                   (FromJSON, ToJSON)
import           GHC.Generics                 (Generic)
import           Network.HTTP.Client          (defaultManagerSettings,
                                               newManager)
import           Network.Wai.Handler.Warp     (run)

import           Servant
import           Servant.Client

newtype Book = Book String deriving (Show, Generic)
instance ToJSON Book
instance FromJSON Book
```

Now, let's define the API for book storage. For the sake of simplicity we'll only have methods for getting all books and adding a new one.

```
type GetBooks = Get '[JSON] [Book]
type AddBook = ReqBody '[JSON] Book :> PostCreated '[JSON] Book
type BooksAPI = "books" :> (GetBooks :<|> AddBook)

api :: Proxy BooksAPI
api = Proxy
```

Next, we define the state and the monad to run our handlers

```
data State = State
  { books :: TVar [Book]
  }

type AppM = ReaderT State Handler
```

Note that we can't use `State` monad here, because state will not be shared between requests.

We can now define handlers in terms of `AppM`...

```
server :: ServerT BooksAPI AppM
server = getBooks :<|> addBook
  where getBooks :: AppM [Book]
        getBooks = do
          State{books = p} <- ask
          liftIO $ atomically $ readTVar p
```

```
        addBook :: Book -> AppM Book
        addBook book = do
          State{books = p} <- ask
          liftIO $ atomically $ readTVar p >>= writeTVar p . (book :)
          return book
```

...and transform `AppM` to `Handler` by simply using `runReaderT`

```
nt :: State -> AppM a -> Handler a
nt s x = runReaderT x s

app :: State -> Application
app s = serve api $ hoistServer api (nt s) server
```

Finally, we end up with the following program

```
main :: IO ()
main = do
  let port = 8080
  mgr <- newManager defaultManagerSettings
  initialBooks <- atomically $ newTVar []
  let runApp = run port $ app $ State initialBooks
  bracket (forkIO runApp) killThread $ \_ -> do
    let getBooksClient :<|> addBookClient = client api
    let printBooks = getBooksClient >>= liftIO . print
    _ <- flip runClientM (mkClientEnv mgr (BaseUrl Http "localhost" port "")) $ do
      _ <- printBooks
      _ <- addBookClient $ Book "Harry Potter and the Order of the Phoenix"
      _ <- printBooks
      _ <- addBookClient $ Book "To Kill a Mockingbird"
      _ <- printBooks
      _ <- addBookClient $ Book "The Picture of Dorian Gray"
      printBooks
    return ()
```

When run, it outputs the following:

```
Running cookbook-using-custom-monad...
[]
[Book "Harry Potter and the Order of the Phoenix"]
[Book "To Kill a Mockingbird",Book "Harry Potter and the Order of the Phoenix"]
[Book "The Picture of Dorian Gray",Book "To Kill a Mockingbird",Book "Harry Potter and the Order of t
```

Basic Authentication

Let's see a simple example of a web application with a single endpoint, protected by Basic Authentication.

First, some throat clearing.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeOperators #-}
import Control.Concurrent
import Control.Exception
import qualified Data.Map as Map
import qualified Data.Text as T
import Data.Text.Encoding (decodeUtf8)
```

```
import Network.HTTP.Client (newManager, defaultManagerSettings)
import Network.Wai.Handler.Warp
import Servant
import Servant.Client
```

We will be dealing with a very simple model of users, as shown below. Our "user database" will just be a map from usernames to full user details. For the sake of simplicity, it will just be read only but the same code could be used with mutable references, database connections, files and more in place of our `Map`.

```
type Username = T.Text
type Password = T.Text
type Website = T.Text

data User = User
  { user :: Username
  , pass :: Password
  , site :: Website
  } deriving (Eq, Show)

-- could be a postgres connection, a file, anything.
type UserDB = Map.Map Username User

-- create a "database" from a list of users
createUserDB :: [User] -> UserDB
createUserDB users = Map.fromList [ (user u, u) | u <- users ]

-- our test database
userDB :: UserDB
userDB = createUserDB
  [ User "john" "shhhh" "john.com"
  , User "foo" "bar" "foobar.net"
  ]
```

Our API will contain a single endpoint, returning the authenticated user's own website.

```
-- a 'GET /mysite' endpoint, protected by basic authentication
type API = BasicAuth "People's websites" User :> "mysite" :> Get '[JSON] Website

{- if there were more endpoints to be protected, one could write:
type API = BasicAuth "People's websites" User :>
    ( "foo" :> Get '[JSON] Foo
 :<|> "bar" :> Get '[JSON] Bar
    )
-}

api :: Proxy API
api = Proxy

server :: Server API
server usr = return (site usr)
```

In order to protect our endpoint (`"mysite" :> Get '[JSON] Website`), we simply drop the `BasicAuth` combinator in front of it. Its first parameter, `"People's websites"` in our example, is the realm, which is an arbitrary string identifying the protected resources. The second parameter, `User` in our example, corresponds to the type we want to use to represent authenticated users. It could be anything.

When using `BasicAuth` in an API, the server implementation "gets" an argument of the authenticated user type used with `BasicAuth`, `User` in our case, in the "corresponding spot". In this example, the server implementation simply returns the `site` field of the authenticated user. More realistic applications would have endpoints that take

other arguments and where a lot more logic would be implemented. But in a sense, `BasicAuth` adds an argument just like `Capture`, `QueryParam`, `ReqBody` and friends. But instead of performing some form of decoding logic behind the scenes, servant runs some "basic auth check" that the user provides.

In our case, we need access to our user database, so we simply take it as an argument. A more serious implementation would probably take a database connection or even a connection pool.

```
-- provided we are given a user database, we can supply
-- a function that checks the basic auth credentials
-- against our database.
checkBasicAuth :: UserDB -> BasicAuthCheck User
checkBasicAuth db = BasicAuthCheck $ \basicAuthData ->
  let username = decodeUtf8 (basicAuthUsername basicAuthData)
      password = decodeUtf8 (basicAuthPassword basicAuthData)
  in
  case Map.lookup username db of
    Nothing -> return NoSuchUser
    Just u  -> if pass u == password
               then return (Authorized u)
               else return BadPassword
```

This check simply looks up the user in the "database" and makes sure the right password was used. For reference, here are the definitions of `BasicAuthResult` and `BasicAuthCheck`:

```
-- | The result of authentication/authorization
data BasicAuthResult usr
  = Unauthorized
  | BadPassword
  | NoSuchUser
  | Authorized usr
  deriving (Eq, Show, Read, Generic, Typeable, Functor)

-- | Datatype wrapping a function used to check authentication.
newtype BasicAuthCheck usr = BasicAuthCheck
  { unBasicAuthCheck :: BasicAuthData
                     -> IO (BasicAuthResult usr)
  }
  deriving (Generic, Typeable, Functor)
```

This is all great, but how is our `BasicAuth` combinator supposed to know that it should use our `checkBasicAuth` from above? The answer is that it simply expects to find a `BasicAuthCheck` value for the right user type in the `Context` with which we serve the application, where `Context` is just servant's way to allow users to communicate some configuration of sorts to combinators. It is nothing more than an heterogeneous list and we can create a context with our auth check and run our application with it with the following code:

```
runApp :: UserDB -> IO ()
runApp db = run 8080 (serveWithContext api ctx server)

  where ctx = checkBasicAuth db :. EmptyContext
```

`ctx` above is just a context with one element, `checkBasicAuth db`, whose type is `BasicAuthCheck User`. In order to say that we want to serve our application using the supplied context, we just have to use `serveWithContext` in place of `serve`.

Finally, let's derive a client to this endpoint as well in order to see our server in action!

```
getSite :: BasicAuthData -> ClientM Website
getSite = client api

main :: IO ()
```

```
main = do
  mgr <- newManager defaultManagerSettings
  bracket (forkIO $ runApp userDB) killThread $ \_ ->
    runClientM (getSite u) (mkClientEnv mgr (BaseUrl Http "localhost" 8080 ""))
      >>= print

  where u = BasicAuthData "foo" "bar"
```

This program prints `Right "foobar.net"`, as expected. Feel free to change this code and see what happens when you specify credentials that are not in the database.

The entire program covered here is available as a literate Haskell file here, along with a `cabal` project.

Combining JWT-based authentication with basic access authentication

In this example we will make a service with basic HTTP authentication for Haskell clients and other programs, as well as with JWT-based authentication for web browsers. Web browsers will still use basic HTTP authentication to retrieve JWTs though.

Warning: this is insecure when done over plain HTTP, so TLS should be used. See warp-tls for that.

While basic authentication comes with Servant itself, servant-auth and servant-auth-server packages are needed for the JWT-based one.

This recipe uses the following ingredients:

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, DataKinds,
  DeriveGeneric, TypeOperators #-}
import Data.Aeson
import GHC.Generics
import Data.Proxy
import System.IO
import Network.HTTP.Client (newManager, defaultManagerSettings)
import Network.Wai.Handler.Warp
import Servant as S
import Servant.Client
import Servant.Auth as SA
import Servant.Auth.Server as SAS
import Control.Monad.IO.Class (liftIO)
import Data.Map as M
import Data.ByteString (ByteString)

port :: Int
port = 3001
```

Authentication

Below is how we'll represent a user: usually user identifier is handy to keep around, along with their role if role-based access control is used, and other commonly needed information, such as an organization identifier:

```
data AuthenticatedUser = AUser { auID :: Int
                               , auOrgID :: Int
                               } deriving (Show, Generic)
```

The following instances are needed for JWT:

```
instance ToJSON AuthenticatedUser
instance FromJSON AuthenticatedUser
instance ToJWT AuthenticatedUser
instance FromJWT AuthenticatedUser
```

We'll have to use a bit of imagination to pretend that the following `Map` is a database connection pool:

```
type Login      = ByteString
type Password   = ByteString
type DB         = Map (Login, Password) AuthenticatedUser
type Connection = DB
type Pool a     = a

initConnPool :: IO (Pool Connection)
initConnPool = pure $ fromList [ (("user", "pass"), AUser 1 1)
                               , (("user2", "pass2"), AUser 2 1) ]
```

See the "PostgreSQL connection pool" recipe for actual connection pooling, and we proceed to an authentication function that would use our improvised DB connection pool and credentials provided by a user:

```
authCheck :: Pool Connection
          -> BasicAuthData
          -> IO (AuthResult AuthenticatedUser)
authCheck connPool (BasicAuthData login password) = pure $
  maybe SAS.Indefinite Authenticated $ M.lookup (login, password) connPool
```

Warning: make sure to use a proper password hashing function in functions like this: see bcrypt, scrypt, pgcrypto.

Unlike `Servant.BasicAuth`, `Servant.Auth` uses `FromBasicAuthData` type class for the authentication process itself. But since our connection pool will be initialized elsewhere, we'll have to pass it somehow: it can be done via a context entry and `BasicAuthCfg` type family. We can actually pass a function at once, to make it a bit more generic:

```
type instance BasicAuthCfg = BasicAuthData -> IO (AuthResult AuthenticatedUser)

instance FromBasicAuthData AuthenticatedUser where
  fromBasicAuthData authData authCheckFunction = authCheckFunction authData
```

API

Test API with a couple of endpoints:

```
type TestAPI = "foo" :> Capture "i" Int :> Get '[JSON] ()
               :<|> "bar" :> Get '[JSON] ()
```

We'll use this for server-side functions, listing the allowed authentication methods using the `Auth` combinator:

```
type TestAPIServer =
  Auth '[SA.JWT, SA.BasicAuth] AuthenticatedUser :> TestAPI
```

But `Servant.Auth.Client` only supports JWT-based authentication, so we'll have to use regular `Servant.BasicAuth` to derive client functions that use basic access authentication:

```
type TestAPIClient = S.BasicAuth "test" AuthenticatedUser :> TestAPI
```

Client

Client code in this setting is the same as it would be with just `Servant.BasicAuth`, using servant-client:

```
testClient :: IO ()
testClient = do
  mgr <- newManager defaultManagerSettings
  let (foo :<|> _) = client (Proxy :: Proxy TestAPIClient)
                            (BasicAuthData "name" "pass")
  res <- runClientM (foo 42)
    (mkClientEnv mgr (BaseUrl Http "localhost" port ""))
  hPutStrLn stderr $ case res of
    Left err -> "Error: " ++ show err
    Right r -> "Success: " ++ show r
```

Server

Server code is slightly different – we're getting `AuthResult` here:

```
server :: Server TestAPIServer
server (Authenticated user) = handleFoo :<|> handleBar
  where
    handleFoo :: Int -> Handler ()
    handleFoo n = liftIO $ hPutStrLn stderr $
      concat ["foo: ", show user, " / ", show n]
    handleBar :: Handler ()
    handleBar = liftIO testClient
```

Catch-all for `BadPassword`, `NoSuchUser`, and `Indefinite`:

```
server _ = throwAll err401
```

With `Servant.Auth`, we'll have to put both `CookieSettings` and `JWTSettings` into context even if we're not using those, and we'll put a partially applied `authCheck` function there as well, so that `FromBasicAuthData` will be able to use it, while it will use our connection pool. Otherwise it is similar to the usual way:

```
mkApp :: Pool Connection -> IO Application
mkApp connPool = do
  myKey <- generateKey
  let jwtCfg = defaultJWTSettings myKey
      authCfg = authCheck connPool
      cfg = jwtCfg :. defaultCookieSettings :. authCfg :. EmptyContext
      api = Proxy :: Proxy TestAPIServer
  pure $ serveWithContext api cfg server
```

Finally, the main function:

```
main :: IO ()
main = do
  connPool <- initConnPool
  let settings =
        setPort port $
        setBeforeMainLoop (hPutStrLn stderr
                           ("listening on port " ++ show port)) $
        defaultSettings
  runSettings settings =<< mkApp connPool
```

Usage

Now we can try it out with `curl`. First of all, let's ensure that it fails with `err401` if we're not authenticated:

```
$ curl -v 'http://localhost:3001/bar'
...
< HTTP/1.1 401 Unauthorized
```

```
$ curl -v 'http://user:wrong_password@localhost:3001/bar'
...
< HTTP/1.1 401 Unauthorized
```

Now let's see that basic HTTP authentication works, and that we get JWTs:

```
$ curl -v 'http://user:pass@localhost:3001/bar'
...
< HTTP/1.1 200 OK
...
< Set-Cookie: XSRF-TOKEN=lQE/sb1fW4rZ/FYUQZskI6RVRllG0CWZrQ0d3fXU4X0=; Path=/; Secure
< Set-Cookie: JWT-Cookie=eyJhbGciOiJIUzUxMiJ9.eyJkYXQiOnsiYXVPcmdJRCI6MSwiYXVJRCI6MX19.6ZQba-Co5Ul4wp
```

And authenticate using JWTs alone, using the token from `JWT-Cookie`:

```
curl -v -H 'Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJkYXQiOnsiYXVPcmdJRCI6MSwiYXVJRCI6MX19.6ZQba
...
< HTTP/1.1 200 OK
```

This program is available as a cabal project here.

File Upload (`multipart/form-data`)

In this recipe, we will implement a web application with a single endpoint that can process `multipart/form-data` request bodies, which most commonly come from HTML forms that allow file upload.

As usual, a bit of throat clearing.

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

import Control.Concurrent
import Control.Exception
import Control.Monad
import Control.Monad.IO.Class
import Data.Text.Encoding (encodeUtf8)
import Network (withSocketsDo)
import Network.HTTP.Client hiding (Proxy)
import Network.HTTP.Client.MultipartFormData
import Network.Wai.Handler.Warp
import Servant
import Servant.Multipart

import qualified Data.ByteString.Lazy as LBS
```

Our API consists in a single `POST` endpoint at `/` that takes a `multipart/form-data` request body and pretty-prints the data it got to stdout before returning `0` (because why not).

```
type API = MultipartForm Mem (MultipartData Mem) :> Post '[JSON] Integer

api :: Proxy API
api = Proxy
```

Because of some technicalities, multipart form data is not represented as a good old content type like `JSON` in servant, that one could use with `ReqBody`, but instead is its own dedicated `ReqBody`-like combinator named `MultiPartForm`.

This combinator takes two parameters. The first one is the "backend" to use. Currently, you only have the choice between `Mem` and `Tmp`. The former loads the entire input in memory, even the uploaded files, while `Tmp` will stream uploaded files to some temporary directory.

The second parameter is the type you want the multipart data to be decoded to. Indeed there is a `FromMultipart` class that allows you to specify how to decode multipart form data from `MultipartData` to a custom type of yours. Here we use the trivial "decoding" to `MultipartData` itself, and simply will get our hands on the raw input. If you want to use a type of yours, see the documentation for `FromMultipart`.

Our only request handler has type `MultipartData Mem -> Handler Integer`. All it does is list the textual and file inputs that were sent in the multipart request body. The textual inputs are in the `inputs` field while the file inputs are in the `files` field of `multipartData`.

```
-- MultipartData consists in textual inputs,
-- accessible through its "inputs" field, as well
-- as files, accessible through its "files" field.
upload :: Server API
upload multipartData = do
  liftIO $ do
    putStrLn "Inputs:"
    forM_ (inputs multipartData) $ \input ->
      putStrLn $ "  " ++ show (iName input)
            ++ " -> " ++ show (iValue input)

    forM_ (files multipartData) $ \file -> do
      let content = fdPayload file
      putStrLn $ "Content of " ++ show (fdFileName file)
      LBS.putStr content
  return 0

startServer :: IO ()
startServer = run 8080 (serve api upload)
```

Finally, a main function that brings up our server and sends some test request with `http-client` (and not servant-client this time, has servant-multipart does not yet have support for client generation.

```
main :: IO ()
main = withSocketsDo . bracket (forkIO startServer) killThread $ \_threadid -> do
  -- we fork the server in a separate thread and send a test
  -- request to it from the main thread.
  manager <- newManager defaultManagerSettings
  req <- parseRequest "http://localhost:8080/"
  resp <- flip httpLbs manager =<< formDataBody form req
  print resp

  where form =
          [ partBS "title" "World"
          , partBS "text" $ encodeUtf8 "Hello"
          , partFileSource "file" "./README.md"
          ]
```

If you run this, you should get:

```
$ cabal new-build cookbook-file-upload
[...]
$ dist-newstyle/build/x86_64-linux/ghc-8.2.1/cookbook-file-upload-0.1/x/cookbook-file-upload/build/co
Inputs:
  "title" -> "World"
  "text" -> "Hello"
Content of "README.md"
# servant - A Type-Level Web DSL

![servant](https://raw.githubusercontent.com/haskell-servant/servant/master/servant.png)

## Getting Started

We have a [tutorial](http://haskell-servant.readthedocs.org/en/stable/tutorial/index.html) that
introduces the core features of servant. After this article, you should be able
to write your first servant webservices, learning the rest from the haddocks'
examples.

[...]

Response {responseStatus = Status {statusCode = 200, statusMessage = "OK"}, responseVersion = HTTP/1.
```

As usual, the code for this recipe is available in a cabal project here.

Pagination

Overview

Let's see an approach to typed pagination with *Servant* using servant-pagination.

This module offers opinionated helpers to declare a type-safe and a flexible pagination mechanism for Servant APIs. This design, inspired by Heroku's API, provides a small framework to communicate about a possible pagination feature of an endpoint, enabling a client to consume the API in different fashions (pagination with offset / limit, endless scroll using last referenced resources, ascending and descending ordering, etc.)

Therefore, client can provide a `Range` header with their request with the following format:

- `Range: <field> [<value>][; offset <o>][; limit <l>][; order <asc|desc>]`

For example: `Range: createdAt 2017-01-15T23:14:67.000Z; offset 5; order desc` indicates that the client is willing to retrieve the next batch of document in descending order that were created after the fifteenth of January, skipping the first 5.

As a response, the server may return the list of corresponding document, and augment the response with 3 headers:

- `Accept-Ranges`: A comma-separated list of fields upon which a range can be defined
- `Content-Range`: Actual range corresponding to the content being returned
- `Next-Range`: Indicate what should be the next `Range` header in order to retrieve the next range

For example:

- `Accept-Ranges: createdAt, modifiedAt`
- `Content-Range: createdAt 2017-01-15T23:14:51.000Z..2017-02-18T06:10:23.000Z`

- `Next-Range: createdAt 2017-02-19T12:56:28.000Z; offset 0; limit 100; order desc`

Getting Started

Code-wise the integration is quite seamless and unobtrusive. `servant-pagination` provides a `Ranges (fields :: [Symbol]) (resource :: *) -> *` data-type for declaring available ranges on a group of *fields* and a target *resource*. To each combination (resource + field) is associated a given type `RangeType (resource :: *) (field :: Symbol) -> *` as described by the type-family in the `HasPagination` type-class.

So, let's start with some imports and extensions to get this out of the way:

```
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE DeriveGeneric         #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeApplications      #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE TypeOperators         #-}

import            Data.Aeson
                  (ToJSON, genericToJSON)
import            Data.Maybe
                  (fromMaybe)
import            Data.Proxy
                  (Proxy (..))
import            GHC.Generics
                  (Generic)
import            Servant
                  ((:>), GetPartialContent, Handler, Header, Headers, JSON, Server, addHeader)
import            Servant.Pagination
                  (HasPagination (..), PageHeaders, Range (..), Ranges, RangeOptions(..),
                   applyRange, extractRange, returnRange)

import qualified Data.Aeson               as Aeson
import qualified Network.Wai.Handler.Warp as Warp
import qualified Servant
import qualified Servant.Pagination       as Pagination
```

Declaring the Resource

Servant APIs are rather resource-oriented, and so is `servant-pagination`. This guide shows a basic example working with `JSON` (as you could tell from the import list already). To make the world a better colored place, let's create an API to retrieve colors – with pagination.

```
data Color = Color
  { name :: String
  , rgb  :: [Int]
  , hex  :: String
  } deriving (Eq, Show, Generic)

instance ToJSON Color where
  toJSON =
    genericToJSON Aeson.defaultOptions
```

```
colors :: [Color]
colors =
  [ Color "Black" [0, 0, 0] "#000000"
  , Color "Blue" [0, 0, 255] "#0000ff"
  , Color "Green" [0, 128, 0] "#008000"
  , Color "Grey" [128, 128, 128] "#808080"
  , Color "Purple" [128, 0, 128] "#800080"
  , Color "Red" [255, 0, 0] "#ff0000"
  , Color "Yellow" [255, 255, 0] "#ffff00"
  ]
```

Declaring the Ranges

Now that we have defined our *resource* (a.k.a `Color`), we are ready to declare a new `Range` that will operate on a "name" field (genuinely named after the `name` fields from the `Color` record). For that, we need to tell `servant-pagination` two things:

- What is the type of the corresponding `Range` values
- How do we get one of these values from our resource

This is done via defining an instance of `HasPagination` as follows:

```
instance HasPagination Color "name" where
  type RangeType Color "name" = String
  getFieldValue _ = name
  -- getRangeOptions :: Proxy "name" -> Proxy Color -> RangeOptions
  -- getDefaultRange :: Proxy Color  -> Range "name" String

defaultRange :: Range "name" String
defaultRange =
  getDefaultRange (Proxy @Color)
```

Note that `getFieldValue :: Proxy "name" -> Color -> String` is the minimal complete definintion of the class. Yet, you can define `getRangeOptions` to provide different parsing options (see the last section of this guide). In the meantime, we've also defined a `defaultRange` as it will come in handy when defining our handler.

API

Good, we have a resource, we have a `Range` working on that resource, we can now declare our API using other Servant combinators we already know:

```
type API =
  "colors"
    :> Header "Range" (Ranges '["name"] Color)
    :> GetPartialContent '[JSON] (Headers MyHeaders [Color])

type MyHeaders =
  Header "Total-Count" Int ': PageHeaders '["name"] Color
```

`PageHeaders` is a type alias provided by the library to declare the necessary response headers we mentionned in introduction. Expanding the alias boils down to the following:

```
-- type MyHeaders =
--   '[ Header "Total-Count"   Int
--    , Header "Accept-Ranges" (AcceptRanges '["name"])
```

```
--   , Header "Content-Range" (ContentRange '["name"] Color)
--   , Header "Next-Range"    (Ranges '["name"] Color)
--   ]
```

As a result, we will need to provide all those headers with the response in our handler. Worry not, *servant-pagination* provides an easy way to lift a collection of resources into such handler.

Server

Time to connect the last bits by defining the server implementation of our colorful API. The `Ranges` type we've defined above (tight to the `Range` HTTP header) indicates the server to parse any `Range` header, looking for the format defined in introduction with fields and target types we have just declared. If no such header is provided, we will end up receiving `Nothing`. Otherwise, it will be possible to *extract* a `Range` from our `Ranges`.

```
server :: Server API
server = handler
  where
    handler :: Maybe (Ranges '["name"] Color) -> Handler (Headers MyHeaders [Color])
    handler mrange = do
      let range =
            fromMaybe defaultRange (mrange >>= extractRange)

      addHeader (length colors) <$> returnRange range (applyRange range colors)

main :: IO ()
main =
  Warp.run 1442 $ Servant.serve (Proxy @API) server
```

Let's try it out using different ranges to observe the server's behavior. As a reminder, here's the format we defined, where `<field>` here can only be `name` and `<value>` must parse to a `String`:

- `Range: <field> [<value>][; offset <o>][; limit <l>][; order <asc|desc>]`

Beside the target field, everything is pretty much optional in the `Range` HTTP header. Missing parts are deducted from the `RangeOptions` that are part of the `HasPagination` instance. Therefore, all following examples are valid requests to send to our server:

- 1 - `curl http://localhost:1442/colors -vH 'Range: name'`
- 2 - `curl http://localhost:1442/colors -vH 'Range: name; limit 2'`
- 3 - `curl http://localhost:1442/colors -vH 'Range: name Green; order asc; offset 1'`

Considering the following default options:

- `defaultRangeLimit: 100`
- `defaultRangeOffset: 0`
- `defaultRangeOrder: RangeDesc`

The previous ranges reads as follows:

- 1 - The first 100 colors, ordered by descending names
- 2 - The first 2 colors, ordered by descending names
- 3 - The 100 colors after `Green` (not included), ordered by ascending names.

Going Forward

Multiple Ranges

Note that in the simple above scenario, there's no ambiguity with `extractRange` and `returnRange` because there's only one possible `Range` defined on our resource. Yet, as you've most probably noticed, the `Ranges` combinator accepts a list of fields, each of which must declare a `HasPagination` instance. Doing so will make the other helper functions more ambiguous and type annotation are highly likely to be needed.

```
instance HasPagination Color "hex" where
  type RangeType Color "hex" = String
  getFieldValue _ = hex

-- to then define: Ranges '["name", "hex"] Color
```

Parsing Options

By default, `servant-pagination` provides an implementation of `getRangeOptions` for each `HasPagination` instance. However, this can be overwritten when defining the instance to provide your own options. This options come into play when a `Range` header is received and isn't fully specified (`limit`, `offset`, `order` are all optional) to provide default fallback values for those.

For instance, let's say we wanted to change the default limit to `5` in a new range on `"rgb"`, we could tweak the corresponding `HasPagination` instance as follows:

```
instance HasPagination Color "rgb" where
  type RangeType Color "rgb" = Int
  getFieldValue _ = sum . rgb
  getRangeOptions _ _ = Pagination.defaultOptions { defaultRangeLimit = 5 }
```

CHAPTER 3

Example Projects

- **example-servant-minimal**:

  A minimal example for a web server written using **servant-server**, including a test-suite using **hspec** and **servant-client**.

- **servant-examples**:

  Similar to the cookbook but with no explanations, for developers who just want to look at code examples to find out how to do X or Y with servant.

- **stack-templates**

  Repository for templates for haskell projects, including some templates using **servant**. These templates can be used with `stack new`.

- **custom-monad**:

  A custom monad that can replace `IO` in servant applications. It adds among other things logging functionality and a reader monad (for database connections). A full usage example of servant/diener is also provided.

- **example-servant-elm**:

  An example for a project consisting of

  - a backend web server written using **servant-server**,
  - a frontend written in elm using servant-elm to generate client functions in elm for the API,
  - test-suites for both the backend and the frontend.

- **servant-purescript**:

  An example consisting of

  - a backend in uses `haskell-servant`
  - a frontend written in PureScript using servant-purescript to generate an API wrapper in PureScript to interface the web API with

- **example-servant-persistent**:

  An example for a web server written with **servant-server** and persistent for writing data into a database.

CHAPTER 4

Helpful Links

- the central documentation (this site): haskell-servant.readthedocs.org
- the github repo: github.com/haskell-servant/servant
- the issue tracker (Feel free to create issues and submit PRs!): https://github.com/haskell-servant/servant/issues
- the irc channel: `#servant` on freenode
- the mailing list: groups.google.com/forum/#!forum/haskell-servant
- blog posts and videos and slides of some talks on servant: haskell-servant.github.io
- the servant packages on hackage:
  - hackage.haskell.org/package/servant
  - hackage.haskell.org/package/servant-server
  - hackage.haskell.org/package/servant-client
  - hackage.haskell.org/package/servant-blaze
  - hackage.haskell.org/package/servant-lucid
  - hackage.haskell.org/package/servant-cassava
  - hackage.haskell.org/package/servant-docs
  - hackage.haskell.org/package/servant-foreign
  - hackage.haskell.org/package/servant-js
  - hackage.haskell.org/package/servant-mock

CHAPTER 5

Principles

servant has the following guiding principles:

- concision

  This is a pretty wide-ranging principle. You should be able to get nice documentation for your web servers, and client libraries, without repeating yourself. You should not have to manually serialize and deserialize your resources, but only declare how to do those things *once per type*. If a bunch of your handlers take the same query parameters, you shouldn't have to repeat that logic for each handler, but instead just "apply" it to all of them at once. Your handlers shouldn't be where composition goes to die. And so on.

- flexibility

  If we haven't thought of your use case, it should still be easily achievable. If you want to use templating library X, go ahead. Forms? Do them however you want, but without difficulty. We're not opinionated.

- separation of concerns

  Your handlers and your HTTP logic should be separate. True to the philosophy at the core of HTTP and REST, with **servant** your handlers return normal Haskell datatypes - that's the resource. And then from a description of your API, **servant** handles the *presentation* (i.e., the Content-Types). But that's just one example.

- type safety

  Want to be sure your API meets a specification? Your compiler can check that for you. Links you can be sure exist? You got it.

To stick true to these principles, we do things a little differently than you might expect. The core idea is *reifying the description of your API*. Once reified, everything follows. We think we might be the first web framework to reify API descriptions in an extensible way. We're pretty sure we're the first to reify it as *types*.