Если вы видите что-то необычное, просто сообщите мне.

Building an API with Servant!

In part 1, we began our series on production Haskell techniques by learning about Persistent. We created a schema that contained a single User type that we could store in a Postgresql database. We examined a couple functions allowing us to make SQL queries about these users.

In this part, we'll see how we can expose this database to the outside world using an API. We'll construct our API using the Servant library. If you've already experienced Servant for yourself, you can move on to part 3, where you'll learn about caching and using Redis from Haskell.

Now, Servant involves some advanced type level constructs, so there's a lot to wrap your head around. There are definitely simpler approaches to HTTP servers than what Servant uses. But I've found that the power Servant gives us is well worth the effort. On the other hand, if you want some simpler approaches, you can take a look at our Production Checklist. It'll give you ideas for some other Web API libraries and a host of other tasks!

As a final note, make sure to look at the Github Repository! For this part, you'll mainly want to look at the Basic Server module.

DEFINING OUR API

The first step in writing an API for our user database is to decide what the different endpoints are. We can decide this independently of what language or library we'll use. For this article, our API will have two different endpoints. The first will be a POST request to /users. This request will contain a "user" definition in its body, and the result will be that we'll create a user in our database. Here's a sample of what this might look like:

```
POST /users
{
  userName : "John Doe",
```

```
  userEmail : "john@doe.com",
  userAge : 29,
  userOccupation: "Teacher"
}
```

It will then return a response containing the database key of the user we created. This will allow any clients to fetch the user again. The second endpoint will use the ID to fetch a user by their database identifier. It will be a GET request to /users/:userid. So for instance, the last request might have returned us something like 16. We could then do the following:

```
GET /users/16
```

And our response would look like the request body from above.

AN API AS A TYPE

So we've got our very simple API. How do we actually define this in Haskell, and more specifically with Servant? Well, Servant does something pretty unique. In Servant we define our API by using a type. Our type will include sub-types for each of the endpoints of our API. We combine the different endpoints by using the (:<|>) operator. I'll sometimes refer to this as "E-plus", for "endpoint-plus". This is a type operator, so remember we'll need the TypeOperators language extension. Here's the blueprint of our API:

```
type UsersAPI =
  fetchEndpoint
  :<|> createEndpoint
```

Now let's define what we mean by fetchEndpoint and createEndpoint. Endpoints combine different combinators that describe different information about the endpoint. We link combinators together with the (:>) operator, which I call "C-plus" (combinator plus). Here's what our final API looks like. We'll go through what each combinator means in the next section:

```
type UsersAPI =
       "users" :> Capture "userid" Int64 :> Get '[JSON] User
  :<|> "users" :> ReqBody '[JSON] User :> Post '[JSON] Int64
```

COMBINATORS

Both of these endpoints have three different combinators. Let's start by examining the fetch endpoint. It starts off with a string combinator. This is a path component, allowing us to specify what url extension the caller should use to hit the endpoint. We can use this combinator multiple times to have a more complicated path for the endpoint. If we instead wanted this endpoint to be at /api/users/:userid then we'd change it to:

```
"api" :> "users" :> Capture "userid" Int64 :> Get '[JSON] User
```

The second combinator (Capture) allows us to get a value out of the URL itself. We give this value a name and then we supply a type parameter. We won't have to do any path parsing or manipulation ourselves. Servant will handle the tricky business of parsing the URL and passing us an Int64. If you want to use your own custom class as a piece of HTTP data, that's not too difficult. You'll just have to write an instance of the FromHttpApiData class. All the basic types like Int64 already have instances.

The final combinator itself contains three important pieces of information for this endpoint. First, it tells us that this is in fact a GET request. Second, it gives us the list of content-types that are allowable in the response. This is a type level list of content formats. Each type in this list must have different classes for serialization and deserialization of our data. We could have used a more complicated list like '[JSON, PlainText, OctetStream]. But for the rest of this article, we'll just use JSON. This means we'll use the ToJSON and FromJSON typeclasses for serialization.

The last piece of this combinator is the type our endpoint returns. So a successful request will give the caller back a response that contains a User in JSON format. Notice this isn't a Maybe User. If the ID is not in our database, we'll return a 401 error to indicate failure, rather than returning Nothing.

Our second endpoint has many similarities. It uses the same string path component. Then its final combinator is the same except that it indicates it is a POST request instead of a GET request. The second combinator then tells us what we can expect the request body to look like. In this case, the request body should contain a JSON representation of a User. It also requires a list of acceptable content types, and then the type we want, like the Get and Post combinators.

That completes the "definition" of our API. We'll need to add ToJSON and FromJSON instances of our User type in order for this to function. You can take a look at those on Github, and check out this article for more details on creating those instances!

WRITING HANDLERS

Now that we've defined the type of our API, we need to write handler functions for each endpoint. This is where Servant's awesomeness kicks in. We can map each endpoint up to a function that has a particular type based on the combinators in the endpoint. So, first let's remember our endpoint for fetching a user:

```
"users" :> Capture "userid" Int64 :> Get '[JSON] User
```

The string path component doesn't add any arguments to our function. The Capture component will result in a parameter of type Int64 that we'll need in our function. Then the return type of our function should be User. This almost completely defines the type signature of our handler. We'll note though that it needs to be in the Handler monad. So here's what it'll look like:

```
fetchUsersHandler :: Int64 -> Handler User
...
```

We can also look at the type for our create endpoint:

```
"users" :> ReqBody '[JSON] User :> Post '[JSON] Int64
```

The parameter for a ReqBody parameter is just the type argument. So it will resolve this endpoint into the handler monad like so:

```
createUserHandler :: User -> Handler Int64
...
```

Now, we'll need to be able to access our Postgres database through both of these handlers. So they'll each get an extra parameter referring to the connection string (recall the PGInfo type alias). We'll pass that from our code so that by the time Servant is resolving the types, the parameter is accounted for:

```
fetchUsersHandler :: PGInfo -> Int64 -> Handler User
createUserHandler :: PGInfo -> User -> Handler Int64
```

THE HANDLER MONAD

Before we go any further, we should discuss the Handler monad. This is a wrapper around the monad ExceptT ServantErr IO. In other words, each of these requests might fail. To make it fail, we can throw errors of type ServantErr. Then of course we can also call IO functions, because these are network operations.

Before we implement these functions, let's first write a couple simple helpers. These will use the runAction function from the last part to run database actions:

```
fetchUserPG :: PGInfo -> Int64 -> IO (Maybe User)
fetchUserPG connString uid = runAction connString (get (toSqlKey uid))

createUserPG :: PGInfo -> User -> IO Int64
createUserPG connString user = fromSqlKey <$> runAction connString (insert user)
```

For completeness (and use later in testing), we'll also add a simple delete function. We need the signature on the where clause for type inference:

```
deleteUserPG :: ConnectionString -> Int64 -> IO ()
deleteUserPG connString uid = runAction connString (delete userKey)
  where
    userKey :: Key User
    userKey = toSqlKey uid
```

Now we'll call these two functions from our Servant handlers. This will completely cover the case of the create endpoint. But we'll need a little bit more logic for the fetch endpoint. Since our functions are in the IO monad, we have to lift them up to Handler.

```
fetchUsersHandler :: ConnectionString -> Int64 -> Handler User
fetchUserHandler connString uid = do
  maybeUser <- liftIO $ fetchUserPG connString uid
  ...
```

```
createUserHandler :: ConnectionString -> User -> Handler Int64
createuserHandler connString user = liftIO $ createUserPG connString user
```

To complete our fetch handler, we need to account for a non-existent user. Instead of making the type of the whole endpoint a Maybe, we'll throw a ServantErr in this case. We can use one of the built-in Servant error functions, which correspond to normal error codes. Then we can update the body. In this case, we'll throw a 401 error. Here's how we do that:

```
fetchUsersHandler :: ConnectionString -> Int64 -> Handler User
fetchUserHandler connString uid = do
  maybeUser <- lift $ fetchUserPG connString uid
  case maybeUser of
    Just user -> return user
    Nothing -> Handler $ (throwE $ err401 { errBody = "Could not find user with ID: " ++ (show
uid)})

createUserHandler :: ConnectionString -> User -> Handler Int64
createuserHandler connString user = lift $ createUserPG connString user
```

And that's it! We're done with our handler functions!

COMBINING IT ALL INTO A SERVER

Our next step is to create an object of type Server over our API. This is actually remarkably simple. When we defined the original type, we combined the endpoints with the (:<|>) operator. To make our Server, we do the same thing but with the handler functions:

```
usersServer :: ConnectionString -> Server UsersAPI
usersServer pgInfo =
  (fetchUsersHandler pgInfo) :<|>
  (createUserHandler pgInfo)
```

And Servant does all the work of ensuring that the type of each endpoint matches up with the type of the handler! Suppose we changed the type of our fetchUsersHandler so that it took a Key User instead of an Int64. We'd get a compile error:

```
fetchUsersHandler :: ConnectionString -> Key User -> Handler User
...

-- Compile Error!
• Couldn't match type ‘Key User' with ‘Int64'
      Expected type: Server UsersAPI
        Actual type: (Key User -> Handler User)
                     :<|> (User -> Handler Int64)
```

There's now a mismatch between our API definition and our handler definition. So Servant knows to throw an error! The one issue is that the error messages can be rather difficult to interpret sometimes. This is especially the case when your API becomes very large! The "Actual type" section of the above error will become massive! So always be careful when changing your endpoints! Frequent compilation is your friend!

BUILDING THE APPLICATION

The final piece of the puzzle is to actually build an Application object out of our server. The first step of this process is to create a Proxy for our API. Remember that our API is a type, and not a term. But a Proxy allows us to represent this type at the term level. The concept is a little complicated, but the code is not!

```
import Data.Proxy

...

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

Now we can make our runnable Application like so (assuming we have a Postgres connection):

```
serve usersAPI (usersServer connString)
```

We'll run this server from port 8000 by using the run function, again from Network.Wai. (See Github for a full list of imports). We'll fetch our connection string, and then we're good to go!

```
runServer :: IO ()
runServer = run 8000 (serve usersAPI (usersServer localConnString))
```

CONCLUSION

The Servant library offers some truly awesome possibilities. We're able to define a web API at the type level. We can then define handler functions using the parameters the endpoints expect. Servant handles all the work of marshalling back and forth between the HTTP request and the native Haskell types. It also ensures a match between the endpoints and the handler function types!

Now you're ready for part 3 of our Real World Haskell series! You'll learn how we can modify our API to be faster by employing a Redis cache!

This part of the series gave a brief overview on Servant. But if you want a more in-depth introduction, you should check out my talk from Bayhac from April 2017! That talk was more exhaustive about the different combinators you can use in your APIs. It also showed authentication techniques, client functions and documentation. You can also check out the slides and code for that presentation!

On the other hand, Servant is also quite involved. If you want some ideas for a simpler solution, check out our Production Checklist! It'll give a couple other suggestions for Web API libraries and so much more!

Revision #1
Created 2022-03-11 06:16:16 UTC by gasick
Updated 2022-03-11 17:11:17 UTC by gasick