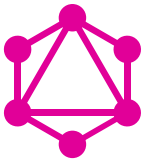


GraphQL and Perl 6

Curt Tilmes

Curt.Tilmes@nasa.gov

Philadelphia Perl Mongers

2017-01-09

Introduction

- GraphQL invented by Facebook to improve mobile app performance
- Now widely used within Facebook, and growing in popularity
- Open Source reference implementation and specification at graphql.org
- Implementations: Javascript, Ruby, PHP, Python, Java, C/C++, Go, Scala, .NET, Elixir, Haskell, SQL, Lua, Elm, Clojure, Swift, OCaml
- Many are migrating from RESTful APIs to GraphQL.
- Often multiple REST queries can be merged into a single GraphQL query with a single round trip client to server.
- Check out https://developer.github.com/early-access/graphql/

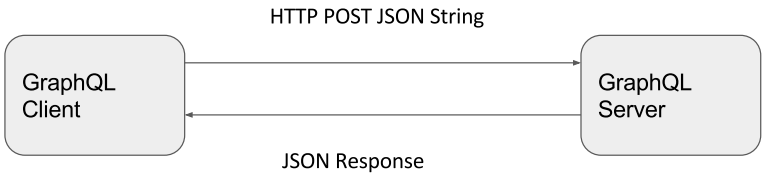
Hello World

```
use GraphQL;

class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);

say $schema.execute('{ hello }').to-json;
```

Hello World

```
use GraphQL;

class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);

say $schema.execute('{ hello }').to-json;
```

```
{
  "data": {
    "hello": "Hello World"
  }
}
```


HTTP POST JSON String
GraphQL
Client
GraphQL
Server
JSON Response

Hello World GraphQL Server

```
use GraphQL;
use GraphQL::Server;

class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);

GraphQL-Server($schema);
```

- Wraps `$schema.execute()` in a simple Bailador server

Hello World GraphQL Server

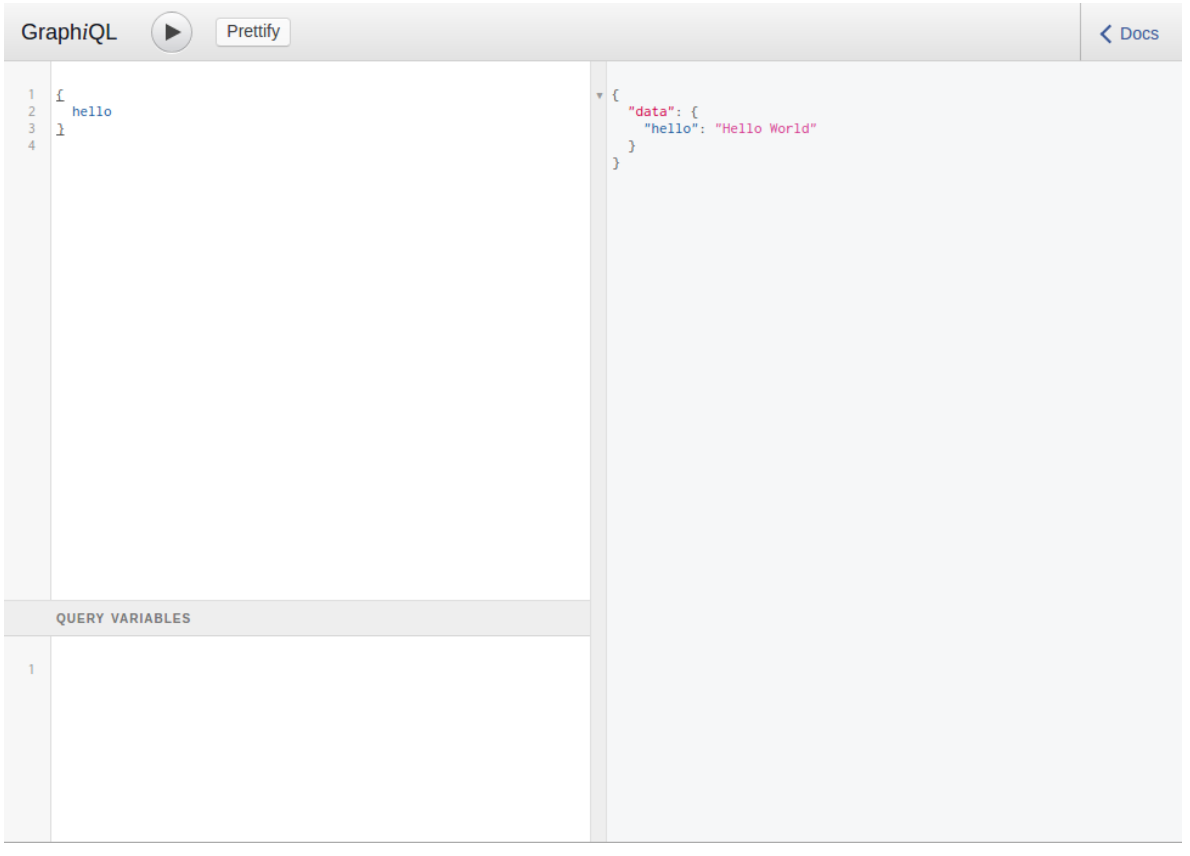
```
use GraphQL;
use GraphQL::Server;

class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);

GraphQL-Server($schema);
```

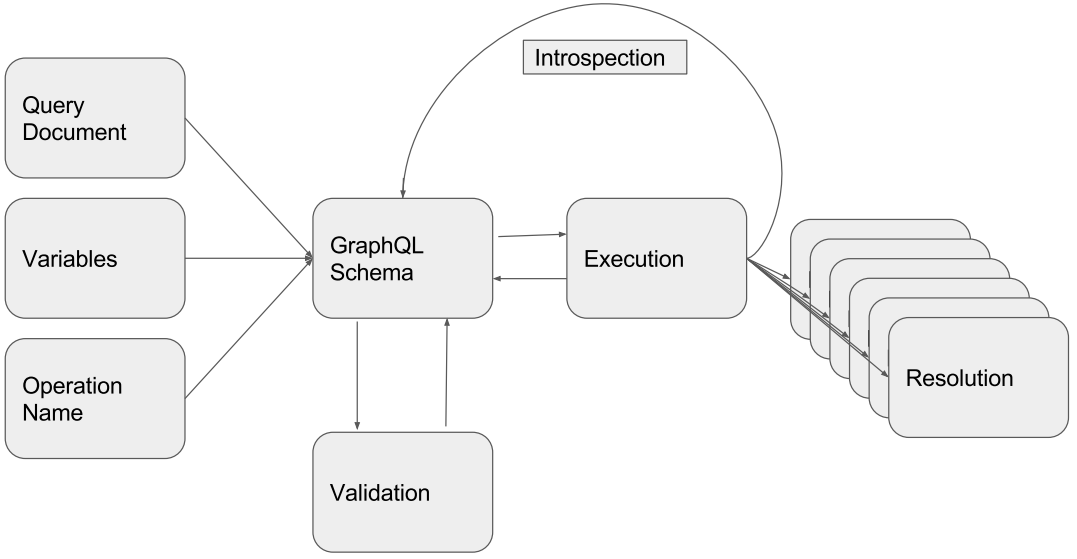
- Wraps `$schema.execute()` in a simple Bailador server
- HTTP POST GraphQL Query to `/graphql`
    - JSON response comes back

Hello World GraphQL Server

```
use GraphQL;
use GraphQL::Server;

class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);

GraphQL-Server($schema);
```

- Wraps `$schema.execute()` in a simple Bailador server
- HTTP POST GraphQL Query to `/graphql`
  - JSON response comes back
- HTTP GET `/graphql` with no parameters
  - returns Facebook Graph*i*QL IDE


GraphiQL
Prettify
Docs
{
hello
}
{
"data": {
"hello": "Hello World"
}
}
QUERY VARIABLES


Introspection
Query Document
Variables
Operation Name
GraphQL Schema
Execution
Resolution
Validation

Perl Schema definition

- Three styles:
    1. Manual
    2. GraphQL Schema Language (GSL)
    3. Perl 6 Class introspection

Perl Schema definition

- Three styles:
    1. Manual
    2. GraphQL Schema Language (GSL)
    3. Perl 6 Class introspection
- All result in the same schema
    - GSL is just parsed to produce the manual objects
    - Perl objects are introspected with Perl 6's Metamodel

Perl Schema definition

- Three styles:
    1. Manual
    2. GraphQL Schema Language (GSL)
    3. Perl 6 Class introspection
- All result in the same schema
    - GSL is just parsed to produce the manual objects
    - Perl objects are introspected with Perl 6's Metamodel
- Can be inter-mixed as desired

Manual Schema Creation

```
my $schema = GraphQL::Schema.new(
    GraphQL::Object.new(
        name => 'Query',
        fieldlist => GraphQL::Field.new(
            name => 'hello',
            type => $GraphQLString,
            resolver => sub { 'Hello World' }
        )
    )
);
```

- GraphQL is strongly, staticly typed
- GraphQL Types can have a 'name' and 'description'
- GraphQL Fields can have a resolver

Some of the GraphQL classes

GraphQL Type	Perl Class
String	GraphQL::String
Int	GraphQL::Int
Float	GraphQL::Float
Boolean	GraphQL::Boolean
ID	GraphQL::ID
Scalar	GraphQL::Scalar
List	GraphQL::List
Non-Null	GraphQL::Non-Null
Object/Type	GraphQL::Object
Field	GraphQL::Field
Interface	GraphQL::Interface
Enum	GraphQL::Enum
Union	GraphQL::Union
Input	GraphQL::Input

GraphQL Schema Language (GSL)

```
my $schema = GraphQL::Schema.new('type Query { hello: String }',
    resolvers =>
    {
        Query =>
        {
            hello => sub { 'Hello World' }
        }
    }
);
```

- Compatible with many other language's schema definitions
- Pass a two level hash with resolving functions. Object/Field.

GraphQL Schema Language Cheat Sheet

The definitive guide to express your GraphQL schema succinctly

Last updated: 10 June 2016 *Prepared by: Hafiz Ismail / @sogko*

What is GraphQL Schema Language?

It is a shorthand notation to succinctly express the basic shape of your GraphQL schema and its type system.

What does it look like?

Would you believe me if I say it is the most beautiful thing you've ever laid your eyes upon?

Below is an example of a typical GraphQL schema expressed in shorthand.

```
interface Entity {
  id: ID!
  name: String
}

scalar Url

type User implements Entity {
  id: ID!
  name: String
  age: Int
  balance: Float
  is_active: Boolean
  friends: [User]!
  website: Url
}

type Root {
   me: User
   friends(limit: Int = 10): [User]!
}

schema {
  query: Root
  mutation: ...
  subscription: ...
}
```

Schema

`schema`	GraphQL schema definition
`query`	A read-only fetch operation
`mutation`	A write followed by fetch operation
`subscription`	A subscription operation (experimental)

Built-in Scalar Types

`Int`	Int
`Float`	Float
`String`	String
`Boolean`	Boolean
`ID`	ID

Type Definitions

`scalar`	Scalar Type
`type`	Object Type
`interface`	Interface Type
`union`	Union Type
`enum`	Enum Type
`input`	Input Object Type

Type Markers

`String`	Nullable String type
`String!`	Non-null String type
`[String]`	List of nullable Strings type
`[String]!`	Non-null list of nullable Strings type
`[String!]!`	Non-null list of non-null Strings type

Input Arguments

Basic Input

```
type Root {
   users(limit: Int): [User]
}
```

Input with default value

```
type Root {
   users(limit: Int = 10): [User]
}
```

Input with multiple arguments

```
type Root {
   users(limit: Int, sort: String): [User]
}
```

Input with multiple arguments and default values

```
type Root {
  users(limit: Int = 10, sort: String): [User]
}

type Root {
  users(limit: Int, sort: String = "asc"): [User]
}

type Root {
  users(limit: Int = 10, sort: String = "asc"): [User]
}
```

Input Object Types

```
input ListUsersInput {
  limit: Int
  since_id: ID
}
type Root {
  users(params: ListUsersInput): [User]!
}
```

Custom Scalars

```
scalar Url
type User {
  name: String
  homepage: Url
}
```

Interfaces

Object implementing one or more Interfaces

```
interface Foo {
  is_foo: Boolean
}
interface Goo {
  is_goo: Boolean
}
type Bar implements Foo {
  is_foo: Boolean
  is_bar: Boolean
}
type Baz implements Foo, Goo {
  is_foo: Boolean
  is_goo: Boolean
  is_baz: Boolean
}
```

Unions

Union of one or more Objects

```
type Foo {
  name: String
}
type Bar {
  is_bar: String
}
union SingleUnion = Foo
union MultipleUnion = Foo | Bar
type Root {
  single: SingleUnion
  multiple: MultipleUnion
}
```

Enums

```
enum USER_STATE {
  NOT_FOUND
  ACTIVE
  INACTIVE
  SUSPENDED
}
type Root {
  stateForUser(userID: ID!): STATE!
  users(state: STATE, limit: Int = 10): [User]
}
```

Perl

```
class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);
```

Perl

```
class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);
```

- Some restrictions on how you construct your objects.
- All named/typed arguments, including typed return.

Perl

```
class Query
{
    method hello(--> Str) { 'Hello World' }
}

my $schema = GraphQL::Schema.new(Query);
```

- Some restrictions on how you construct your objects.
- All named/typed arguments, including typed return.

GraphQL Type	Perl Type
String	Str
Int	Int
Float	Num
Boolean	Bool
ID	ID (subset of Cool)

Example Database

```
class User
{
    has Int $.id;
    has Str $.name;
    has Str $.birthday;
    has Bool $.status;
}

my User @users =
    User.new(id => "0",
             name => 'Gilligan',
             birthday => 'Friday',
             status => True),
    User.new(id => "1",
             name => 'Skipper',
             birthday => 'Monday',
             status => False),
    User.new(id => "2",
             name => 'Professor',
             birthday => 'Tuesday',
             status => True);
```

Example Schema (GSL):

```
type User {
  id: ID!
  name: String!
  birthday: String
  status: Boolean
}

type Query {
  user(id: ID!): User
  listusers(start: ID = "0", count: Int = 3): [User]
}
```

Example Schema (GSL):

```
type User {
  id: ID!
  name: String!
  birthday: String
  status: Boolean
}

type Query {
  user(id: ID!): User
  listusers(start: ID = "0", count: Int = 3): [User]
}
```

```
my $resolvers = {
    Query => {
        user => sub (:$id) {
            @users[$id]
        },
        listusers => sub (:$start, :$count) {
            @users[$start ..^ $start+$count]
        }
    }
};
```

Example Schema (Perl)

```
class User
{
    has ID:D $.id is required;
    has Str:D $.name is required;
    has Str $.birthday;
    has Bool $.status;
}
```

Example Schema (Perl)

```
class User
{
    has ID:D $.id is required;
    has Str:D $.name is required;
    has Str $.birthday;
    has Bool $.status;
}
```

```
class Query
{
    method user(ID :$id! --> User)
    {
        @users[$id]
    }

    method listusers(ID :$start = "0", Int :$count = 3 --> Array[User])
    {
        Array[User].new(@users[$start ..^ $start+$count])
    }
}
```

Example Schema (Perl)

```
class User
{
    has ID:D $.id is required;
    has Str:D $.name is required;
    has Str $.birthday;
    has Bool $.status;
}
```

```
class Query
{
    method user(ID :$id! --> User)
    {
        @users[$id]
    }

    method listusers(ID :$start = "0", Int :$count = 3 --> Array[User])
    {
        Array[User].new(@users[$start ..^ $start+$count])
    }
}
```

```
my $schema = GraphQL::Schema.new(User, Query);
```

Simple Query

```
{
  user(id: "0") {
    name
    birthday
    status
  }
}
```

Simple Query

```
{
  user(id: "0") {
    name
    birthday
    status
  }
}
```

```
{
  "data": {
    "user": {
      "name": "Gilligan",
      "birthday": "Friday",
      "status": "true"
    }
  }
}
```

Multiple queries, distinguish with aliases

```
{
  a: user(id: "0") {
    name
    birthday
    status
  }
  b: user(id: "1") {
    name
    birthday
    status
  }
}
```

Multiple queries, distinguish with aliases

```
{
  a: user(id: "0") {
    name
    birthday
    status
  }
  b: user(id: "1") {
    name
    birthday
    status
  }
}
```

```
{
  "data": {
    "a": {
      "name": "Gilligan",
      "birthday": "Friday",
      "status": "true",
    },
    "b": {
      "name": "Skipper",
      "birthday": "Monday",
      "status": "false",
    }
  }
}
```

Reuse fieldlists with fragments

```
{
  a: user(id: "0") {
    ...somefields
  }
  b: user(id: "1"){
    ...somefields
  }
}

fragment somefields on User {
  name
  birthday
  status
}
```

Reuse fieldlists with fragments

```
{
  a: user(id: "0") {
    ...somefields
  }
  b: user(id: "1"){
    ...somefields
  }
}

fragment somefields on User {
  name
  birthday
  status
}
```

```
{
  "data": {
    "a": {
      "name": "Gilligan",
      "birthday": "Friday",
      "status": "true",
    },
    "b": {
      "name": "Skipper",
      "birthday": "Monday",
      "status": "false",
    }
  }
}
```

```
{
  listusers(start: "1",
            count: 3) {
    ...somefields
  }
}
```

```
{
  listusers(start: "1",
            count: 3) {
    ...somefields
  }
}
```

```
{
  "data": {
    "listusers": [
      {
        "name": "Skipper",
        "birthday": "Monday",
        "status": "false"
      },
      {
        "name": "Professor",
        "birthday": "Tuesday",
        "status": "true"
      },
      {
        "name": "Ginger",
        "birthday": "Wednesday",
        "status": "true"
      }
    ]
  }
}
```

Example - Enum

```
enum State <NOT_FOUND ACTIVE INACTIVE SUSPENDED>;

class User
{
    has ID:D $.id is required;
    has Str:D $.name is required;
    has Str $.birthday;
    has Bool $.status;
    has State $.state;
}


my $schema = GraphQL::Schema.new(State, User, Query);
```

```
{
  listusers(start: "1",
            count: 3) {
    name
    birthday
    status
    state
  }
}
```

```
{
  "data": {
    "listusers": [
      {
        "name": "Skipper",
        "birthday": "Monday",
        "status": "false",
        "state": "ACTIVE"
      },
      {
        "name": "Professor",
        "birthday": "Tuesday",
        "status": "true",
        "state": "INACTIVE"
      },
      {
        "name": "Ginger",
        "birthday": "Wednesday",
        "status": "true",
        "state": "SUSPENDED"
      }
    ]
  }
}
```

Input Object

```
class UserInput is GraphQL::InputObject
{
    has Str $.name;
    has Str $.birthday;
    has Bool $.status;
    has State $.state;
}
```

Mutation

```
class Mutation
{
    method adduser(UserInput :$newuser! --> ID) {
        push @users, User.new(id => @users.elems,
                              name => $newuser.name,
                              birthday => $newuser.birthday,
                              status => $newuser.status,
                              state => $newuser.state);
        return @users.elems - 1;
    }

    method updateuser(ID :$id!, UserInput :$userinput! --> User) {
        for <name birthday status state> -> $field {
            if $userinput."$field"().defined {
                @users[$id]."$field"() = $userinput."$field"();
            }
        }
        return Query.user(:$id);
    }
}

my $schema = GraphQL::Schema.new(State, User, Query, UserInput, Mutation);
```

```
mutation {
  adduser(newuser: { name: "Thurston" })
}
```

```
mutation {
  adduser(newuser: { name: "Thurston" })
}
```

```
{
  "data": {
    "adduser": "5"
  }
}
```

```
mutation {
  adduser(newuser: { name: "Thurston" })
}
```

```
{
  "data": {
    "adduser": "5"
  }
}
```

```
{
  user(id: "5") {
    name
    birthday
    status
    state
  }
}
```

```
mutation {
  adduser(newuser: { name: "Thurston" })
}
```

```
{
  "data": {
    "adduser": "5"
  }
}
```

```
{
  user(id: "5") {
    name
    birthday
    status
    state
  }
}
```

```
{
  "data": {
    "user": {
      "name": "Thurston",
      "birthday": null,
      "status": "false",
      "state": null
    }
  }
}
```

Include variables

```
mutation ($bday: String) {
  updateuser(id: "5",
             userinput: {
               birthday: $bday,
               state: ACTIVE
             })
  {
    name
    birthday
    status
    state
  }
}
```

```
{
  "bday":"May 1"
}
```

- Variables in JSON

Include variables

```
mutation ($bday: String) {
  updateuser(id: "5",
             userinput: {
               birthday: $bday,
               state: ACTIVE
             })
  {
    name
    birthday
    status
    state
  }
}
```

```
{
  "bday":"May 1"
}
```

```
{
  "data": {
    "updateuser": {
      "name": "Thurston",
      "birthday": "May 1",
      "status": "false",
      "state": "ACTIVE"
    }
  }
}
```

- Variables in JSON

Variable data structure

```
mutation ($newuser: UserInput) {
  adduser(newuser: $newuser)
}
```

Variable data structure

```
mutation ($newuser: UserInput) {
  adduser(newuser: $newuser)
}
```

```
{
  "newuser": {
    "name": "Lovey",
    "birthday": "Oct 17",
    "status": true,
    "state": "INACTIVE"
  }
}
```

Variable data structure

```
mutation ($newuser: UserInput) {
  adduser(newuser: $newuser)
}
```

```
{
  "newuser": {
    "name": "Lovey",
    "birthday": "Oct 17",
    "status": true,
    "state": "INACTIVE"
  }
}
```

```
{
  "data": {
    "adduser": "6"
  }
}
```

Control fields with directives (false)

```
query ($extrafields: Boolean) {
  listusers(count: 2) {
    id @skip(if: $extrafields)
    name
    ...extrafields
      @include(if: $extrafields)
  }
}

fragment extrafields on User {
  birthday
  status
  state
}
```

```
{
  "extrafields": false
}
```

```
{
  "data": {
    "listusers": [
      {
        "id": "0",
        "name": "Gilligan"
      },
      {
        "id": "1",
        "name": "Skipper"
      }
    ]
  }
}
```

Control fields with directives (true)

```
query ($extrafields: Boolean) {
  listusers(count: 2) {
    id @skip(if: $extrafields)
    name
    ...extrafields
      @include(if: $extrafields)
  }
}

fragment extrafields on User {
  birthday
  status
  state
}
```

```
{
  "extrafields": true
}
```


```
{
  "data": {
    "listusers": [
      {
        "name": "Gilligan",
        "birthday": "Friday",
        "status": "true",
        "state": "NOT_FOUND"
      },
      {
        "name": "Skipper",
        "birthday": "Monday",
        "status": "false",
        "state": "ACTIVE"
      }
    ]
  }
}
```

Use promises to do slow queries in parallel

- Resolver can return promise:

```
user => sub (:$id)
{
    start {
        sleep 2;
        @users[$id];
    }
}
```

Use promises to do slow queries in parallel

- Resolver can return promise:

```
user => sub (:$id)
{
    start {
        sleep 2;
        @users[$id];
    }
}
```

- Perl method is typed, it can't return a promise, use trait

```
method user(ID :$id! --> User) is graphql-background
{
    sleep 2;
    @users[$id];
}
```

GraphQL

- Individual fields and enum values can be marked deprecated, but still work

GraphQL

- Individual fields and enum values can be marked deprecated, but still work
- A query can be validated against a schema prior to execution.

GraphQL

- Individual fields and enum values can be marked deprecated, but still work
- A query can be validated against a schema prior to execution.
- A parsed, validated query could be cached. Some applications just identify pre-validated queries by identifier rather than allowing arbitrary queries.

GraphQL

- Individual fields and enum values can be marked deprecated, but still work
- A query can be validated against a schema prior to execution.
- A parsed, validated query could be cached. Some applications just identify pre-validated queries by identifier rather than allowing arbitrary queries.
- Reuse queries with variables for things that change rather than making a new query.

GraphQL

- Individual fields and enum values can be marked deprecated, but still work
- A query can be validated against a schema prior to execution.
- A parsed, validated query could be cached. Some applications just identify pre-validated queries by identifier rather than allowing arbitrary queries.
- Reuse queries with variables for things that change rather than making a new query.
- Publish/Subscribe coming soon for spec, not yet in the Perl version

GraphQL

- Individual fields and enum values can be marked deprecated, but still work
- A query can be validated against a schema prior to execution.
- A parsed, validated query could be cached. Some applications just identify pre-validated queries by identifier rather than allowing arbitrary queries.
- Reuse queries with variables for things that change rather than making a new query.
- Publish/Subscribe coming soon for spec, not yet in the Perl version
- Client libraries and tools are rapidly developing, including client side validation and caching.

Conclusion

- Perl 6 implementation still restricted access, but contact me if you are interested in it.

Conclusion

- Perl 6 implementation still restricted access, but contact me if you are interested in it.
- Loads more information available at graphql.org, including many youtube videos of talks.
- Almost all public information about GraphQL fully applies to the Perl 6 version.

Conclusion

- Perl 6 implementation still restricted access, but contact me if you are interested in it.
- Loads more information available at graphql.org, including many youtube videos of talks.
- Almost all public information about GraphQL fully applies to the Perl 6 version.

Thank You!