
servant Documentation

Release

Servant Contributors

March 21, 2016

1	Introduction	3
2	Tutorial	5
2.1	A web API as a type	5
2.2	Serving an API	10
2.3	Querying an API	25
2.4	Generating Javascript functions to query an API	28
2.5	Documenting an API	31
3	Helpful Links	37

servant is a set of packages for declaring web APIs at the type-level and then using those API specifications to:

- write servers (this part of **servant** can be considered a web framework),
- obtain client functions (in haskell),
- generate client functions for other programming languages,
- generate documentation for your web applications
- and more...

All in a type-safe manner.

Introduction

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*.

This is an introductory tutorial to **servant**.

(Any comments, issues or feedback about the tutorial can be handled through [servant's issue tracker](#).)

2.1 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 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
}
```

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.

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 the combinators that **servant** comes with.

2.1.1 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]
```

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 `FromText` 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¶m=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 (contentType :: [*]) 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 Authentictaion 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 ProtectedAPI12
  = UserAPI -- this is public
  :<|> BasicAuth "my-real" User :> UserAPI2 -- this is protected by auth
```

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 UserAPI11 = "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`.

2.2 Serving an API

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

2.2.1 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 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 web servers 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": "1683-03-01"
, { "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)
  ]
```

Let's also write our API type.

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

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 `ExceptT ServantErr IO` 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 `ExceptT ServantErr IO [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"}, {"e
```

2.2.2 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`.

2.2.3 From combinators to handler arguments

Fine, we can write trivial webservice 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 -> ExceptT ServantErr IO Position
        position x y = return (Position x y)

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

        marketing :: ClientInfo -> ExceptT ServantErr IO 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 technically 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}' http://localhost:8081/hello
{"subject":"Hey Alp Mestanogullari, we miss you!","body":"Hi Alp Mestanogullari,\n\nSince you've received this message, you have been added to our mailing list. You will receive our newsletter every two weeks."}
```

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 `ExceptT ServantErr IO <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" a` gets turned into an argument of type `Bool`.
- `QueryParams "something" a` gets turned into an argument of type `[a]`.
- `ReqBody contentTypeList a` gets turned into an argument of type `a`.

2.2.4 The `FromHttpApiData/ToHttpApiData` classes

Wait... How does **servant** know how to decode the `Ints` 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.

2.2.5 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>A
# or just point your browser to http://localhost:8081/persons
```

2.2.6 The `ExceptT ServantErr IO` monad

At the heart of the handlers is the monad they run in, namely `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 `ExceptT ServantErr IO 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` is an instance 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 instance from the list above is `MonadIO m => MonadIO (ExceptT e m)`. `MonadIO` is a class from the `transformers` package defined as:

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

The IO monad provides a `MonadIO` instance. Hence for any type `e`, `ExceptT e IO` has a `MonadIO` instance. 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 :: ExceptT ServantErr IO ()
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 alone" }
```

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"}
```

2.2.7 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 x` is different than the type of `x`!

2.2.8 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
serveDirectory :: FilePath -> Server Raw
```

`serveDirectory`'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 = serveDirectory "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.

2.2.9 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 -> Delete '[] ()
```

We can instead factor out the `userid`:

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

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

```
Server UserAPI3 = (Int -> ExceptT ServantErr IO User)
  :<|> (Int -> ExceptT ServantErr IO ())

Server UserAPI4 = Int -> (   ExceptT ServantErr IO User
  :<|> ExceptT ServantErr IO ()
  )
```

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 `ExceptT`, with no arguments. In other words:

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

  where getUser :: Int -> ExceptT ServantErr IO User
        getUser _userid = error "..."
```

deleteUser :: Int -> ExceptT ServantErr IO ()
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 = getUser :<|> deleteUser

  where getUser :: Int -> ExceptT ServantErr IO User
        getUser = error "..."
```

```
deleteUser :: Int -> ExceptT ServantErr IO ()
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
  :<|> Post '[JSON] () -- 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 :> Post '[] () -- add some secret data, if authorized
  )

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 :> Post '[] () -- add a user
  :<|> Capture "userid" Int :>
    ( Get '[JSON] User -- view a user
    :<|> ReqBody '[JSON] User :> Put '[] () -- update a user
    :<|> Delete '[] () -- delete a user
    )

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

where getUsers :: ExceptT ServantErr IO [User]
      getUsers = error "...

      newUser :: User -> ExceptT ServantErr IO ()
      newUser = error "...

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

      where
        viewUser :: Int -> ExceptT ServantErr IO User
        viewUser = error "...

        updateUser :: Int -> User -> ExceptT ServantErr IO ()
        updateUser = error "...

        deleteUser :: Int -> ExceptT ServantErr IO ()
        deleteUser = error "..."
```

```

type ProductsAPI =
  Get '[JSON] [Product] -- list products
  :<|> ReqBody '[JSON] Product :> Post '[] () -- add a product
  :<|> Capture "productid" Int :>
    ( Get '[JSON] Product -- view a product
    :<|> ReqBody '[JSON] Product :> Put '[] () -- update a product
    :<|> Delete '[] () -- delete a product
    )

data Product = Product { productId :: Int }

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

where getProducts :: ExceptT ServantErr IO [Product]
      getProducts = error "..."

      newProduct :: Product -> ExceptT ServantErr IO ()
      newProduct = error "..."

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

      where
        viewProduct :: Int -> ExceptT ServantErr IO Product
        viewProduct = error "..."

        updateProduct :: Int -> Product -> ExceptT ServantErr IO ()
        updateProduct = error "..."

        deleteProduct :: Int -> ExceptT ServantErr IO ()
        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 :> Post '[] () -- add an 'a'
  :<|> Capture "id" i :>
    ( Get '[JSON] a -- view an 'a' given its "identifier" of type 'i'
    :<|> ReqBody '[JSON] a :> Put '[] () -- update an 'a'
    :<|> Delete '[] () -- delete an 'a'
    )

-- Build the appropriate 'Server'
-- given the handlers of the right type.
serverFor :: ExceptT ServantErr IO [a] -- handler for listing of 'a's
          -> (a -> ExceptT ServantErr IO ()) -- handler for adding an 'a'
          -> (i -> ExceptT ServantErr IO a) -- handler for viewing an 'a' given its identifier of type 'i'
          -> (i -> a -> ExceptT ServantErr IO ()) -- updating an 'a' with given id
```

```

-> (i -> ExceptT ServantErr IO ()) -- 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!

```

2.2.10 Using another monad for your handlers

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

```
type Server api = ServerT api (ExceptT ServantErr IO)
```

`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?

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

For example:

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

(`Nat` comes from “natural transformation”, in case you're wondering.)

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

```
readerToHandler :: Reader String :~> ExceptT ServantErr IO
```

Let's start with `readerToHandler'`. 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 `ExceptT`. We can then just wrap that function with the `Nat` constructor to make it have the fancier type.

```
readerToHandler' :: forall a. Reader String a -> ExceptT ServantErr IO a
readerToHandler' r = return (runReader r "hi")

readerToHandler :: Reader String :~> ExceptT ServantErr IO
readerToHandler = Nat readerToHandler'
```

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

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

readerAPI :: Proxy ReaderAPI
readerAPI = Proxy
```

```

readerServerT :: ServerT ReaderAPI (Reader String)
readerServerT = a :<|> b

  where a :: Reader String Int
        a = return 1797

        b :: Reader String String
        b = ask

```

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

Enter `enter`

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 `enter` which takes a natural transformation between two parametrized 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 `enter readerToHandler` on our handlers.

```

readerServer :: Server ReaderAPI
readerServer = enter 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"

```

2.2.11 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**.

2.3 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. Except that 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 a bunch of 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. 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 Control.Monad.Trans.Except
import Data.Aeson
import Data.Proxy
import GHC.Generics
import Network.HTTP.Client (Manager, newManager, defaultManagerSettings)
import Servant.API
import Servant.Client
import System.IO.Unsafe

```

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

```

data Position = Position
  { x :: Int
  , y :: 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 3 functions, one to query each endpoint:

```

position :: Int -- ^ value for "x"
         -> Int -- ^ value for "y"
         -> ExceptT ServantError IO Position

```

```
hello :: Maybe String -- ^ an optional value for "name"
      -> ExceptT ServantError IO HelloMessage

marketing :: ClientInfo -- ^ value for the request body
          -> ExceptT ServantError IO 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 three arguments:

- a `Proxy` to your API,
- a `BaseUrl`, consisting of the protocol, the host, the port and an optional subpath – this basically tells `client` where the service that you want to query is hosted,
- a `Manager`, (from `http-client`) which manages http connections.

```
api :: Proxy API
api = Proxy

{-# NOINLINE __manager #-}
__manager :: Manager
__manager = unsafePerformIO $ newManager defaultManagerSettings

position <|> hello <|> marketing =
  client api (BaseUrl Http "localhost" 8081 "") __manager
```

(Yes, the usage of `unsafePerformIO` is very ugly, we know. Hopefully soon it'll be possible to do without.)

As you can see in the code above, we just “pattern match our way” to these functions. If we try to derive less or more functions than there are endpoints in the API, we obviously get an error. The `BaseUrl` value there is just:

```
-- | URI scheme to use
data Scheme =
  Http -- ^ http://
  | Hhttps -- ^ 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)
  }
```

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

```
queries :: ExceptT ServantError IO (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
  res <- runExceptT queries
  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 {x = 10, y = 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. You now know how to use **servant-client**!

2.4 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 Multiple Paradigms" 2000
  , 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 $-1 \leq x, y \leq 1$. 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
       :<|> serveDirectory "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.

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

```
apiJS :: Text
apiJS = jsForAPI api vanillaJS
```

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

```
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);
}

```

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" apiJS
  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.

2.5 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

```

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")
    ]
    -- mutiple 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.

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`.

```

## Welcome

This is our super webservice's API.

Enjoy!

## 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`

- When a value is provided for 'name'

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

- When 'name' is not specified

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

```

```

    ...

## POST /marketing

#### Request:

- Supported content types are:
    - `application/json`

- Example: `application/json`

    ```javascript
 {"email":"alp@foo.com","interested_in":["haskell","mathematics"],"age":26,"name":"Alp"}
    ```

#### Response:

- Status code 201
- Headers: []

- Supported content types are:
    - `application/json`

- Response body as below.

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

## 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`

- Response body as below.

    ```javascript
 {"x":3,"y":14}
    ```

```

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.

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 :<|> 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.

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