# GraphQL API tutorial Documentation

***Release 0.1***

**teh, jml**

December 19, 2016

Contents

Contents:

CHAPTER 1

# Defining GraphQL type APIs

First some imports:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE OverloadedStrings #-}
module Introduction where

import Protolude

import GraphQL.TypedSchema (Object, Field, Argument, (:>))
import GraphQL.TypedApi (Handler, (:<>)(..))
```

The core idea for this library is that we define a composite type that specifies the whole API, and then implement a matching handler.

The main GraphQL entities we care about are Objects and Fields. Each Field can have arguments.

```
type HelloWorld = Object "HelloWorld" '[]
  '[ Argument "greeting" Text :> Field "me" Text
   ]
```

The example above is equivalent to the following GraphQL type:

```
type HelloWorld {
  me(greeting: String!): String!
}
```

And if we had a code to handle that type (more later) we could query it like this:

```
{ me(greeting: "hello") }
```

## 1.1 The handler

We defined a corresponding handler via the `Handler m a` which takes the monad to run in (`IO` in this case) and the actual API definition (`HelloWorld`).

```
handler :: Handler IO HelloWorld
handler = pure $ (\greeting -> pure (greeting <> " to me")) :<> ()
```

The implementation looks slightly weird, but it's weird for good reasons. In order:

- The first `pure` allows us to run actions in the base monad (`IO` here) before returning anything. This is useful to allocate a resource like a database connection.

- The `pure` in the function call allows us to **avoid running actions** when the field hasn't been requested: Each handler is a separate monadic action so we only perform the side effects for fields present in the query.
- Finally, we have to terminate each handler with `:<> ()`. This is an implementation artifact which we'd prefer to avoid but can not at the moment.

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search