# GraphQL Erlang

Jesper Louis Andersen
jesper.louis.andersen@gmail.com
ShopGun

June 8, 2017

Overview:

- Background
- GraphQL Itself
- The implementation

# Not covered

I can't cover everything. A list of things which has a story in GraphQL, but I skip:

- Subscriptions
- Abstract types: Interfaces, Unions
- Authentication/Authorization
- Error handling
- Schema Loading and Validation
- Directives
- Aliasing of field names
- …

# Once upon a time...

- ShopGun's mission: index the worlds shopping data
- Shopping data: Semi-structured data set
- Think Time Zones and Calendars
- Densely populated dataset, many links

How do we create an API for such a data model?

# State

We started with some analysis:

- Have existing HTTP/1.1 API
- HTTP/1.1 or HTTP/2 ng?
- Falcor?
- GraphQL?

Ended with GraphQL: heaviest solution but also solves our problems.

What are our major problems in the current API?

- Multiple clients: Each client needs different data
- Some clients use typed languages, some use untyped languages
- Many obvious type errors occur and slows development
- The data evolves over time, and requires lots of server-side tuning
- Documentation is added ad-hoc to the API
- Request/Response structure is unclear and client developers spend time adapting

# GraphQL: Initial Commit

- Created by Facebook in 2012, public (draft) spec in 2015
- Used on Android (Java), iOS (Obj-C), Web (Javascript)
- Can be used to replace (RESTful) web services
- Client/Server Query Language
- Some ideas from Armstrong's UBF are in there
- Often JSON output, but isn't bound to JSON

# GraphQL Major features we like

- There is a schema-definition of data (contract)
- The schema is checked for internal consistency (contract checking)
- Client declares what it wants through query (declarative)
- Client declarations must explicitly mention the data wanted in the request
- The server handles and processes the queries (query execution)

- The schema is fully typed
- An request with a (non-coercible) type error is rejected
- A response with a type error is coerced into a valid response
- The server allows introspection queries on the meta-structure of the contract (automatic discovery)

---

Note: These things solves our current major problems.

# Example

Input (GraphQL):

```
query Planet {
  node(id: "UGxhbmV0OjE=") {
    ... on Planet {
      id
      name
      orbitalPeriod
    }
  }
}
```

Output (JSON):

```
{
  "data": {
    "node": {
      "id": "UGxhbmV0OjE=",
      "name": "Tatooine",
      "orbitalPeriod": 304
    }
  }
}
```

- Only requested fields are returned
- Must request all fields
- Output structure reflects input structure

# Example

```
query Q {
  room(id: "cm9vbToz") {
    description
    exits {
      direction
      room {
        id
        description
      }
    }
  }
}
```


```
"room": {
  "description": "Dungeon Entrance",
  "exits": [
   {
    "direction": "north",
    "room": {
     "description": "A dark tunnel",
     "id": "cm9vbTox" } },
   {
    "direction": "secret_passage",
    "room": {
     "description":
       "In a secret passage",
     "id": "cm9vbToy" } }]
}
```


- Schema defines if a field is a scalar or object
- Schema defines if a field is composite: (array, non-null)

Our current API responds slowly at times, due to the round–trip time between the client and the server. Especially on mobile phones with bad connectivity.

- How do we solve this?

- One query, all operations happen on the server side
- Round trip time is between servers, often a few milliseconds at most
- Lower latency achieved as a result
- Can avoid lots of “boiler plate” endpoints
- Move most “looping” in RESTful services to the GraphQL execution engine

# Fragments

```
query Q {
  monster(id:"...") {
    ...MonsterFrag
  }
  room(id:"...") {
    contents {
      ...MonsterFrag
    }
  }
}

fragment MonsterFrag on Monster {
  id
  name
  hitpoints
}
```

- Fragments allow concise reference to fields
- Fragments also provide “downcasting” (contents “can” be a monster)

# Fragments (2)

- Clients build a fragment for each of their UI elements
- Throws every fragment they got at the server
- Server performs “field collection” to merge the fragments into one query
- Clients are free to use these features or not
- Clients can evolve at different pace

# Parameterized queries

```
query Q($monsterId: Id!) {
  monster(id: $monsterId) {
    ...MonsterFrag
  }
}
```

- Parameterize Q so it can be reused again and again
- Query document contains 50–60 queries. You select one query by its name and provide its parameters
- Arguably safer once you lock down the query document in production
- Maximally flexible in development, execution of "stored procedures" in production

# Mutations

```
mutation NewMonster {
  introduceMonster(input:
    {clientMutationId: "123",
     name: "Succubus",
     hitpoints: 24,
     color: "#bbbb00"}) {
    clientMutationId
    monster {
      id
      name
    }
  }
}
```

```
"introduceMonster": {
  "clientMutationId": "123",
  "monster": {
    "id": "bW9uc3Rlcjoz",
    "name": "Succubus"
  }
}
```

- Changes are through mutations
- A mutation is like a query (but the server handles it differently)
- Note input objects!

# What you have seen until now

- The queries are from GraphQL test cases
- There is a GraphQL server written in Erlang
- There is a complete tutorial implementing a database for Star Wars™ entities.
- The tutorial is backed by an in-memory, disk-backed persistent mnesia instance

DEMO(!!)

# Server-side

- We have a parser for typical GraphQL specifications
- You then map Erlang modules to schema types
- Creates relationship between type and code

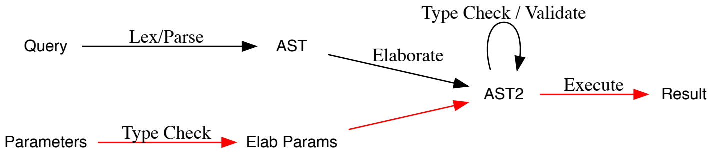
```
%% In Schema spec:
type Planet implements Node {
  id : ID!
  name : String
  diameter : Int
  rotationPeriod : Int
  orbitalPeriod : Int
  ...
}

%% In Erlang code:
#{ 'Planet' => sw_core_planet, ... }
```

# Erlang Implementation

Insight: The GraphQL system is a programming language

- Turn GQL query documents into (optimized) query plans
- Currently about 1/3 of the official de-facto Node.js implementation
- Almost feature complete
- Many other engines use an Object-Oriented visitor pattern scheme. We thought we could use a functional approach



# Lexing and Parsing

- Standard Erlang lexer generator leex
- Could be hand rolled
- Not on the critical path

# Elaboration

- Trick from Standard ML compilers (type inference, defunctorization, phase splitting etc)
- Elaborate the query by annotating schema types
- Makes the later stages far easier to write
- Not on the critical path

# Type Check and Validation

- Fairly standard type checker
- Validator steps further verifies query document correctness for common mistakes.
- Not on the critical path
- Note: digression from the spec—Push more things to the type checker where it belongs. Push more to the elaborator where it belongs.

# Execution

- Runs the query
- On the critical path!
- Uses user-supplied “resolver” modules to resolve the actual data query.
- Resolvers can be backed any code you want
- Note: we resolve by modules whereas everyone else resolves by functions. (Pattern matching FTW!)

# Resolver example: Planets

```
execute(_Ctx, #planet { id = PlanetId } = Planet, Field, Args) ->
    case Field of
        <<"id">> -> {ok, sw_core_id:encode({'Planet', Planet#planet.id})};
        <<"edited">> -> {ok, Planet#planet.edited};
        <<"climate">> -> {ok, Planet#planet.climate};
        <<"surfaceWater">> -> {ok, Planet#planet.surface_water};
        <<"name">> -> {ok, Planet#planet.name};
        <<"diameter">> -> {ok, integer(Planet#planet.diameter)};
        <<"rotationPeriod">> -> {ok, integer(Planet#planet.rotation_period)};
        ...;
    end
```

## Generic resolver:

```
execute(_Ctx, Obj, Field, _Args) ->
    {ok, maps:get(Field, Obj, null)}.
```

# Performance

- Only parameter checking and execution is time critical
- execution, even for large queries are measured in $\mu$s, usually in the 5–10 range
- Fetching data is measured in ms and some times much higher
- Your efficiency kernel is likely to be in data fetching
- Allows our code to be cleaner as efficiency isn't that important
- Can really play Erlang's concurrency strength here

# Further Work

- Introduce a small functional language as the IR
- Translate GraphQL to IR, type check IR
- Hunch: This is way easier
- Type system obviously has modes/polarity in it
- Want to formalize type system in Coq/Agda/Twelf at some point (Twelf is alluring if we manage to build a $\lambda$-calc based IR)
- QuickCheck approaches are obvious for testing as well

# Further Work (2)

- Some validations are still missing
- The code is somewhat mature, but has failing corner cases
- Concurrent/Parallel query execution is not yet in. Foci: correctness first
- Some older ideas in the system can be cut out
- Build a dedicated handler for Cowboy (awaits Cowboy 2.0)

# Wanna try it?

- Code is at `https://github.com/shopgun/graphql-erlang`
- Tutorial is at `https://github.com/shopgun/graphql-erlang-tutorial`
- Tutorial can be viewed at `https://shopgun.github.io/graphql-erlang-tutorial/`

QUESTIONS?

# What went right

- Write a tutorial early
- Documentation forces specification
- Iterate the solution. The current one is iteration 3
- Don't care about efficiency early

# Subscriptions

- Method to subscribe to updates on an object
- Rather new functionality, not yet part of the draft spec
- Works just like a mutation, however, trivially implemented

# Authentication

- Pass around a context to each resolver.
- Store Authentication/Authorization info in the context.
- Write the resolver such that it inspects the context for auth information.
- special objects: me, viewer, ....