

# GraphQL Haskell Tutorial

## Contents

Getting started . . . . .	1
First example . . . . .	2
Monadic actions . . . . .	2
Errors . . . . .	3
Combining resolvers . . . . .	3
Further examples . . . . .	4

## Getting started

Welcome to graphql-haskell!

We have written a small tutorial to help you (and ourselves) understand the graphql package.

Since this file is a literate haskell file, we start by importing some dependencies.

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE LambdaCase #-}
module Main where

import Prelude hiding (empty, putStrLn)
import Data.GraphQL
import Data.GraphQL.Schema
import qualified Data.GraphQL.Schema as Schema

import Control.Applicative
import Data.Text hiding (empty)
import Data.Aeson
import Data.ByteString.Lazy.Char8 (putStrLn)

import Data.Time

import Debug.Trace
```

## First example

Now, as our first example, we are going to look at the example from graphql.js.

First we build a GraphQL schema.

```
schema1 :: Alternative f => Schema f
schema1 = Schema [hello]
```

```
hello :: Alternative f => Resolver f
hello = Schema.scalar "hello" ("it's me" :: Text)
```

This defines a simple schema with one type and one field, that resolves to a fixed value.

Next we define our query.

```
query1 :: Text
query1 = "{ hello }"
```

To run the query, we call the `graphql` with the schema and the query.

```
main1 :: IO ()
main1 = putStrLn =<< encode <$> graphql schema1 query1
```

This runs the query by fetching the one field defined, returning

```
{"data" : {"hello":"it's me"}}
```

## Monadic actions

For this example, we're going to be using `time`.

```
schema2 :: Schema IO
schema2 = Schema [time]
```

```
time :: Resolver IO
time = Schema.scalarA "time" $ \case
  [] -> do t <- getCurrentTime
         return $ show t
  _   -> empty
```

This defines a simple schema with one type and one field, which resolves to the current time.

Next we define our query.

```
query2 :: Text
query2 = "{ time }"
```

```
main2 :: IO ()
main2 = putStrLn =<< encode <$> graphql schema2 query2
```

This runs the query, returning the current time

```
{"data": {"time": "2016-03-08 23:28:14.546899 UTC"}}
```

## Errors

Errors are handled according to the spec, with fields that cause errors being resolved to null, and an error being added to the error list.

An example of this is the following query:

```
queryShouldFail :: Text
queryShouldFail = "{ boyhowdy }"
```

Since there is no `boyhowdy` field in our schema, it will not resolve, and the query will fail, as we can see in the following example.

```
mainShouldFail :: IO ()
mainShouldFail = do
  r <- graphql schema1 query1
  putStrLn $ encode r
  putStrLn "This will fail"
  r <- graphql schema1 queryShouldFail
  putStrLn $ encode r
```

This outputs:

```
{"data": {"hello": "it's me"}}
This will fail
{"data": {"boyhowdy": null}, "errors": [{"message": "the field boyhowdy did not resolve."}]}
```

## Combining resolvers

Now that we have two resolvers, we can define a schema which uses them both.

```
schema3 :: Schema IO
schema3 = Schema [hello, time]
```

```
query3 :: Text
query3 = "query timeAndHello { time hello }"
```

```
main3 :: IO ()
main3 = putStrLn =<< encode <$> graphql schema3 query3
```

This queries for both time and hello, returning

```
{ "data": {"hello": "it's me", "time": "2016-03-08 23:29:11.62108 UTC"}}
```

Notice that we can name our queries, as we did with `timeAndHello`. Since we have only been using single queries, we can use the shorthand `{ time hello}`, as we have been doing in the previous examples.

In GraphQL there can only be one operation per query.

## Further examples

More examples on queries and a more complex schema can be found in the test directory, in the `Test.StarWars` module. This includes a more complex schema, and more complex queries.