Grackle: Functional GraphQL for the Typelevel stack

Miles Sabin, @milessabin@types.pl

TYPELEVEL
SCALA

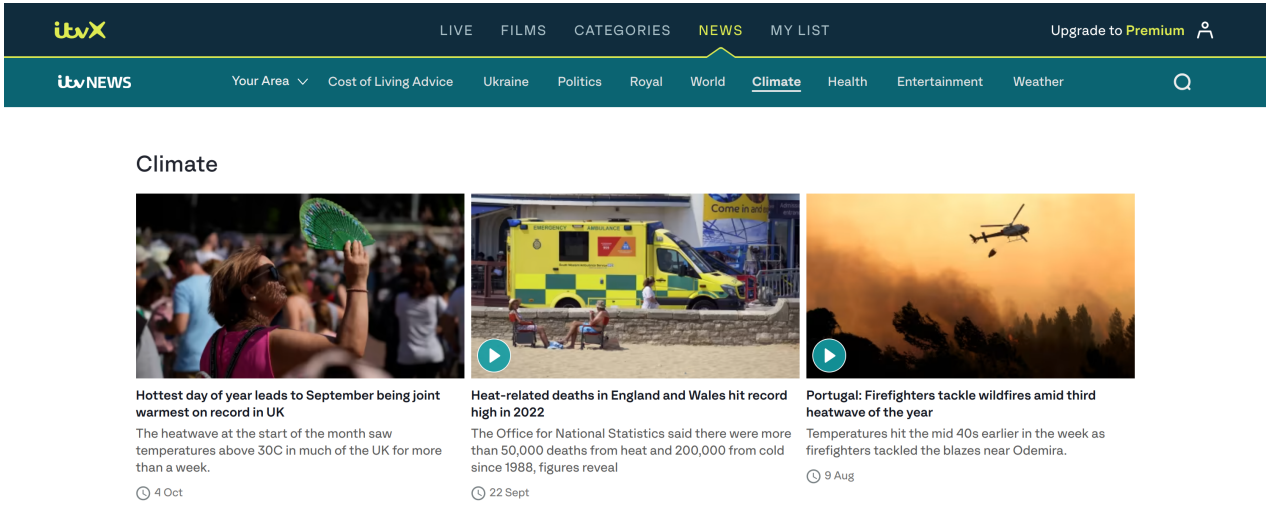
Introducing Grackle

Introducing Grackle

- GraphQL is a query language for APIs
- Grackle is a GraphQL server built on the Typelevel stack
- Work sponsored by ITV and Aura/Gemini over the last four years

Grackle at ITV — Content metadata

Grackle at ITV — News & Distribution



Grackle at Aura/Gemini

Gemini South on the summit of Cerro Pachón in Chile (left) and Gemini North on the summit of Maunakea in Hawai'i (right). Image credit: Gemini/NSF/AURA

What is GraphQL?

What is GraphQL?

- Open API standard Created by Facebook (https://graphql.org/)
- Structured query and update language for APIs
- Safe, flexible and evolvable alternative to REST
  - Requests/responses defined by strongly typed schemas
  - Clients choose what to fetch
  - Server returns response with the same shape as queries

GraphQL Demo

What is Grackle?

What is Grackle?

- GraphQL server powered by cats, cats-effect, fs2 and http4s
- Supports GraphQL queries, mutations and subscriptions
- Has an abstract model of data sources via mappings and cursors
  - Supports in-memory, DB backed, and effectful data sources

What is Grackle?

- Grackle is structured as a compiler/interpreter
- Queries are type-checked against the GraphQL schema
- Queries are compiled into an internal query algebra
- The query algebra may be further compiled in a backend-specific way to materialize data
  - In particular it can be compiled to efficient SQL
  - Currently supports Postgres via Doobie or Skunk

What is Grackle?

- The query is linked to backing data via,
  - The compiled query algebra term
  - An elaboration phase which gives meaning to application specifics
  - A backend-specific declarative mapping
  - A backend-specific cursor into the backing data
- The elaborated query algebra term is executed, driving the cursor, generating the result

In-memory models

In-memory models

```
type Query {
  character(id: ID!): Character

  # ...
}


interface Character {
  name: String!

  # ...
}
```

In-memory models

```
query {
  character(id: 1000) {
    name
  }
}
```

In-memory models

```
case class UntypedSelect(
  name: String, alias: Option[String],
  args: List[Binding], directives: List[Directive],
  child: Query
)
case class Select(name: String, alias: Option[String], child: Query)
case class Group(queries: List[Query])
case class Unique(child: Query)
case class Filter(pred: Predicate, child: Query)
case class Introspect(schema: Schema, child: Query)
case class Environment(env: Env, child: Query)
case class Narrow(subtpe: TypeRef, child: Query)
case class Limit(num: Int, child: Query)
case class Offset(num: Int, child: Query)
case class OrderBy(selections: OrderSelections, child: Query)
case class Count(child: Query)
case class TransformCursor(f: Cursor => Result[Cursor], child: Query)
case class Component[F[_]](mapping: Mapping[F], ...)
case class Effect[F[_]](handler: EffectHandler[F], child: Query)
case object Empty
```

In-memory models

```
UntypedSelect(
  "character",
  None,
  List(IntBinding("id", 1000)),
  Nil,
  UntypedSelect(
    "name", None, Nil, Nil, Empty
  )
)
```

In-memory models

```
case (QueryType, "character", List(Binding("id", IDValue(id)))) =>
  Elab.transformChild { child =>
    Unique(Filter(Eql(CharacterType / "id", Const(id)), child))
  }
```

In-memory models

```
Select(
  "character",
  None,
  Unique(
    Filter(
      Eql(CharacterType / "id", Const("1000")),
      Select("name", None, Empty)
    )
  )
)
```

In-memory models

```
val characters: List[Character] = ...

...

ObjectMapping(
  tpe = QueryType,
  fieldMappings =
    List(
      GenericField("character", characters)
    )
)
```

In-memory models

```
{
  "data": {
    "character": {
      "name": "Luke Skywalker"
    }
  }
}
```

DB backed models

DB backed models

```
type Query {
  country(code: String): Country
}

type Country {
  name: String!
  cities: [City!]!
}

type City {
  name: String!
  population: Int!
}
```

DB backed models

```
query {
  country(code: "GBR") {
    name
    cities {
      name
    }
  }
}
```

DB backed models

```
Select(
  "country", None,
  Unique(
    Filter(Eql(CountryType / "code", Const("GBR")),
      Group(
        Select("name", None, Empty),
        Select("cities", None,
          Select("name", None, Empty)
        )
      )
    )
  )
)
```

DB backed models

```
CREATE TABLE country
(
  code        character(3) NOT NULL,
  name        text         NOT NULL
);

CREATE TABLE city
(
  id          integer      NOT NULL,
  name        text         NOT NULL,
  countrycode character(3) NOT NULL
);
```

DB backed models

```
object country extends TableDef("country") {
  val code        = col("code", Meta[String])
  val name        = col("name", Meta[String])
}

object city extends TableDef("city") {
  val id          = col("id", Meta[Int])
  val countrycode = col("countrycode", Meta[String])
  val name        = col("name", Meta[String])
}
```

DB backed models

```
ObjectMapping(
  tpe = CountryType,
  fieldMappings = List(
    SqlField("code",         country.code, key = true),
    SqlField("name",         country.name),
    SqlObject("cities",      Join(country.code, city.countrycode))
  )
)
ObjectMapping(
  tpe = CityType,
  fieldMappings = List(
    SqlField("id",           city.id, key = true, hidden = true),
    SqlField("countrycode", city.countrycode, hidden = true),
    SqlField("name",         city.name)
  )
)
```

DB backed models

```
SELECT
  city.id,
  city.name,
  country.code,
  country.name
FROM
  country
  LEFT JOIN city ON (city.countrycode = country.code)
WHERE
  country.code = 'GBR';
```

DB backed models

```
{
  "data": {
    "country": {
      "name": "United Kingdom",
      "cities": [
        { "name": "Gloucester" },
        { "name": "Ipswich" },
        { "name": "Rochdale" },
        ...
      ]
    }
  }
}
```

Project status

Project status

Things I haven't had time to talk about today ...

- Nested effects with batching
- Composition of mappings and schemas

Project status

Things I haven't had time to talk about today ...

- Grackle has full support for custom query and schema directives
  - Examples in the repo demonstrate authentication and authorization
- Query algebra and elaboration model supports implementation of things that go beyond the GraphQL spec, eg.
  - Paging
  - Filter/attribute cascading

Project status

- Grackle is an Apache 2.0 licensed Typelevel project
  - https://github.com/typelevel/grackle
- Available for Scala 2/3 and for Scala.js and Scala Native
- Current version 0.15.0 (version 1.0 soon!)
- Still evolving but battle tested on highly visible, high traffic sites

Project status

- Demo and tutorial
  - https://typelevel.org/grackle
- Lots of scope for contributions ... come and join us!
  - Compilers are fun!
  - More backends (eg. KV stores, MongoDB, ElasticSearch, etc.)
  - Lots of scope for building higher level abstractions
  - `#grackle` on the Typelevel discord

Questions?

Thank You

Miles Sabin, @milessabin@types.pl

http://typelevel.org/grackle

TYPELEVEL
SCALA