com.walmartlabs/lacinia Documentation

Release 0.32.0-rc-3

Walmartlabs

Jan 25, 2019

Contents

Lacinia is a library for implementing Facebook's GraphQL specification in idiomatic Clojure.

GraphQL is a way for clients to efficiently obtain data from servers.

Compared to traditional REST approaches, GraphQL ensures that clients can access exactly the data that they need (and no more), and do so with fewer round-trips to the server.

This is especially useful for mobile clients, where bandwidth is always at a premium.

In addition, GraphQL is self describing; the shape of the data that can be exposed, and the queries by which that data can be accessed, are all accessible using GraphQL queries. This allows for sophisticated, adaptable clients, such as the in-browser GraphQL IDE graphiql.

Warning: This library is still under active development and should not be considered complete. If you would like to contribute, please create a pull request.

Although GraphQL is quite adept at handling requests from client web browsers and responding with JSON, it is also exceptionally useful for allowing backend systems to communicate.

CHAPTER 1

Overview

GraphQL consists of two main parts:

- A server-side *schema* that defines the available queries and types of data that may be returned.
- A client query language that allows the client to specify what query to execute, and what data to return.

The GraphQL specification goes into detail about the format of the client query language, and the expected behavior of the server.

This library, Lacinia, is an implementation of the key component of the server, in idiomatic Clojure.

1.1 Schema

The GraphQL specification includes a language to define the server-side schema; the `type` keyword is used to introduce a new kind of object.

In Lacinia, the schema is Clojure data: a map of keys and values; top level keys indicate the type of data being defined:

```
{:enums
 {:episode
  {:description "The episodes of the original Star Wars trilogy."
   :values [:NEWHOPE :EMPIRE :JEDI]}}

 :interfaces
 {:character
  {:fields {:id {:type String}
            :name {:type String}
            :appears_in {:type (list :episode)}
            :friends {:type (list :character)}}}}

 :objects
 {:droid
  {:implements [:character]
   :fields {:id {:type String}
```

(continues on next page)

(continued from previous page)

```
            :name {:type String}
            :appears_in {:type (list :episode)}
            :friends {:type (list :character)
                      :resolve :friends}
            :primary_function {:type (list String)}}}

  :human
  {:implements [:character]
   :fields {:id {:type String}
            :name {:type String}
            :appears_in {:type (list :episode)}
            :friends {:type (list :character)
                      :resolve :friends}
            :home_planet {:type String}}}}

 :queries
 {:hero {:type (non-null :character)
         :args {:episode {:type :episode}}
         :resolve :hero}

  :human {:type (non-null :human)
          :args {:id {:type String
                      :default-value "1001"}}
          :resolve :human}

  :droid {:type :droid
          :args {:id {:type String
                      :default-value "2001"}}
          :resolve :droid}}}
```

Here, we are defining *human* and *droid* objects. These have a lot in common, so we define a shared *character* interface.

But how to access that data? That's accomplished using one of three queries:

- hero
- human
- droid

In this example, each query returns a single instance of the matching object. Often, a query will return a list of matching objects.

1.2 Injecting Data

The schema defines the *shape* of the data that can be queried, but leaves out where that data comes from. Unlike an object/relational mapping layer, where we might discuss database tables and rows, GraphQL (and by extension, Lacinia) has *no* idea where the data comes from.

That's the realm of the *field resolver function*. Since EDN files are just data, we leave placeholder keywords in the EDN data and attach the actual functions once the EDN data is read into memory.

1.3 Compiling the Schema

The schema starts as a data structure, we need to add in the field resolver and then *compile* the result.

```
(ns org.example.schema
  (:require
    [clojure.edn :as edn]
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.schema :as schema]
    [com.walmartlabs.lacinia.util :as util]
    [org.example.db :as db]))

(defn star-wars-schema
  []
  (-> (io/resource "star-wars-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers {:hero db/resolve-hero
                              :human db/resolve-human
                              :droid db/resolve-droid
                              :friends db/resolve-friends})
      schema/compile))
```

The `attach-resolvers` function walks the schema tree and replaces the values for `:resolve` keys. With actual functions in place, the schema can be compiled for execution.

Compilation performs a number of checks, applies defaults, merges in introspection data about the schema, and performs a number of other operations to ready the schema for use. The structure passed into `compile` is quite complex, so it is always validated using *clojure.spec*.

1.4 Parsing GraphQL IDL Schemas

Lacinia also offers support for parsing schemas defined in the GraphQL Interface Definition Language and tranforming them into the Lacinia schema data structure.

See *GraphQL IDL Schema Parsing* for details.

1.5 Executing Queries

With that in place, we can now execute queries.

```
(require
  '[com.walmartlabs.lacinia :refer [execute]]
  '[org.example.schema :refer [star-wars-schema]])

(def compiled-schema (star-wars-schema))

(execute compiled-schema
  "query { human(id: \"1001\") { name }}"
  nil nil)
=> {:data {:human #ordered/map([:name "Darth Vader"])}}
```

Ordered Map?

The `#ordered/map` indicates that the fields in the result map are returned in the *same order*[1] as they are specified in the query document.

In most examples, for conciseness, a standard (unordered) map is shown.

The query string is parsed and matched against the queries defined in the schema.

The two nils are variables to be used executing the query, and an application context.

In GraphQL, queries can pass arguments (such as `id`) and queries identify exactly which fields of the matching objects are to be returned. This query can be stated as *just provide the name of the human with id '1001'*.

This is a successful query, it returns a result map[2] with a `:data` key. A failed query would return a map with an `:errors` key. A query can even be partially successful, returning as much data as it can, but also errors.

Inside `:data` is a key corresponding to the query, `:human`, whose value is the single matching human. Other queries might return a list of matches. Since we requested just a slice of a full human object, just the human's name, the map has just a single `:name` key.

[1] This shouldn't be strictly necessary (JSON and EDN don't normally care about key order, and keys can appear in arbitrary order), but having consistent ordering makes writing tests involving GraphQL queries easier: you can typically check the textual, not parsed, version of the result map directly against an expected string value.

[2] In GraphQL's specification, this is referred to as the "response"; in practice, this result data forms the body of a response map (when using Ring or Pedestal). Lacinia uses the terms *result map* or *result data* to keep these ideas distinct.

CHAPTER 2

Tutorial

2.1 Pre-Requisites

You should be familiar with, but by no means an expert in, Clojure.

You should have Leiningen, the Clojure build tool, installed and be familiar with editing a `project.clj` file.

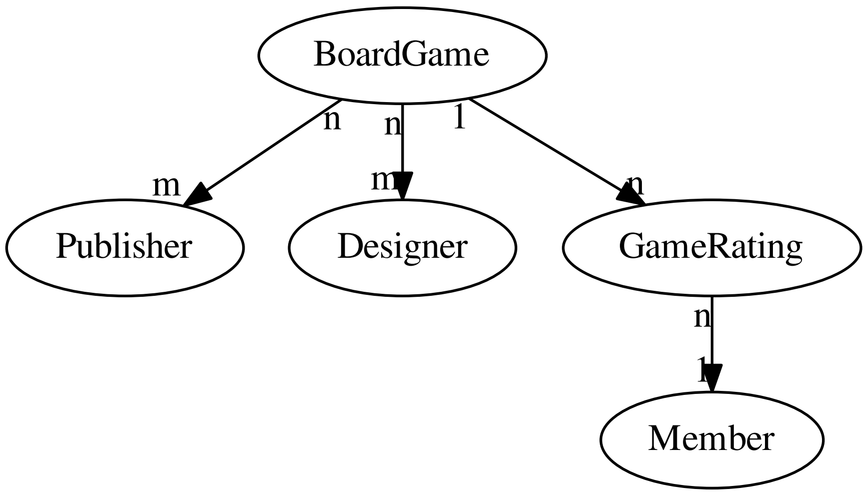
You should have an editor or IDE ready to go, set up for editing Clojure code.

A skim of the Lacinia reference documentation (the rest of this manual, outside of this tutorial) is also helpful. Or you can follow links provided as we go.

2.2 Domain

Our goal will be to provide a GraphQL interface to data about board games (one of the author's hobbies), as a limited version of Board Game Geek.

The basic types in the final system are as follows:



A BoardGame may be published by multiple Publisher companies (the Publisher may be different for different countries, or may simply vary over time).

A BoardGame may have any number of Designers.

Users of Clojure Game Geek, represented as type Member, may provide their personal ratings for board games.

This is a tiny sliver of the functionality of Board Game Geek, but sufficiently meaty to give us a taste for building full applications.

2.3 Creating the Project

In this first step of the tutorial, we'll create the initial empty project, and set up the initial dependencies.

Let's get started.

The first step is to create a new, empty project:

```
12:51:44 ~/workspaces/github > lein new clojure-game-geek
Generating a project called clojure-game-geek based on the 'default' template.
The default template is intended for library projects, not applications.
To see other templates (app, plugin, etc), try `lein help new`.
```

Although Leiningen has an `app` template, it is not significantly different than the default template.

Normally, we might delete some files we don't need, such as `.hgignore` and the default `doc` directory, but for simplicity we'll just ignore those.

Initially, the `project.clj` is almost empty:

Listing 1: project.clj

```
(defproject clojure-game-geek "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
```

(continues on next page)

(continued from previous page)

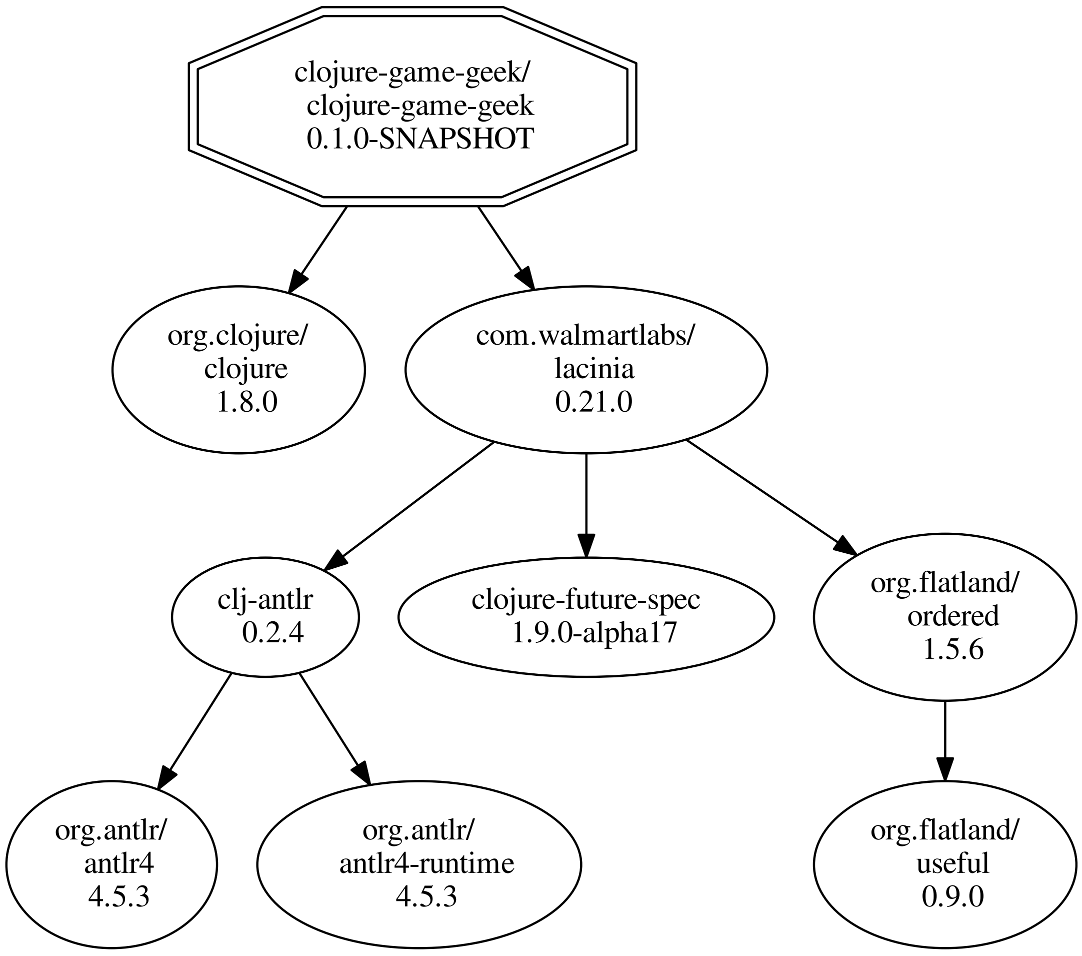
```
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]])
```

Our first step is to fill in a few details, including the dependency on Lacinia:

Listing 2: project.clj

```
(defproject clojure-game-geek "0.1.0-SNAPSHOT"
  :description "A tiny BoardGameGeek clone written in Clojure with Lacinia"
  :url "https://github.com/walmartlabs/clojure-game-geek"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [com.walmartlabs/lacinia "0.21.0"]])
```

Lacinia has just a few dependencies of its own:


clojure-game-geek/ clojure-game-geek 0.1.0-SNAPSHOT
org.clojure/ clojure 1.8.0
com.walmartlabs/ lacinia 0.21.0
clj-antlr 0.2.4
clojure-future-spec 1.9.0-alpha17
org.flatland/ ordered 1.5.6
org.antlr/ antlr4 4.5.3
org.antlr/ antlr4-runtime 4.5.3
org.flatland/ useful 0.9.0

Antlr is used to parse GraphQL queries and schemas. `org.flatland/ordered` provides the ordered map type, used to ensure that response keys and values are in the client-specified order, as per the GraphQL spec.

Like any open source project, Lacinia is evolving all the time. In later tutorial steps, we will change these dependencies to access newly added Lacinia features.

2.4 Initial Schema

At this stage, we're still just taking baby steps, and getting our bearings.

By the end of this stage, we'll have a minimal schema and be able to execute our first query.

2.4.1 Schema EDN File

Our initial schema is just for the BoardGame entity, and a single operation to retrieve a game by its id:

Listing 3: resources/cgg-schema.edn

```
{:objects
 {:BoardGame
  {:description "A physical or virtual board game."
   :fields
   {:id {:type (non-null ID)}
    :name {:type (non-null String)}
    :summary {:type String
              :description "A one-line summary of the game."}
    :description {:type String
                  :description "A long-form description of the game."}
    :min_players {:type Int
                  :description "The minimum number of players the game supports."}
    :max_players {:type Int
                  :description "The maximum number of players the game supports."}
    :play_time {:type Int
                :description "Play time, in minutes, for a typical game."}}}}

 :queries
 {:game_by_id
  {:type :BoardGame
   :description "Access a BoardGame by its unique id, if it exists."
   :args
   {:id {:type ID}}
   :resolve :query/game-by-id}}}
```

Details

See documentation about *Objects*, *Fields*, and *Queries*.

A Lacinia schema is an EDN file. It is a map of maps; the top level keys identify the type of definition: `:objects`, `:queries`, `:interfaces`, `:enums`, and so forth. The inner maps are keywords to a type-specific structure. This schema defines a single query, `game_by_id` that returns an object as defined by the `BoardGame` type.

A schema is declarative: it defines what operations are possible, and what types and fields exist, but has nothing to say about where any of the data comes from. In fact, Lacinia has no opinion about that either! GraphQL is a contract between a consumer and a provider for how to request and present data, it's not any form of database layer, object relational mapper, or anything similar.

Instead, Lacinia handles the parsing of a client query, and guides the execution of that query, ultimately invoking application-specific callback hooks: *field resolvers*. Field resolvers are the only source of actual data. Ultimately, field resolvers are simple Clojure functions, but those can't, and shouldn't, be expressed inside an EDN file. Instead we put a placeholder in the EDN, and then *attach* the actual resolver later.

The keyword `:query/game-by-id` is just such a placeholder; we'll see how it is used shortly.

We've made liberal use of the `:description` property in the schema. These descriptions are intended for developers who will make use of your GraphQL interface. Descriptions are the equivalent of doc-strings on Clojure functions, and we'll see them show up later when we *discuss GraphiQL*. It's an excellent habit to add descriptions early, rather than try and go back and add them in later.

We'll add more fields, more types, relationships between types, and more operations in later chapters.

We've also demonstrated the use of a few Lacinia conventions in our schema:

- Built-in scalar types, such as ID, String, and Int are referenced as symbols.[1]
- Schema-defined types, such as `:BoardGame`, are referenced as keywords.
- Fields are lower-case names, and types are CamelCase.

In addition, all GraphQL names (for fields, types, and so forth) must contain only alphanumerics and the underscore. The dash character is, unfortunately, not allowed. If we tried to name the query `query-by-id`, Lacinia would throw a clojure.spec validation exception when we attempted to use the schema.[2]

In Lacinia, there are base types, such as `String` and `:BoardGame` and wrapped types, such as `(non-null String)`. The two wrappers are `non-null` (a value *must* be present) and `list` (the type is a list of values, not a single value). These can even be combined!

Notice that the return type of the `game_by_id` query is `:BoardGame` and *not* `(non-null :BoardGame)`. This is because we can't guarantee that a game can be resolved, if the id provided in the client query is not valid. If the client provides an invalid id, then the result will be nil, and that's not considered an error.

In any case, this single BoardGame entity is a good starting point.

2.4.2 schema namespace

With the schema defined, the next step is to write code to load the schema into memory, and make it operational for queries:

Listing 4: src/clojure_game_geek/schema.clj

```
(ns clojure-game-geek.schema
  "Contains custom resolvers and a function to provide the full schema."
  (:require
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.util :as util]
    [com.walmartlabs.lacinia.schema :as schema]
    [clojure.edn :as edn]))

(defn resolver-map
  []
```

(continues on next page)

[1] Internally, *everything* is converted to keywords, so if you prefer to use symbols everywhere, nothing will break. This is part of the schema compilation process.

[2] Because the input schema format is so complex, it *always* validated using clojure.spec. This helps to ensure that minor typos or other gaffes are caught early rather than causing you great confusion later.

(continued from previous page)

```
  {:query/game-by-id (fn [context args value]
                       nil)})

(defn load-schema
  []
  (-> (io/resource "cgg-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers (resolver-map))
      schema/compile))
```

This code loads the schema EDN file, *attaches field resolvers* to the schema, then *compiles* the schema. The compilation step is necessary before it is possible to execute queries. Compilation reorganizes the schema, computes various defaults, perform verifications, and does a number of other necessary steps.

We're using a namespaced keyword for the resolver in the schema, and in the `resolver-map` function; this is a good habit to get into early, before your schema gets very large.

The field resolver in this case is a placeholder; it ignores all the arguments passed to it, and simply returns nil. Like all field resolver functions, it accepts three arguments: a context map, a map of field arguments, and a container value. We'll discuss what these are and how to use them shortly.

2.4.3 user namespace

Not too much!

An annoyance with putting code into the `user` namespace is that you can't start a new REPL unless and until the `user` namespace loads. Every so often, you have to go in your `user` namespace and comment everything out just to get a REPL running, to start debugging an error elsewhere.

A key advantage of Clojure is REPL-oriented[3] development: we want to be able to run our code through its paces almost as soon as we've written it - and when we change code, we want to be able to try out the changed code instantly.

Clojure, by design, is almost uniquely good at this interactive style of development. Features of Clojure exist just to support REPL-oriented development, and its one of the ways in which using Clojure will vastly improve your productivity!

We can add a bit of scaffolding to the `user` namespace, specific to our needs in this project. When you launch a REPL, it always starts in this namespace.

We can define the user namespace in the `dev-resources` folder; this ensures that it is not packaged up with the rest of our application when we eventually package and deploy the application.

Listing 5: dev-resources/user.clj

```
(ns user
  (:require
    [clojure-game-geek.schema :as s]
    [com.walmartlabs.lacinia :as lacinia]))
```

(continues on next page)

[3] Read Eval Print Loop: you type in an expression, and Clojure evaluates and prints the result. This is an innovation that came early to Lisps, and is integral to other languages such as Python, Ruby, and modern JavaScript. Stuart Halloway has a talk, Running with Scissors: Live Coding With Data, that goes into a lot more detail on how important and useful the REPL is.

(continued from previous page)

```
(def schema (s/load-schema))

(defn q
  [query-string]
  (lacinia/execute schema query-string nil nil))
```

The key function is `q`, which invokes com.walmartlabs.lacinia/execute.

We'll use that to test GraphQL queries against our schema and see the results directly in the REPL: no web browser necessary!

With all that in place, we can launch a REPL and try it out:

```
14:26:41 ~/workspaces/github/clojure-game-geek > lein repl
nREPL server started on port 56053 on host 127.0.0.1 - nrepl://127.0.0.1:56053
REPL-y 0.3.7, nREPL 0.2.12
Clojure 1.8.0
Java HotSpot(TM) 64-Bit Server VM 1.8.0_74-b02
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
    Exit: Control+D or (exit) or (quit)
 Results: Stored in vars *1, *2, *3, an exception in *e

user=> (q "{ game_by_id(id: \"foo\") { id name summary }}")
{:data #ordered/map ([:game_by_id nil])}
```

The value returned makes use of an ordered map. Again, that's part of the GraphQL specification: the order in which things appear in the query dictates the order in which they appear in the result. In any case, this result is equivalent to `{:data {:game_by_id nil}}`.

That's as it should be: the resolver was unable to resolve the provided id to a BoardGame, so it returned nil. This is not an error . . . remember that we defined the type of the `game_by_id` operation to allow nulls, just for this specific situation.

However, Lacinia still returns a map with the operation name and operation selection. Failure to return a result with a `:data` key would signify an error executing the query, such as a parse error. That's not the case here at all.

2.4.4 Summary

We've defined an exceptionally simple schema in EDN, but still have managed to load it into memory and compile it. We've also used the REPL to execute a query against the schema and seen the initial (and quite minimal) result.

In the next chapter, we'll build on this modest start, introducing more schema types, and

2.5 Placeholder Game Data

It would be nice to do some queries that actually return some data!

One option would be to fire up a database, define some tables, and load some data in.

. . . but that would slow us down, and not teach us anything about Lacinia and GraphQL.

Instead, we'll create an EDN file with some test data in it, and wire that up to the schema. We can fuss with database access and all that later in the tutorial.

2.5.1 cgg-data.edn

Listing 6: dev-resources/cgg-data.edn

```
{:games
 [{:id "1234"
   :name "Zertz"
   :summary "Two player abstract with forced moves and shrinking board"
   :min_players 2
   :max_players 2
   }
  {:id "1235"
   :name "Dominion"
   :summary "Created the deck-building genre; zillions of expansions"
   :min_players 2}
  {:id "1236"
   :name "Tiny Epic Galaxies"
   :summary "Fast dice-based sci-fi space game with a bit of chaos"
   :min_players 1
   :max_players 4}
  {:id "1237"
   :name "7 Wonders: Duel"
   :summary "Tense, quick card game of developing civilizations"
   :min_players 2
   :max_players 2}]}
```

This file defines just a few games I've recently played. It will take the place of an external database. Later, we can add more data for the other entities and their relationships.

Later still, we'll actually connect our application up to an external database.

2.5.2 Resolver

Inside our `schema` namespace, we need to read the data and provide a resolver that can access it.

Listing 7: src/clojure_game_geek/schema.clj

```
(ns clojure-game-geek.schema
  "Contains custom resolvers and a function to provide the full schema."
  (:require
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.util :as util]
    [com.walmartlabs.lacinia.schema :as schema]
    [clojure.edn :as edn]))

(defn resolve-game-by-id
  [games-map context args value]
  (let [{:keys [id]} args]
    (get games-map id)))

(defn resolver-map
  []
  (let [cgg-data (-> (io/resource "cgg-data.edn")
                     slurp
                     edn/read-string)
        games-map (->> cgg-data
```

(continues on next page)

(continued from previous page)

```
                        :games
                        (reduce #(assoc %1 (:id %2) %2) {}))]
    {:query/game-by-id (partial resolve-game-by-id games-map)}))

(defn load-schema
  []
  (-> (io/resource "cgg-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers (resolver-map))
      schema/compile))
```

The `attach-resolvers` function walks a schema, locating the `:resolve` keys and swapping the placeholder keywords with actual functions. It needs a map of placeholder keywords to field resolver functions; that map is provided by the `resolver-map` function.

You can see a bit of the philosophy of Lacinia inside the `load-schema` function: Lacinia strives to provide only what is most essential, or truly useful and universal.

Lacinia explicitly *does not* provide a single function to read, parse, attach resolvers, and compile an EDN file in a single line. That may seem odd – it feels like very application will just cut-and-paste something virtually identical to `load-schema`.

In fact, not all schemas will come directly from a single EDN file. Because the schema is Clojure *data* it can be constructed, modified, merged, and otherwise transformed right up to the point that it is compiled. By starting with a pipeline like the one inside `load-schema`, it becomes easy to inject your own application-specific bits into the steps leading up to `schema/compile`, which ultimately becomes quite essential.

Back to the schema; the resolver itself is the `resolve-game-by-id` function. It is provided with a map of games, and the standard triumvirate of resolver function arguments: context, field arguments, and container value.

Field resolvers are passed a map of the field arguments (from the client query). This map contains keyword keys, and values of varying types (because field arguments have a type in the GraphQL schema).

We use a bit of destructuring to extract the id[1]. The data in the map is already in a form that matches the GraphQL schema, so it's just a matter of `get`-ing it out of the games map.

Inside `resolver-map`, we read the sample game data, then use typical Clojure data manipulation to get it into the form that we want: we convert a seq of BoardGame maps into a map of maps, keyed on the `:id` of each BoardGame.

The `partial` function is a real workhorse in Clojure code; it takes an existing function and a set of initial arguments to that function and returns a new function that collects the remaining arguments needed by the original function. This returned function will accept the standard field resolver arguments – `context`, `args`, and `value`, and pass the `games-map` and those arguments to `resolve-game-by-id`.

This is one common example of the use of *higher orderered functions*. It's not as complicated as the term might lead you to believe - just that functions can be arguments to, and return values from, other functions.

2.5.3 Running Queries

We're finally almost ready to run queries … but first, let's get rid of that `#ordered/map` business.

[1] This is overkill for this very simple case, but it's nice to demonstrate techniques that are likely to be used in real applications.

Listing 8: dev-resources/user.clj

```
(ns user
  (:require
    [clojure-game-geek.schema :as s]
    [com.walmartlabs.lacinia :as lacinia]
    [clojure.walk :as walk])
  (:import (clojure.lang IPersistentMap)))

(def schema (s/load-schema))

(defn simplify
  "Converts all ordered maps nested within the map into standard hash maps, and
   sequences into vectors, which makes for easier constants in the tests, and␣
↪eliminates ordering problems."
  [m]
  (walk/postwalk
    (fn [node]
      (cond
        (instance? IPersistentMap node)
        (into {} node)

        (seq? node)
        (vec node)

        :else
        node))
    m))

(defn q
  [query-string]
  (-> (lacinia/execute schema query-string nil nil)
      simplify))
```

This simplify function finds all the ordered maps and converts them into ordinary maps. It also finds any lists and converts them to vectors.

With that in place, we're ready to run some queries:

```
(q "{ game_by_id(id: \"anything\") { id name summary }}")
=> {:data {:game_by_id nil}}
```

This hasn't changed[2], except that, because of simplify, the final result is just standard maps, which are easier to look at in the REPL.

However, we can also get real data back from our query:

```
(q "{ game_by_id(id: \"1236\") { id name summary }}")
=>
{:data {:game_by_id {:id "1236",
                     :name "Tiny Epic Galaxies",
                     :summary "Fast dice-based sci-fi space game with a bit of chaos"}
↪}}
```

[2] This REPL output is a bit different than earlier examples; we've switched from the standard Leiningen REPL to the Cursive REPL; the latter pretty-prints the returned values.

Where's the JSON?

It's perfectly acceptable to return EDN rather than JSON. The GraphQL specification goes to some length to identify JSON as just one possible over-the-wire format. It's easy enough to convert EDN to JSON, and even reasonable to support clients that can consume the EDN directly.

Success! Lacinia has parsed our query string and executed it against our compiled schema. At the correct time, it dropped into our resolver function, which supplied the data that it then sliced and diced to compose the result map.

You should be able to devise and execute other queries at this point.

2.5.4 Summary

We've extended our schema and field resolvers with test data and are getting some actual data back when we execute a query.

Next up, we'll continue extending the schema, and start discussing relationships between GraphQL types.

2.6 Adding Designers

So far, we've been working with just a single entity type, BoardGame.

Let's see what we can do when we add the Designer entity type to the mix.

Initially, we'll define each Designer in terms of an id, a name, and an optional home page URL.

Listing 9: dev-resources/cgg-data.edn

```
{:games
 [{:id "1234"
   :name "Zertz"
   :summary "Two player abstract with forced moves and shrinking board"
   :min_players 2
   :max_players 2
   :designers #{"200"}}
  {:id "1235"
   :name "Dominion"
   :summary "Created the deck-building genre; zillions of expansions"
   :designers #{"204"}
   :min_players 2}
  {:id "1236"
   :name "Tiny Epic Galaxies"
   :summary "Fast dice-based sci-fi space game with a bit of chaos"
   :designers #{"203"}
   :min_players 1
   :max_players 4}
  {:id "1237"
   :name "7 Wonders: Duel"
   :summary "Tense, quick card game of developing civilizations"
   :designers #{"201" "202"}
   :min_players 2
   :max_players 2}]

 :designers
 [{:id "200"
```

(continues on next page)

(continued from previous page)

```
   :name "Kris Burm"
   :url "http://www.gipf.com/project_gipf/burm/burm.html"}
  {:id "201"
   :name "Antoine Bauza"
   :url "http://www.antoinebauza.fr/"}
  {:id "202"
   :name "Bruno Cathala"
   :url "http://www.brunocathala.com/"}
  {:id "203"
   :name "Scott Almes"}
  {:id "204"
   :name "Donald X. Vaccarino"}]}
```

If this was a relational database, we'd likely have a join table between BoardGame and Designer, but that can come later. For now, we have a set of designer *ids* inside each BoardGame.

2.6.1 Schema Changes

Listing 10: resources/cgg-schema.edn

```
{:objects
 {:BoardGame
  {:description "A physical or virtual board game."
   :fields
   {:id {:type (non-null ID)}
    :name {:type (non-null String)}
    :summary {:type String
              :description "A one-line summary of the game."}
    :description {:type String
                  :description "A long-form description of the game."}
    :designers {:type (non-null (list :Designer))
                :description "Designers who contributed to the game."
                :resolve :BoardGame/designers}
    :min_players {:type Int
                  :description "The minimum number of players the game supports."}
    :max_players {:type Int
                  :description "The maximum number of players the game supports."}
    :play_time {:type Int
                :description "Play time, in minutes, for a typical game."}}}

  :Designer
  {:description "A person who may have contributed to a board game design."
   :fields
   {:id {:type (non-null ID)}
    :name {:type (non-null String)}
    :url {:type String
          :description "Home page URL, if known."}
    :games {:type (non-null (list :BoardGame))
            :description "Games designed by this designer."
            :resolve :Designer/games}}}}

 :queries
 {:game_by_id
  {:type :BoardGame
   :description "Access a BoardGame by its unique id, if it exists."
```

(continues on next page)

(continued from previous page)

```
   :args
   {:id {:type ID}}
   :resolve :query/game-by-id}}}
```

We've added a `:designers` field to BoardGame, and added a new Designer type.

In Lacinia, we use a wrapper, `list`, around a type, to denote a list of that type. In the EDN, the `list` wrapper is applied using the syntax of a function call in Clojure code.

A second wrapper, `non-null`, is used when a value must be present, and not null (or `nil` in Clojure). By default, all values *can be* nil and that flexibility is encouraged, so `non-null` is rarely used.

Here we've defined the `:designers` field as `(non-null (list :Designer))`. This is somewhat overkill (the world won't end if the result map contains a `nil` instead of an empty list), but demonstrates that the `list` and `non-null` modifiers can nest properly.

We could go further: `(non-null (list (non-null :Designer))) ...` but that's adding far more complexity than value.

Limits of Types

You can indicate that, for example, a list contains non-nil values, but there isn't anyway in GraphQL to signify a non-*empty* list.

We need a field resolver for the `:designers` field, to convert from what's in our data (a set of designer ids) into what we are promising in the schema: a list of Designer objects.

Likewise, we need a field resolver in the Designer entity to figure out which BoardGames are associated with the designer.

2.6.2 Code Changes

Listing 11: src/clojure_game_geek/schema.clj

```
(ns clojure-game-geek.schema
  "Contains custom resolvers and a function to provide the full schema."
  (:require
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.util :as util]
    [com.walmartlabs.lacinia.schema :as schema]
    [clojure.edn :as edn]))

(defn resolve-game-by-id
  [games-map context args value]
  (let [{:keys [id]} args]
    (get games-map id)))

(defn resolve-board-game-designers
  [designers-map context args board-game]
  (->> board-game
       :designers
       (map designers-map)))

(defn resolve-designer-games
```

(continues on next page)

(continued from previous page)

```
  [games-map context args designer]
  (let [{:keys [id]} designer]
    (->> games-map
         vals
         (filter #(-> % :designers (contains? id))))))

(defn entity-map
  [data k]
  (reduce #(assoc %1 (:id %2) %2)
          {}
          (get data k)))

(defn resolver-map
  []
  (let [cgg-data (-> (io/resource "cgg-data.edn")
                     slurp
                     edn/read-string)
        games-map (entity-map cgg-data :games)
        designers-map (entity-map cgg-data :designers)]
    {:query/game-by-id (partial resolve-game-by-id games-map)
     :BoardGame/designers (partial resolve-board-game-designers designers-map)
     :Designer/games (partial resolve-designer-games games-map)}))

(defn load-schema
  []
  (-> (io/resource "cgg-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers (resolver-map))
      schema/compile))
```

As with all field resolvers[1], `resolve-board-game-designers` is passed the containing resolved value (a BoardGame, in this case) and in turn, resolves the next step down, in this case, a list of Designers.

This is an important point: the data from your external source does not have to be in the shape described by your schema … you just must be able to transform it into that shape. Field resolvers come into play both when you need to fetch data from an external source, and when you need to reshape that data to match the schema.

GraphQL doesn't make any guarantees about order of values in a list field; when it matters, it falls on us to add documentation to describe the order, or even provide field arguments to let the client specify the order.

The inverse of `resolve-board-game-designers` is `resolve-designer-games`. It starts with a Designer and uses the Designer's id as a filter to find BoardGames whose `:designers` set contains the id.

2.6.3 Testing It Out

After reloading code in the REPL, we can exercise these new types and relationships:

```
(q "{ game_by_id(id: \"1237\") { name designers { name }}}")
=> {:data {:game_by_id {:name "7 Wonders: Duel",
                        :designers [{:name "Antoine Bauza"}
                                    {:name "Bruno Cathala"}]}}}
```

For the first time, we're seeing the "graph" in GraphQL.

[1] Root resolvers, such as for the `game_by_id` query operation, are the exception: they are passed nil.

An important part of GraphQL is that your query must always extend to scalar fields; if you select a field that is a compound type, such as `BoardGame/designers`, Lacinia will report an error instead:

```
(q "{ game_by_id(id: \"1237\") { name designers }}")
=>
{:errors [{:message "Field `designers' (of type `Designer') must have at least one␣
↪selection.",
           :locations [{:line 1, :column 25}]}]}
```

Notice how the `:data` key is not present here . . . that indicates that the error occured during the parse and prepare phases, before execution in earnest began.

To really demonstrate navigation, we can go from BoardGame to Designer and back:

```
(q "{ game_by_id(id: \"1234\") { name designers { name games { name }}}}")
=> {:data {:game_by_id {:name "Zertz",
                         :designers [{:name "Kris Burm",
                                      :games [{:name "Zertz"}]}]}}}
```

2.6.4 Summary

Lacinia provides the mechanism to create relationships between entities, such as between BoardGame and Designer. It still falls on the field resolvers to provide that data for such linkages.

With that in place, the same com.walmartlabs.lacinia/execute function that gives us data about a single entity can traverse the graph and return data from a variety of entities, organized however you need it.

Next up, we'll take what we have and make it easy to access via HTTP.

2.7 Lacinia Pedestal

Working from the REPL is important, but ultimately GraphQL exists to provide a web-based API. Fortunately, it is very easy to get your Lacinia application up on the web, on top of the Pedestal web tier, using Lacinia-Pedestal.

In addition, for free, we get GraphQL's own REPL: GraphiQL.

2.7.1 Add Dependencies

Listing 12: project.clj

```
(defproject clojure-game-geek "0.1.0-SNAPSHOT"
  :description "A tiny BoardGameGeek clone written in Clojure with Lacinia"
  :url "https://github.com/walmartlabs/clojure-game-geek"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [com.walmartlabs/lacinia-pedestal "0.5.0"]
                 [io.aviso/logging "0.2.0"]])
```

We've added two libraries: `lacinia-pedestal` and `io.aviso/logging`. We no longer need to list `lacinia`, as that is a transitive dependency of `lacinia-pedestal`.[1]

[1] Occasionally, you'll list an explicit lacinia dependency to get a newer version than the one lacinia-pedestal declares. Adding such a dependency directly to your `project.clj` is the correct way to override such a transitive dependency.

The former brings in quite a few dependencies, including Pedestal, and the underlying Jetty layer that Pedestal builds upon.

The `io.aviso/logging` library sets up Logback as the logging library.

Clojure and Java are both rich with web and logging frameworks; Pedestal and Logback are simply particular choices that we've made and prefer; many other people are using Lacinia on the web without using Logback *or* Pedestal.

2.7.2 Some Configuration

For best results, we can configure Logback; this keeps startup and request handling from being very chatty:

Listing 13: dev-resources/logback-test.xml

```
<configuration scan="true" scanPeriod="1 seconds">

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%-5level %logger - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="warn">
    <appender-ref ref="STDOUT"/>
  </root>

</configuration>
```

This configuration hides log events below the warning level (that is, debug and info events). If any warnings or errors do occur, minimal output is sent to the console.

A `logback-test.xml` takes precendence over the production `logback.xml` configuration we will eventually supply.

2.7.3 User Namespace

We'll add more scaffolding to the `user` namespace, to make it possible to start and stop the Pedestal server.

Listing 14: dev-resources/user.clj

```
(ns user
  (:require
    [clojure-game-geek.schema :as s]
    [com.walmartlabs.lacinia :as lacinia]
    [com.walmartlabs.lacinia.pedestal :as lp]
    [io.pedestal.http :as http]
    [clojure.java.browse :refer [browse-url]]
    [clojure.walk :as walk])
  (:import (clojure.lang IPersistentMap)))

(def schema (s/load-schema))

(defn simplify
  "Converts all ordered maps nested within the map into standard hash maps, and
   sequences into vectors, which makes for easier constants in the tests, and␣
↪eliminates ordering problems."
```

(continues on next page)

(continued from previous page)

```
  [m]
  (walk/postwalk
    (fn [node]
      (cond
        (instance? IPersistentMap node)
        (into {} node)

        (seq? node)
        (vec node)

        :else
        node))
    m))

(defn q
  [query-string]
  (-> (lacinia/execute schema query-string nil nil)
      simplify))

(defonce server nil)

(defn start-server
  [_]
  (let [server (-> schema
                   (lp/service-map {:graphiql true})
                   http/create-server
                   http/start)]
    (browse-url "http://localhost:8888/")
    server))

(defn stop-server
  [server]
  (http/stop server)
  nil)

(defn start
  []
  (alter-var-root #'server start-server)
  :started)

(defn stop
  []
  (alter-var-root #'server stop-server)
  :stopped)
```

This new code is almost entirely boilerplate for Pedestal and for Lacinia-Pedestal. The core function is `com.walmartlabs.lacinia.pedestal/service-map` which is passed the compiled schema and a map of options, and returns a Pedestal service map which is then used to create the Pedestal server.

GraphiQL is not enabled by default; it is opt-in, and should generally only be enabled for development servers, or behind a firewall that limits access from the outside world.

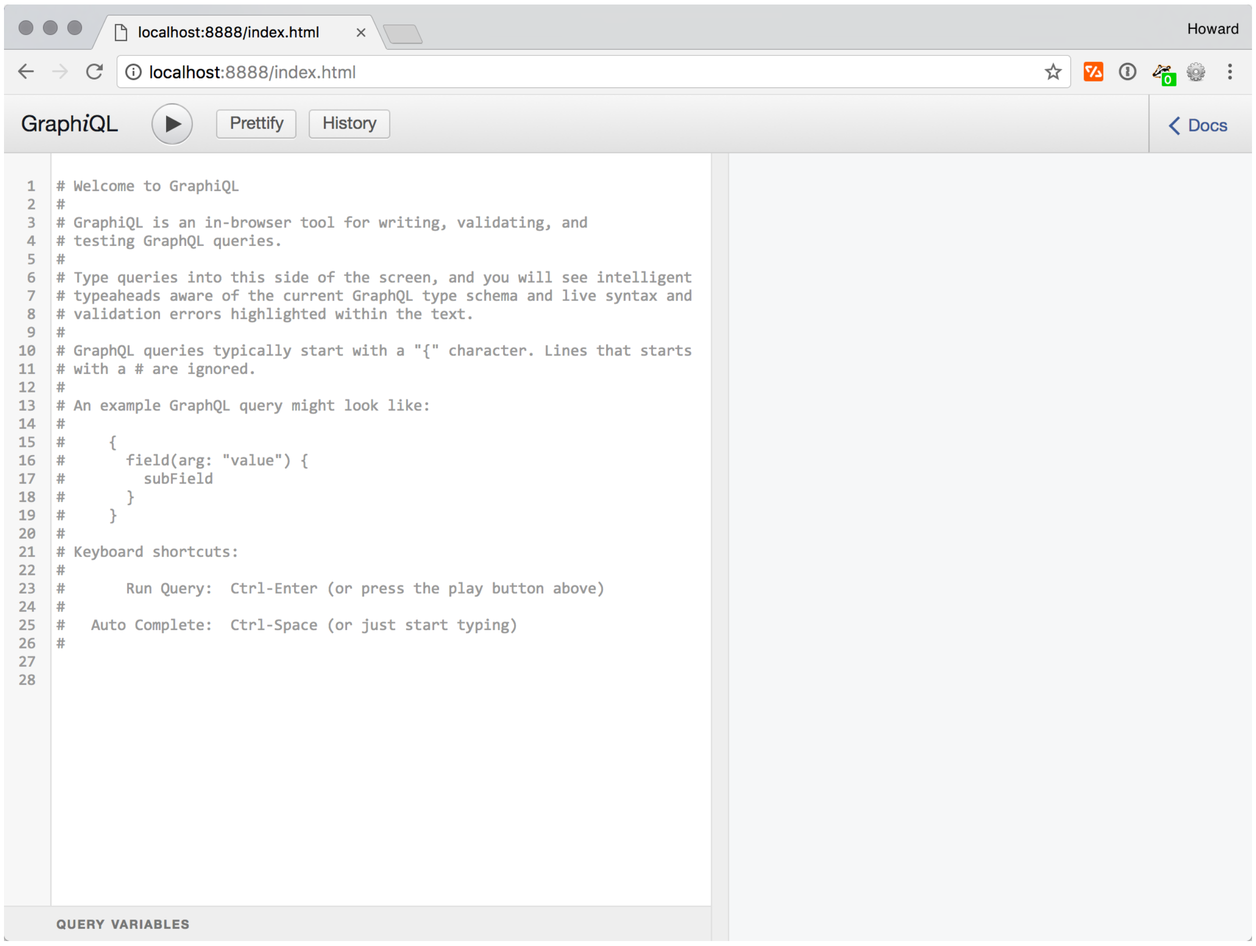
Lacinia-Pedestal services GraphQL requests at the `/graphql` path. The default port is 8888. It handles both GET and POST requests. We'll get to the details later.

The `/` and `/index.html` paths, and related JavaScript and CSS resources, can only be accessed when GraphiQL is enabled.

2.7.4 Starting The Server

With the above scaffolding in place, it is just a matter of starting the REPL and evaluating `(start)`.

At this point, your web browser should open to the GraphiQL application:

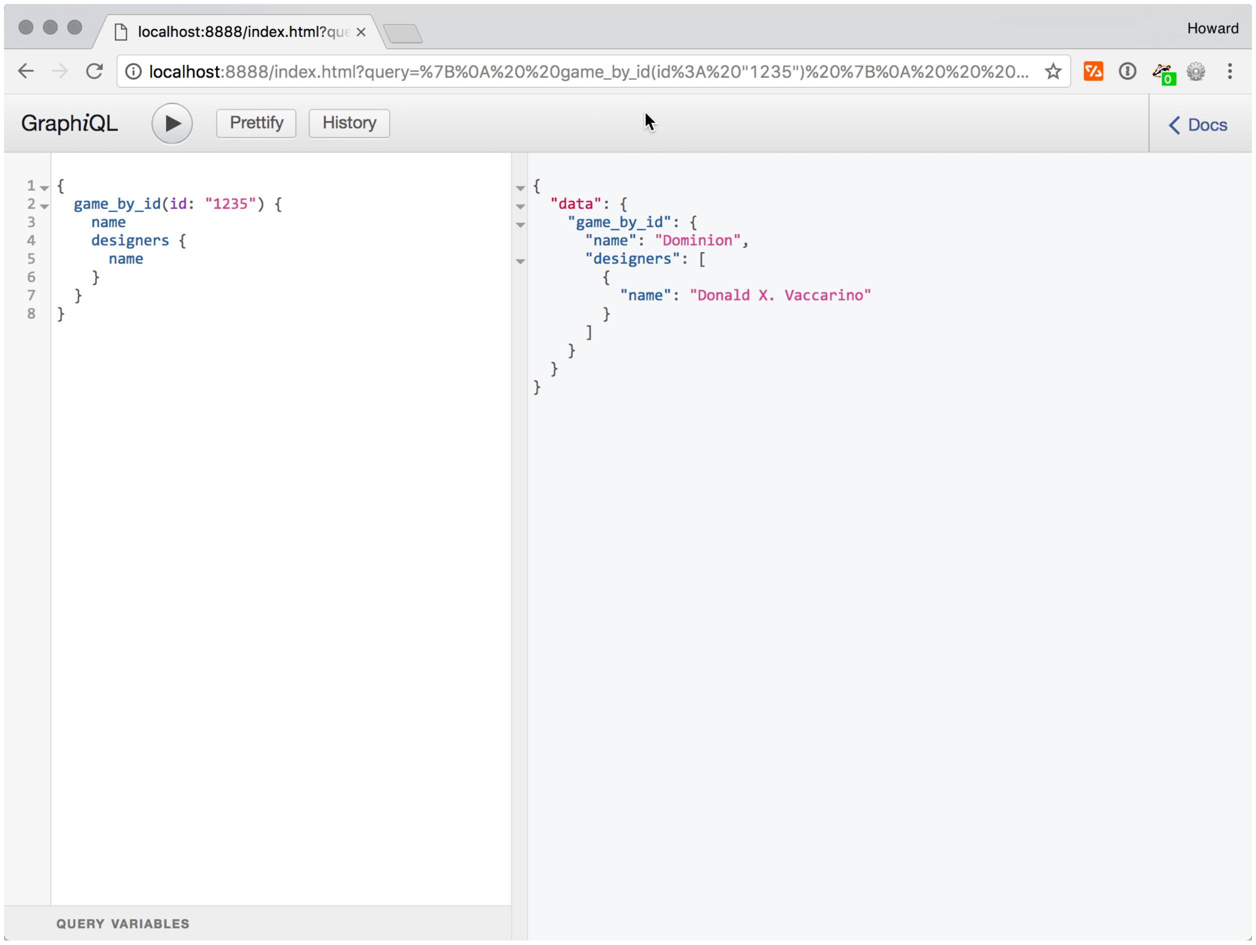


```
# Welcome to GraphiQL
#
# GraphiQL is an in-browser tool for writing, validating, and
# testing GraphQL queries.
#
# Type queries into this side of the screen, and you will see intelligent
# typeaheads aware of the current GraphQL type schema and live syntax and
# validation errors highlighted within the text.
#
# GraphQL queries typically start with a "{" character. Lines that starts
# with a # are ignored.
#
# An example GraphQL query might look like:
#
#     {
#       field(arg: "value") {
#         subField
#       }
#     }
#
# Keyboard shortcuts:
#
#       Run Query:  Ctrl-Enter (or press the play button above)
#
#   Auto Complete:  Ctrl-Space (or just start typing)
#
```

Tip: It's really worth following along with this section, especially if you haven't played with GraphiQL before. GraphiQL assists you with formatting, provides pop-up help, flags errors in your query, and supplies automatic input completion. It can even pretty print your query. It makes for quite the demo!

2.7.5 Running Queries

We can now type a query into the large text area on the left and then click the right arrow button (or type `Command+Enter`), and see the server response as pretty-printed JSON on the right:


localhost:8888/index.html?que
Howard
localhost:8888/index.html?query=%7B%0A%20%20game_by_id(id%3A%20"1235")%20%7B%0A%20%20%20...
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

```
{
  game_by_id(id: "1235") {
    name
    designers {
      name
    }
  }
}
```

```
{
  "data": {
    "game_by_id": {
      "name": "Dominion",
      "designers": [
        {
          "name": "Donald X. Vaccarino"
        }
      ]
    }
  }
}
```

Notice that the URL bar in the browser has updated: it contains the full query string. This means that you can bookmark a query you like for later (though it's easier to do that using the `History` button). Alternately, and more importantly, you can copy that URL and provide it to other developers. They can start up the application on their workstations and see exactly what you see, a real boon for describing and diagnosing problems.

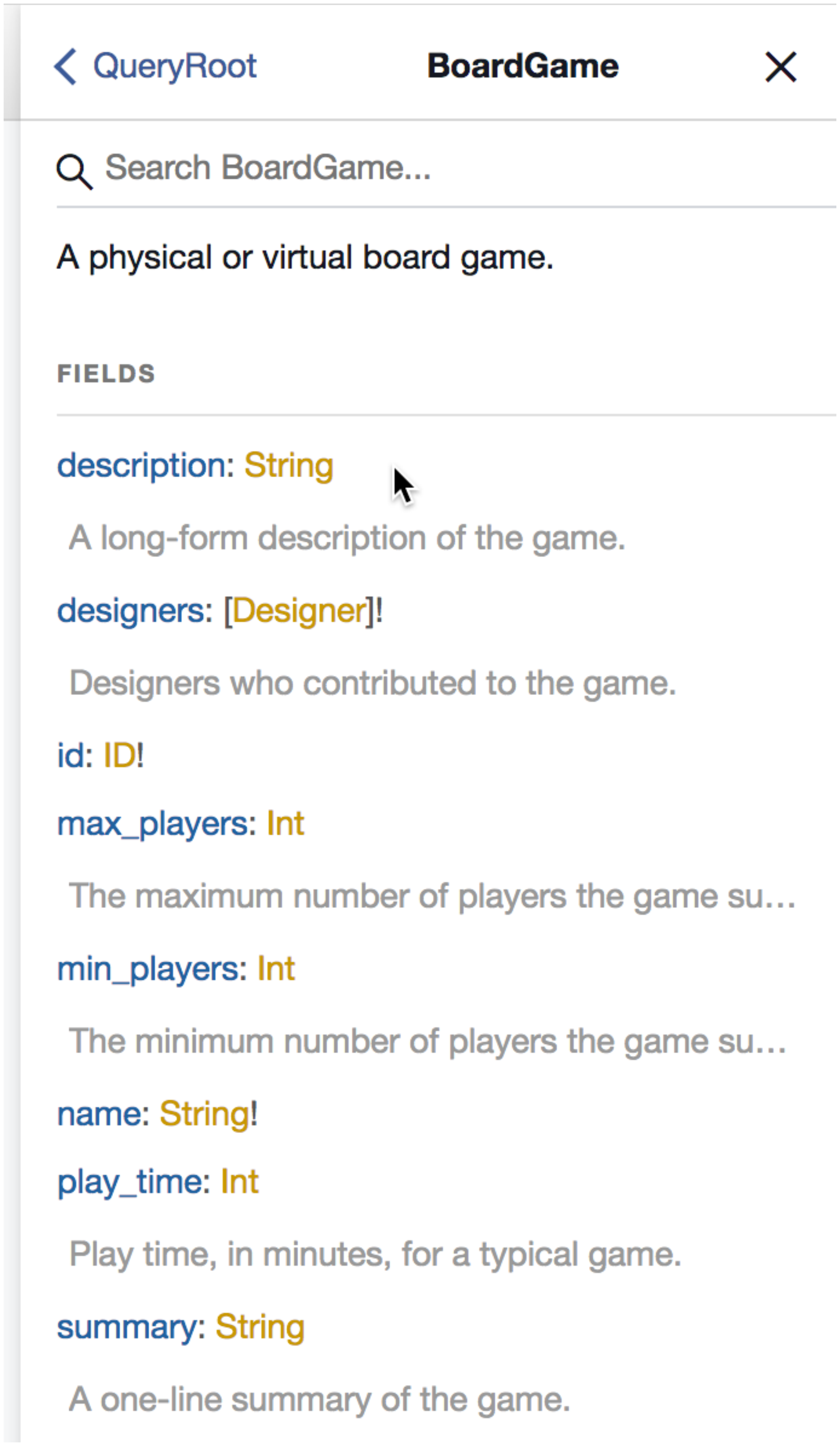
This approach works even better when you keep a GraphQL server running on a shared staging server. On split[2] teams, the developers creating the application can easily explore the interface exposed by the GraphQL server, even before writing their first line of code.

Trust me, they love that.

2.7.6 Documentation Browser

The `< Docs` button on the right opens the documentation browser:

[2] That is, where one team or set of developers *just* does the user interface, and the other team *just* does the server side (including Lacinia). Part of the value proposition for GraphQL is how clean and uniform this split can be.


QueryRoot BoardGame

Search BoardGame...

A physical or virtual board game.

FIELDS

description: String

A long-form description of the game.

designers: [Designer]!

Designers who contributed to the game.

id: ID!

max_players: Int

The maximum number of players the game su...

min_players: Int

The minimum number of players the game su...

name: String!

play_time: Int

Play time, in minutes, for a typical game.

summary: String

A one-line summary of the game.

The documentation browser is invaluable: it allows you to navigate around your schema, drilling down to queries, objects, and fields to see a summary of each declaration, as well as documentation - those `:documentation` values we added way back *at the beginning*.

Take some time to learn what GraphiQL can do for you.

2.8 Refactoring to Components

Before we add the next bit of functionality to our application, it's time to take a small detour, into the use of Stuart Sierra's component library.[1]

As Clojure programs grow, the namespaces, and relationships between those namespaces, grow in number and complexity. In our *previous example*, we saw that the logic to start the Jetty instance was strewn across the `user` namespace.

This isn't a problem in our toy application, but as a real application grows, we'd start to see some issues and concerns:

- A single 'startup' namespace (maybe with a `-main` method) imports every single other namespace.
- Potential for duplication or conflict between the *real* startup code and the *test* startup code.[2]
- Is there a good way to *stop* things, say, between tests?
- Is there a way to mock parts of the system (for testing purposes)?
- We really want to avoid a proliferation of global variables. Ideally, none!

Component is a simple, no-nonsense way to achieve the above goals. It gives you a clear way to organize your code, and it does things in a fully *functional* way: no globals, no update-in-place, and easy to reason about.

The building-block of Component is, unsurprisingly, components. These components are simply ordinary Clojure maps – though for reasons we'll discuss shortly, Clojure record types are more typically used.

The components are formed into a system, which again is just a map. Each component has a unique, well-known key in the system map.

Components *may* have dependencies on other components. That's where the fun really starts.

Components *may* have a lifecycle; if they do, they implement the Lifecycle protocol containing methods `start` and `stop`. This is why many components are implemented as Clojure records . . . records can implement a protocol, but simple maps can't.

Rather than get into the minutiae, let's see how it all fits together in our Clojure Game Geek application.

2.8.1 Add Dependencies

Listing 15: project.clj

```
(defproject clojure-game-geek "0.1.0-SNAPSHOT"
  :description "A tiny BoardGameGeek clone written in Clojure with Lacinia"
  :url "https://github.com/walmartlabs/clojure-game-geek"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [com.stuartsierra/component "0.3.2"]
                 [com.walmartlabs/lacinia-pedestal "0.5.0"]
                 [io.aviso/logging "0.2.0"]])
```

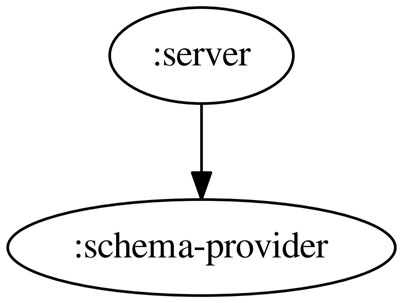
We've added the `component` library.

[1] Stuart provides a really good explanation of Component in his Clojure/West 2014 talk.

[2] We've been sloppy so far, in that we haven't even thought about testing. That will change shortly.

2.8.2 System Map

We're starting quite small, with just two components in our system:



The `:server` component is responsible for setting up the Pedestal service, which requires a compiled Lacinia schema. The `:schema-provider` component exposes that schema as its `:schema` key.

Later, we'll be adding additional components for other logic, such as database connections, thread pools, authentication/authorization checks, caching, and so forth. But it's easier to start small.

What does it mean for one service to depend on another? Dependencies are acted upon when the system is started (and again when the system is stopped).

The dependencies influence the order in which each component is started. Here, `:schema-provider` is started before `:server`, as `:server` depends on `:schema-provider`.

Secondly, the *started* version of a dependency is `assoc`-ed into the dependening component. After `:schema-provider` starts, the started version of the component will be `assoc`-ed as the `:schema-provider` key of the `:server` component.

Once a component has its dependencies `assoc`-ed in, and is itself started (more on that in a moment), it may be `assoc`-ed into further components.

The Component library embraces the identity vs. state concept; the identity of the component is its key in the system map . . . its state is a series of transformations of the initial map.

2.8.3 :schema-provider component

The `clojure-game-geek.schema` namespace has been extended to provide the `:schema-provider` component.

Listing 16: src/clojure_game_geek/schema.clj

```
(ns clojure-game-geek.schema
  "Contains custom resolvers and a function to provide the full schema."
  (:require
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.util :as util]
    [com.walmartlabs.lacinia.schema :as schema]
    [com.stuartsierra.component :as component]
```

(continues on next page)

(continued from previous page)

```
    [clojure.edn :as edn]))

(defn resolve-game-by-id
  [games-map context args value]
  (let [{:keys [id]} args]
    (get games-map id)))

(defn resolve-board-game-designers
  [designers-map context args board-game]
  (->> board-game
       :designers
       (map designers-map)))

(defn resolve-designer-games
  [games-map context args designer]
  (let [{:keys [id]} designer]
    (->> games-map
         vals
         (filter #(-> % :designers (contains? id))))))

(defn entity-map
  [data k]
  (reduce #(assoc %1 (:id %2) %2)
          {}
          (get data k)))

(defn resolver-map
  [component]
  (let [cgg-data (-> (io/resource "cgg-data.edn")
                     slurp
                     edn/read-string)
        games-map (entity-map cgg-data :games)
        designers-map (entity-map cgg-data :designers)]
    {:query/game-by-id (partial resolve-game-by-id games-map)
     :BoardGame/designers (partial resolve-board-game-designers designers-map)
     :Designer/games (partial resolve-designer-games games-map)}))

(defn load-schema
  [component]
  (-> (io/resource "cgg-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers (resolver-map component))
      schema/compile))

(defrecord SchemaProvider [schema]

  component/Lifecycle

  (start [this]
    (assoc this :schema (load-schema this)))

  (stop [this]
    (assoc this :schema nil)))

(defn new-schema-provider
  []
```

(continues on next page)

(continued from previous page)

```
  {:schema-provider (map->SchemaProvider {})})
```

The real changes are at the bottom of the namespace. There's a new record, SchemaProvider, that implements the Lifecycle protocol.

Lifecycle is optional; trivial components may not need it. In our case, we use the `start` method as an opportunity to load and compile the Lacinia schema.

Notice that we are passing the component into `load-schema`. This isn't necessary yet, but in later iterations of the Clojure Game Geek application, the `:schema-provider` component will have dependencies on other components, generally because a field resolver will need access to the component.

When you implement a protocol, you must implement all the methods of the protocol. In Component's Lifecycle protocol, you typically will undo in `stop` whatever you did in `start`. For example, a Component that manages a database connection will open it in `start` and close it in `stop`.

Here we just get rid of the compiled schema,[3] but it is also common and acceptable for a `stop` method to just return `this` if the component doesn't have external resources, such as a database connection, to manage.

Finally, the `new-schema-provider` function is a constructor around the SchemaProvider record. It returns a single-element map, associating the `:schema-provider` system key for the component with the initial iteration of the component itself.[4]

2.8.4 :server component

Likewise, the `clojure-game-geek.server` namespace now provides the `:server` component.

Listing 17: src/clojure_game_geek/server.clj

```
(ns clojure-game-geek.server
  (:require [com.stuartsierra.component :as component]
            [com.walmartlabs.lacinia.pedestal :as lp]
            [io.pedestal.http :as http]))

(defrecord Server [schema-provider server]

  component/Lifecycle
  (start [this]
    (assoc this :server (-> schema-provider
                            :schema
                            (lp/service-map {:graphiql true})
                            http/create-server
                            http/start)))

  (stop [this]
    (http/stop server)
    (assoc this :server nil)))

(defn new-server
  []
  {:server (component/using (map->Server {})
                            [:schema-provider])})
```

[3] You might be tempted to use a `dissoc` here, but if you `dissoc` a declared key of a record, the result is an ordinary map, which can break tests that rely on repeatedly starting and stopping the system.

[4] This is just one approach; another would be to provide a function that `assoc`-ed the component into the system map.

Much of the code previously in the `user` namespace has moved here.

You can see how the components work together, inside the `start` method. The Component library has `assoc`-ed the `:schema-provider` component into the `:server` component, so it's possible to get the `:schema` key and build the Pedestal server from it.

`start` and `stop` methods often have side-effects. This is explicit here, with the call to `http/stop` before clearing the `:server` key.

The `new-server` function not only gives the component its system key and initial state, but also invokes `component/using` to establish the dependency on the `:schema-provider` component.

2.8.5 system namespace

A new, tiny namespace has been created, just to put together the Component system map.

Listing 18: src/clojure_game_geek/system.clj

```
(ns clojure-game-geek.system
  (:require
    [com.stuartsierra.component :as component]
    [clojure-game-geek.schema :as schema]
    [clojure-game-geek.server :as server]))

(defn new-system
  []
  (merge (component/system-map)
         (server/new-server)
         (schema/new-schema-provider)))
```

You can imagine that, as the system grows larger, so will this namespace. But at the same time, individual components will only need to know about the components they directly depend upon.

2.8.6 user namespace

Listing 19: dev-resources/user.clj

```
(ns user
  (:require
    [com.walmartlabs.lacinia :as lacinia]
    [clojure.java.browse :refer [browse-url]]
    [clojure-game-geek.system :as system]
    [clojure.walk :as walk]
    [com.stuartsierra.component :as component])
  (:import (clojure.lang IPersistentMap)))

(defn simplify
  "Converts all ordered maps nested within the map into standard hash maps, and
   sequences into vectors, which makes for easier constants in the tests, and␣
↪eliminates ordering problems."
  [m]
  (walk/postwalk
    (fn [node]
      (cond
        (instance? IPersistentMap node)
        (into {} node)
```

(continues on next page)

(continued from previous page)

```
        (seq? node)
        (vec node)

        :else
        node))
    m))

(defonce system (system/new-system))

(defn q
  [query-string]
  (-> system
      :schema-provider
      :schema
      (lacinia/execute query-string nil nil)
      simplify))

(defn start
  []
  (alter-var-root #'system component/start-system)
  (browse-url "http://localhost:8888/")
  :started)

(defn stop
  []
  (alter-var-root #'system component/stop-system)
  :stopped)
```

The user namespace has shrunk; previously it was responsible for loading the schema, and creating and starting the Pedestal service; this has all shifted to the individual components.

Instead, the user namespace creates a system map, and can use `start-system` and `stop-system` on that system map: no direct knowledge of loading schemas or starting and stopping Pedestal is present any longer.

The user namespace previously had vars for both the schema and the Pedestal service. Now it only has a single var, for the Component system map.

Interestingly, as our system grows later, the user namespace will likely not change at all, just the system map it gets from `system/new-system` will expand.

The only wrinkle here is in the `q` function; since there's no longer a local `schema` var it is necessary to pull the `:schema-provider` component from the system map, and extract the schema from that component.

2.9 Adding Members and Ratings

We're now starting an arc towards adding our first mutations.

We're going to extend our schema to add Members (the name for a user of the Clojure Game Geek web site), and GameRatings . . . how a member has rated a game, on a scale of one to five.

Each Member can rate any BoardGame, but can only rate any single game once.

2.9.1 Schema Changes

First, let's add new fields, types, and queries to support these new features:

Listing 20: resources/cgg-schema.edn

```
{:objects
 {:BoardGame
  {:description "A physical or virtual board game."
   :fields
   {:id {:type (non-null ID)}
    :name {:type (non-null String)}
    :rating_summary {:type (non-null :GameRatingSummary)
                     :resolve :BoardGame/rating-summary}
    :summary {:type String
              :description "A one-line summary of the game."}
    :description {:type String
                  :description "A long-form description of the game."}
    :designers {:type (non-null (list :Designer))
                :description "Designers who contributed to the game."
                :resolve :BoardGame/designers}
    :min_players {:type Int
                  :description "The minimum number of players the game supports."}
    :max_players {:type Int
                  :description "The maximum number of players the game supports."}
    :play_time {:type Int
                :description "Play time, in minutes, for a typical game."}}}

  :GameRatingSummary
  {:description "Summary of ratings for a single game."
   :fields
   {:count {:type (non-null Int)
            :description "Number of ratings provided for the game.  Ratings are 1 to␣
↪5 stars."}
    :average {:type (non-null Float)
              :description "The average value of all ratings, or 0 if never rated."}}}

  :Member
  {:description "A member of Clojure Game Geek.  Members can rate games."
   :fields
   {:id {:type (non-null ID)}
    :member_name {:type (non-null String)
                  :description "Unique name of member."}
    :ratings {:type (list :GameRating)
              :description "List of games and ratings provided by this member."
              :resolve :Member/ratings}}}

  :GameRating
  {:description "A member's rating of a particular game."
   :fields
   {:game {:type (non-null :BoardGame)
           :description "The Game rated by the member."
           :resolve :GameRating/game}
    :rating {:type (non-null Int)
             :description "The rating as 1 to 5 stars."}}}

  :Designer
  {:description "A person who may have contributed to a board game design."
   :fields
   {:id {:type (non-null ID)}
    :name {:type (non-null String)}
```

(continues on next page)

(continued from previous page)

```
    :url {:type String
          :description "Home page URL, if known."}
    :games {:type (non-null (list :BoardGame))
            :description "Games designed by this designer."
            :resolve :Designer/games}}}}

 :queries
 {:game_by_id
  {:type :BoardGame
   :description "Select a BoardGame by its unique id, if it exists."
   :args
   {:id {:type (non-null ID)}}
   :resolve :query/game-by-id}

  :member_by_id
  {:type :Member
   :description "Select a ClojureGameGeek Member by their unique id, if it exists."
   :args
   {:id {:type (non-null ID)}}
   :resolve :query/member-by-id}}}
```

For a particular BoardGame, you can get just a simple summary of the ratings: the total number, and a simple average.

We've added a new top-level entity, Member. From a Member, you can get a detailed list of all the game's that member has rated.

2.9.2 Data Changes

We'll model these ratings in our test data, much as we would a many-to-many relationship within a SQL database:

Listing 21: dev-resources/cgg-data.edn

```
{:games
 [{:id "1234"
   :name "Zertz"
   :summary "Two player abstract with forced moves and shrinking board"
   :min_players 2
   :max_players 2
   :designers #{"200"}}
  {:id "1235"
   :name "Dominion"
   :summary "Created the deck-building genre; zillions of expansions"
   :designers #{"204"}
   :min_players 2}
  {:id "1236"
   :name "Tiny Epic Galaxies"
   :summary "Fast dice-based sci-fi space game with a bit of chaos"
   :designers #{"203"}
   :min_players 1
   :max_players 4}
  {:id "1237"
   :name "7 Wonders: Duel"
   :summary "Tense, quick card game of developing civilizations"
   :designers #{"201" "202"}
   :min_players 2
   :max_players 2}]
```

(continues on next page)

(continued from previous page)

```
:members
[{:id "37"
  :member_name "curiousattemptbunny"}
 {:id "1410"
  :member_name "bleedingedge"}
 {:id "2812"
  :member_name "missyo"}]

:ratings
[{:member_id "37" :game_id "1234" :rating 3}
 {:member_id "1410" :game_id "1234" :rating 5}
 {:member_id "1410" :game_id "1236" :rating 4}
 {:member_id "1410" :game_id "1237" :rating 4}
 {:member_id "2812" :game_id "1237" :rating 4}
 {:member_id "37" :game_id "1237" :rating 5}]

:designers
[{:id "200"
  :name "Kris Burm"
  :url "http://www.gipf.com/project_gipf/burm/burm.html"}
 {:id "201"
  :name "Antoine Bauza"
  :url "http://www.antoinebauza.fr/"}
 {:id "202"
  :name "Bruno Cathala"
  :url "http://www.brunocathala.com/"}
 {:id "203"
  :name "Scott Almes"}
 {:id "204"
  :name "Donald X. Vaccarino"}]}
```

2.9.3 New Resolvers

Our schema changes introduced a few new field resolvers, which we must implement:

Listing 22: src/clojure_game_geek/schema.clj

```
(ns clojure-game-geek.schema
  "Contains custom resolvers and a function to provide the full schema."
  (:require
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.util :as util]
    [com.walmartlabs.lacinia.schema :as schema]
    [com.stuartsierra.component :as component]
    [clojure.edn :as edn]))

(defn resolve-element-by-id
  [element-map context args value]
  (let [{:keys [id]} args]
    (get element-map id)))

(defn resolve-board-game-designers
  [designers-map context args board-game]
  (->> board-game
```

(continues on next page)

(continued from previous page)

```
       :designers
       (map designers-map)))

(defn resolve-designer-games
  [games-map context args designer]
  (let [{:keys [id]} designer]
    (->> games-map
         vals
         (filter #(-> % :designers (contains? id))))))

(defn entity-map
  [data k]
  (reduce #(assoc %1 (:id %2) %2)
          {}
          (get data k)))

(defn rating-summary
  [cgg-data]
  (fn [_ _ board-game]
    (let [id (:id board-game)
          ratings (->> cgg-data
                       :ratings
                       (filter #(= id (:game_id %)))
                       (map :rating))
          n (count ratings)]
      {:count n
       :average (if (zero? n)
                  0
                  (/ (apply + ratings)
                     (float n)))})))

(defn member-ratings
  [ratings-map]
  (fn [_ _ member]
    (let [id (:id member)]
      (filter #(= id (:member_id %)) ratings-map))))

(defn game-rating->game
  [games-map]
  (fn [_ _ game-rating]
    (get games-map (:game_id game-rating))))

(defn resolver-map
  [component]
  (let [cgg-data (-> (io/resource "cgg-data.edn")
                     slurp
                     edn/read-string)
        games-map (entity-map cgg-data :games)
        members-map (entity-map cgg-data :members)
        designers-map (entity-map cgg-data :designers)]
    {:query/game-by-id (partial resolve-element-by-id games-map)
     :query/member-by-id (partial resolve-element-by-id members-map)
     :BoardGame/designers (partial resolve-board-game-designers designers-map)
     :BoardGame/rating-summary (rating-summary cgg-data)
     :GameRating/game (game-rating->game games-map)
     :Designer/games (partial resolve-designer-games games-map)
     :Member/ratings (member-ratings (:ratings cgg-data))}))
```

(continues on next page)

(continued from previous page)

```
(defn load-schema
  [component]
  (-> (io/resource "cgg-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers (resolver-map component))
      schema/compile))

(defrecord SchemaProvider [schema]

  component/Lifecycle

  (start [this]
    (assoc this :schema (load-schema this)))

  (stop [this]
    (assoc this :schema nil)))

(defn new-schema-provider
  []
  {:schema-provider (map->SchemaProvider {})})
```

We've generalized `resolve-game-by-id` into `resolve-element-by-id`.

We've introduced three new resolvers, `rating-summary`, `member-ratings`, and `game-rating->game`.

These new resolvers are implemented using a new pattern. The existing resolvers, such as `resolve-designer-games`, took an initial parameter (a slice of the in-memory database), plus the standard triumvirate of context, field arguments, and container value. This approach is concise, but requires the use of `partial` (to supply that initial parameter) when building the resolvers map.

These new resolvers use a factory pattern instead: the extra value (the database map) is the only parameter, which is captured in a closure; a stand-alone field resolver function, one that accepts exactly the standard triumvirate, is returned. No use of `partial` needed anymore.

Further, this new pattern is closer to what we'll end up with in a later tutorial chapter, when we see how to use a Component as a field resolver.

It's worth emphasising again that field resolvers don't just access data, they can transform it. The `rating-summary` field resolver is an example of that; there's no database entity directly corresponding to the schema type `:GameRatingSummary`, but the field resolver can build that information directly. There doesn't even have to be a special type or record . . . just a standard Clojure map with the correctly named keys.

2.9.4 Testing it Out

Back at the REPL, we can test out the new functionality. First, select the rating summary data for a game:

```
(q "{ game_by_id(id: \"1237\") { name rating_summary { count average }}}")
=>
{:data {:game_by_id {:name "7 Wonders: Duel",
        :rating_summary {:count 3,
                         :average 4.333333333333333}}}}
```

We can also lookup a member, and find all the games they've rated:

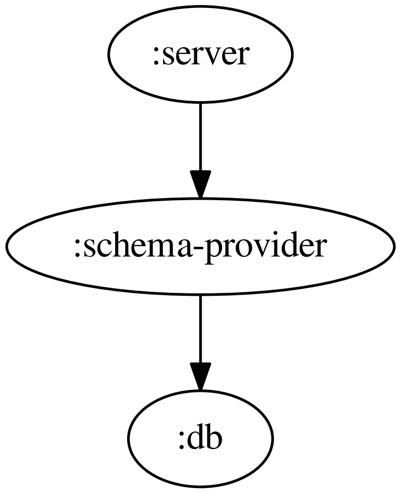
```
(q "{ member_by_id(id: \"1410\") { member_name ratings { game { name } rating }}}")
=>
{:data {:member_by_id {:member_name "bleedingedge",
                       :ratings [{:game {:name "Zertz"}, :rating 5}
                                 {:game {:name "Tiny Epic Galaxies"}, :rating 4}
                                 {:game {:name "7 Wonders: Duel"}, :rating 4}]}}}
```

2.10 Mutable Database

We're still not quite ready to implement our mutation . . . because we're storing our data in an immutable map. Once again, we're not going to take on running an external database, instead we'll put our immutable map inside an Atom. We'll also do some refactoring that will make our eventual transition to an external database that much easier.

2.10.1 System Map

In the previous versions of the application, the database data was an immutable map, and all the logic for traversing that map was inside the `clojure-game-geek.schema` namespace. With this change, we're breaking things apart, there'll be a new namespace, and new component, to encapsulate the database itself.


$$\begin{tikzcd}
\text{:server} \arrow[d] \\
\text{:schema-provider} \arrow[d] \\
\text{:db}
\end{tikzcd}$$

2.10.2 db namespace

Listing 23: src/clojure_game_geek/db.clj

```
(ns clojure-game-geek.db
  (:require
    [clojure.edn :as edn]
    [clojure.java.io :as io]
```

(continues on next page)

(continued from previous page)

```
    [com.stuartsierra.component :as component]))

(defrecord ClojureGameGeekDb [data]

  component/Lifecycle

  (start [this]
    (assoc this :data (-> (io/resource "cgg-data.edn")
                          slurp
                          edn/read-string
                          atom)))

  (stop [this]
    (assoc this :data nil)))

(defn new-db
  []
  {:db (map->ClojureGameGeekDb {})})

(defn find-game-by-id
  [db game-id]
  (->> db
       :data
       deref
       :games
       (filter #(= game-id (:id %)))
       first))

(defn find-member-by-id
  [db member-id]
  (->> db
       :data
       deref
       :members
       (filter #(= member-id (:id %)))
       first))

(defn list-designers-for-game
  [db game-id]
  (let [designers (:designers (find-game-by-id db game-id))]
    (->> db
         :data
         deref
         :designers
         (filter #(contains? designers (:id %))))))

(defn list-games-for-designer
  [db designer-id]
  (->> db
       :data
       deref
       :games
       (filter #(-> % :designers (contains? designer-id)))))

(defn list-ratings-for-game
  [db game-id]
  (->> db
```

(continues on next page)

(continued from previous page)

```
       :data
       deref
       :ratings
       (filter #(= game-id (:game_id %)))))

(defn list-ratings-for-member
  [db member-id]
  (->> db
       :data
       deref
       :ratings
       (filter #(= member-id (:member_id %)))))
```

This namespace does two things:

- Defines a component in terms of a record and a constructor function
- Provides an API for database access focused upon that component

At this point, the Component is nothing more than a home for the `:data` Atom. That Atom is created and initialized inside the `start` lifecycle method.

All of those data access functions follow.

This code employs a few reasonable conventions:

- `find-` prefix for functions that get data by primary key, and may return nil if not found
- `list-` prefix is like `find-`, but returns a seq of matches
- The `:db` component is always the first parameter, as `db`

Later, when we add some mutations, we'll define further functions and new naming and coding conventions.

The common trait for all of these is the `(-> db :data deref ...)` code; in other words, reach into the component, access the `:data` property (the Atom) and deref the Atom to get the immutable map.

Looking forward to when we do have an external database . . . these functions will change, but their signatures will not. Any code that invokes these functions, for example the field resolver functions defined in `clojure-game-geek.schema`, will work, unchanged, after we swap in the external database implementation.

2.10.3 system namespace

Listing 24: src/clojure_game_geek/system.clj

```
(ns clojure-game-geek.system
  (:require
    [com.stuartsierra.component :as component]
    [clojure-game-geek.schema :as schema]
    [clojure-game-geek.server :as server]
    [clojure-game-geek.db :as db]))

(defn new-system
  []
  (merge (component/system-map)
         (server/new-server)
         (schema/new-schema-provider)
         (db/new-db)))
```

The :db component doesn't effectively exist until it is part of the system map. This change adds it in. As promised previously, namespaces that use the system (such as the user namespace) don't change at all.

2.10.4 schema namespace

The schema namespace has shrunk, and improved:

Listing 25: src/clojure_game_geek/schema.clj

```
(ns clojure-game-geek.schema
  "Contains custom resolvers and a function to provide the full schema."
  (:require
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.util :as util]
    [com.walmartlabs.lacinia.schema :as schema]
    [com.stuartsierra.component :as component]
    [clojure-game-geek.db :as db]
    [clojure.edn :as edn]))

(defn game-by-id
  [db]
  (fn [_ args _]
    (db/find-game-by-id db (:id args))))

(defn member-by-id
  [db]
  (fn [_ args _]
    (db/find-member-by-id db (:id args))))

(defn board-game-designers
  [db]
  (fn [_ _ board-game]
    (db/list-designers-for-game db (:id board-game))))

(defn designer-games
  [db]
  (fn [_ _ designer]
    (db/list-games-for-designer db (:id designer))))

(defn rating-summary
  [db]
  (fn [_ _ board-game]
    (let [ratings (map :rating (db/list-ratings-for-game db (:id board-game)))
          n (count ratings)]
      {:count n
       :average (if (zero? n)
                  0
                  (/ (apply + ratings)
                     (float n)))})))

(defn member-ratings
  [db]
  (fn [_ _ member]
    (db/list-ratings-for-member db (:id member))))

(defn game-rating->game
  [db]
```

(continues on next page)

(continued from previous page)

```
  (fn [_ _ game-rating]
    (db/find-game-by-id db (:game_id game-rating))))

(defn resolver-map
  [component]
  (let [db (:db component)]
    {:query/game-by-id (game-by-id db)
     :query/member-by-id (member-by-id db)
     :BoardGame/designers (board-game-designers db)
     :BoardGame/rating-summary (rating-summary db)
     :GameRating/game (game-rating->game db)
     :Designer/games (designer-games db)
     :Member/ratings (member-ratings db)}))

(defn load-schema
  [component]
  (-> (io/resource "cgg-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers (resolver-map component))
      schema/compile))

(defrecord SchemaProvider [schema]

  component/Lifecycle

  (start [this]
    (assoc this :schema (load-schema this)))

  (stop [this]
    (assoc this :schema nil)))

(defn new-schema-provider
  []
  {:schema-provider (-> {}
                        map->SchemaProvider
                        (component/using [:db]))})
```

Now all of the resolver functions are following the factory style, but they're largely just wrappers around the functions from the `clojure-game-geek.db` namespace.

And we still don't have any tests (the shame!), but we can exercise a lot of the system from the REPL:

```
(q "{ member_by_id(id: \"1410\") { member_name ratings { game { name rating_summary {␣
↪count average } designers { name  games { name }}} rating }}}")
=>
{:data {:member_by_id {:member_name "bleedingedge",
                       :ratings [{:game {:name "Zertz",
                                         :rating_summary {:count 2, :average 4.0},
                                         :designers [{:name "Kris Burm", :games [
↪{:name "Zertz"}]}]},
                                  :rating 5}
                                 {:game {:name "Tiny Epic Galaxies",
                                         :rating_summary {:count 1, :average 4.0},
                                         :designers [{:name "Scott Almes", :games [
↪{:name "Tiny Epic Galaxies"}]}]},
                                  :rating 4}
```

(continues on next page)

(continued from previous page)

```
                                {:game {:name "7 Wonders: Duel",
                                        :rating_summary {:count 3, :average 4.
↪333333333333333},
                                        :designers [{:name "Antoine Bauza", :games [
↪{:name "7 Wonders: Duel"}]}
                                                    {:name "Bruno Cathala", :games [
↪{:name "7 Wonders: Duel"}]}]},
                                 :rating 4}]}}}
```

2.11 Game Rating Mutation

We're finally ready to add our first mutation, which will be used to add a GameRating.

Our goal is a mutation which allows a member of ClojureGameGeek to apply a rating to a game. We must cover two cases: one where the member is adding an entirely new rating, and one where the member is revising a prior rating.

Along the way, we'll also start to see how to handle errors, which tend to be more common when implementing mutations than with queries.

It is implicit that queries are idempotent (can be repeated getting the same results, and don't change server-side state), whereas mutations are expected to make changes to server-side state as a side-effect. However, that side-effect is essentially invisible to Lacinia, as it will occur during the execution of a field resolver function.

The difference between a query and a mutation in GraphQL is razor thin. When the incoming query document contains only a single top level operation, as is the case in all the examples so far in this tutorial, then there is no difference at all between them.

When the query document contains multiple mutations, then the top-level mutations execute sequentially; the first completes before the second begins execution. For queries, execution order is not in a specified order (though the order of keys and values is specified by the client query).

We'll consider the changes here back-to-front, starting with our database (which is still just a map inside an Atom).

2.11.1 Database Layer Changes

Listing 26: src/clojure_game_geek/db.clj

```
(ns clojure-game-geek.db
  (:require
    [clojure.edn :as edn]
    [clojure.java.io :as io]
    [com.stuartsierra.component :as component]))

(defrecord ClojureGameGeekDb [data]

  component/Lifecycle

  (start [this]
    (assoc this :data (-> (io/resource "cgg-data.edn")
                          slurp
                          edn/read-string
                          atom)))

  (stop [this]
```

(continues on next page)

(continued from previous page)

```
    (assoc this :data nil)))

(defn new-db
  []
  {:db (map->ClojureGameGeekDb {})})

(defn find-game-by-id
  [db game-id]
  (->> db
       :data
       deref
       :games
       (filter #(= game-id (:id %)))
       first))

(defn find-member-by-id
  [db member-id]
  (->> db
       :data
       deref
       :members
       (filter #(= member-id (:id %)))
       first))

(defn list-designers-for-game
  [db game-id]
  (let [designers (:designers (find-game-by-id db game-id))]
    (->> db
         :data
         deref
         :designers
         (filter #(contains? designers (:id %))))))

(defn list-games-for-designer
  [db designer-id]
  (->> db
       :data
       deref
       :games
       (filter #(-> % :designers (contains? designer-id)))))

(defn list-ratings-for-game
  [db game-id]
  (->> db
       :data
       deref
       :ratings
       (filter #(= game-id (:game_id %)))))

(defn list-ratings-for-member
  [db member-id]
  (->> db
       :data
       deref
       :ratings
       (filter #(= member-id (:member_id %)))))
```

(continues on next page)

(continued from previous page)

```
(defn ^:private apply-game-rating
  [game-ratings game-id member-id rating]
  (->> game-ratings
       (remove #(and (= game-id (:game_id %))
                     (= member-id (:member_id %))))
       (cons {:game_id game-id
              :member_id member-id
              :rating rating})))

(defn upsert-game-rating
  "Adds a new game rating, or changes the value of an existing game rating."
  [db game-id member-id rating]
  (-> db
      :data
      (swap! update :ratings apply-game-rating game-id member-id rating)))
```

What's an upsert?

Simply put, an upsert will be either an insert (if the row is new) or an update (if the row exists). This terminology is used in the Cassandra database; in some SQL dialects it is called a merge.

Now, our goal here is not efficiency, it's to provide clear and concise code. Efficiency comes later.

To that goal, the meat of the upsert, the `apply-game-rating` function, simply removes any prior row, and then adds a new row with the provided rating value.

2.11.2 Schema Changes

Our only change to the schema is to introduce the new mutation.

Listing 27: resources/cgg-schema.edn

```
{:objects
 {:BoardGame
  {:description "A physical or virtual board game."
   :fields
   {:id {:type (non-null ID)}
    :name {:type (non-null String)}
    :rating_summary {:type (non-null :GameRatingSummary)
                     :resolve :BoardGame/rating-summary}
    :summary {:type String
              :description "A one-line summary of the game."}
    :description {:type String
                  :description "A long-form description of the game."}
    :designers {:type (non-null (list :Designer))
                :description "Designers who contributed to the game."
                :resolve :BoardGame/designers}
    :min_players {:type Int
                  :description "The minimum number of players the game supports."}
    :max_players {:type Int
                  :description "The maximum number of players the game supports."}
    :play_time {:type Int
                :description "Play time, in minutes, for a typical game."}}}
```

(continues on next page)

(continued from previous page)

```
  :GameRatingSummary
  {:description "Summary of ratings for a single game."
   :fields
   {:count {:type (non-null Int)
            :description "Number of ratings provided for the game.  Ratings are 1 to␣
↪5 stars."}
    :average {:type (non-null Float)
              :description "The average value of all ratings, or 0 if never rated."}}}

  :Member
  {:description "A member of Clojure Game Geek.  Members can rate games."
   :fields
   {:id {:type (non-null ID)}
    :member_name {:type (non-null String)
                  :description "Unique name of member."}
    :ratings {:type (list :GameRating)
              :description "List of games and ratings provided by this member."
              :resolve :Member/ratings}}}

  :GameRating
  {:description "A member's rating of a particular game."
   :fields
   {:game {:type (non-null :BoardGame)
           :description "The Game rated by the member."
           :resolve :GameRating/game}
    :rating {:type (non-null Int)
             :description "The rating as 1 to 5 stars."}}}

  :Designer
  {:description "A person who may have contributed to a board game design."
   :fields
   {:id {:type (non-null ID)}
    :name {:type (non-null String)}
    :url {:type String
          :description "Home page URL, if known."}
    :games {:type (non-null (list :BoardGame))
            :description "Games designed by this designer."
            :resolve :Designer/games}}}}

 :queries
 {:game_by_id
  {:type :BoardGame
   :description "Select a BoardGame by its unique id, if it exists."
   :args
   {:id {:type (non-null ID)}}
   :resolve :query/game-by-id}

  :member_by_id
  {:type :Member
   :description "Select a ClojureGameGeek Member by their unique id, if it exists."
   :args
   {:id {:type (non-null ID)}}
   :resolve :query/member-by-id}}

 :mutations
 {:rate_game
```

(continues on next page)

(continued from previous page)

```
  {:type :BoardGame
   :description "Establishes a rating of a board game, by a Member.

   On success (the game and member both exist), selects the BoardGame.
   Otherwise, selects nil and an error."
   :args
   {:game_id {:type (non-null ID)}
    :member_id {:type (non-null ID)}
    :rating {:type (non-null Int)
             :description "Game rating as a number between 1 and 5."}}
   :resolve :mutation/rate-game}}}
```

Mutations nearly always include field arguments to define what will be affected by the mutation, and how. Here we have to provide field arguments to identify the game, the member, and the new rating.

Just as with queries, it is necessary to define what value will be resolved by the mutation; typically, when a mutation modifies a single object, that object is resolved.

Here, resolving a GameRating didn't seem to provide value, and we arbitrarily decided to instead resolve the BoardGame . . . we could have just as easily resolved the Member instead. The right option is often revealed based on client requirements.

GraphQL doesn't have a way to describe error cases comparable to how it defines types: *every* field resolver may return errors instead of, or in addition to, an actual value. We attempt to document the kinds of errors that may occur as part of the operation's documentation.

2.11.3 Code Changes

Finally, we knit together the schema changes and the database changes in the `schema` namespace.

Listing 28: src/clojure_game_geek/schema.clj

```
(ns clojure-game-geek.schema
  "Contains custom resolvers and a function to provide the full schema."
  (:require
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.util :as util]
    [com.walmartlabs.lacinia.schema :as schema]
    [com.walmartlabs.lacinia.resolve :refer [resolve-as]]
    [com.stuartsierra.component :as component]
    [clojure-game-geek.db :as db]
    [clojure.edn :as edn]))

(defn game-by-id
  [db]
  (fn [_ args _]
    (db/find-game-by-id db (:id args))))

(defn member-by-id
  [db]
  (fn [_ args _]
    (db/find-member-by-id db (:id args))))

(defn rate-game
  [db]
  (fn [_ args _]
```

(continues on next page)

(continued from previous page)

```
    (let [{game-id :game_id
           member-id :member_id
           rating :rating} args
          game (db/find-game-by-id db game-id)
          member (db/find-member-by-id db member-id)]
      (cond
        (nil? game)
        (resolve-as nil {:message "Game not found."
                         :status 404})

        (nil? member)
        (resolve-as nil {:message "Member not found."
                         :status 404})

        (not (<= 1 rating 5))
        (resolve-as nil {:message "Rating must be between 1 and 5."
                         :status 400})

        :else
        (do
          (db/upsert-game-rating db game-id member-id rating)
          game)))))

(defn board-game-designers
  [db]
  (fn [_ _ board-game]
    (db/list-designers-for-game db (:id board-game))))

(defn designer-games
  [db]
  (fn [_ _ designer]
    (db/list-games-for-designer db (:id designer))))

(defn rating-summary
  [db]
  (fn [_ _ board-game]
    (let [ratings (map :rating (db/list-ratings-for-game db (:id board-game)))
          n (count ratings)]
      {:count n
       :average (if (zero? n)
                  0
                  (/ (apply + ratings)
                     (float n)))})))

(defn member-ratings
  [db]
  (fn [_ _ member]
    (db/list-ratings-for-member db (:id member))))

(defn game-rating->game
  [db]
  (fn [_ _ game-rating]
    (db/find-game-by-id db (:game_id game-rating))))

(defn resolver-map
  [component]
  (let [db (:db component)]
```

(continues on next page)

(continued from previous page)

```
    {:query/game-by-id (game-by-id db)
     :query/member-by-id (member-by-id db)
     :mutation/rate-game (rate-game db)
     :BoardGame/designers (board-game-designers db)
     :BoardGame/rating-summary (rating-summary db)
     :GameRating/game (game-rating->game db)
     :Designer/games (designer-games db)
     :Member/ratings (member-ratings db)}))

(defn load-schema
  [component]
  (-> (io/resource "cgg-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers (resolver-map component))
      schema/compile))

(defrecord SchemaProvider [schema]

  component/Lifecycle

  (start [this]
    (assoc this :schema (load-schema this)))

  (stop [this]
    (assoc this :schema nil)))

(defn new-schema-provider
  []
  {:schema-provider (-> {}
                        map->SchemaProvider
                        (component/using [:db]))})
```

It all comes together in the `rate-game` function; we first check that the `game_id` and `member_id` passed in are valid (that is, they map to actual BoardGames and Members).

The `resolve-as` function is essential here: the first parameter is the value to resolve and is often nil when there are errors. The second parameter is an error map.[1]

`resolve-as` returns a wrapper object around the resolved value (which is nil in these examples) and the error map. Lacinia will later pull out the error map, add additional details, and add it to the `:errors` key of the result map.

These examples also show the use of the `:status` key in the error map. lacinia-pedestal will look for such values in the result map, and will set the HTTP status of the response to any value it finds (if there's more than one, the HTTP status will be the maximum). The `:status` keys are stripped out of the error maps *before* the response is sent to the client.

2.11.4 At the REPL

Let's start by seeing the initial state of things, using the default database:

```
(q "{ member_by_id(id: \"1410\") { member_name ratings { game { id name } rating }}}")
=>
{:data {:member_by_id {:member_name "bleedingedge",
```

(continues on next page)

[1] It can also be a seq of error maps, each containing, at a minimum, a `:message` key.

(continued from previous page)

```
                        :ratings [{:game {:id "1234", :name "Zertz"}, :rating 5}
                                  {:game {:id "1236", :name "Tiny Epic Galaxies"},␣
↪:rating 4}
                                  {:game {:id "1237", :name "7 Wonders: Duel"},␣
↪:rating 4}]}}}
```

Ok, so maybe we've soured on Tiny Epic Galaxies for the moment:

```
(q "mutation { rate_game(member_id: \"1410\", game_id: \"1236\", rating: 3) { rating_
↪summary { count average }}}")
=> {:data {:rate_game {:rating_summary {:count 1, :average 3.0}}}}

(q "{ member_by_id(id: \"1410\") { member_name ratings { game { id name } rating }}}")
=>
{:data {:member_by_id {:member_name "bleedingedge",
                        :ratings [{:game {:id "1236", :name "Tiny Epic Galaxies"},␣
↪:rating 3}
                                  {:game {:id "1234", :name "Zertz"}, :rating 5}
                                  {:game {:id "1237", :name "7 Wonders: Duel"},␣
↪:rating 4}]}}}
```

Dominion is a personal favorite, so let's rate that:

```
(q "mutation { rate_game(member_id: \"1410\", game_id: \"1235\", rating: 4) { name␣
↪rating_summary { count average }}}")
=> {:data {:rate_game {:name "Dominion", :rating_summary {:count 1, :average 4.0}}}}
```

We can also see what happens when the query contains mistakes[2]:

```
(q "mutation { rate_game(member_id: \"1410\", game_id: \"9999\", rating: 4) { name␣
↪rating_summary { count average }}}")
=>
{:data {:rate_game nil},
 :errors [{:message "Game not found.",
           :status 404,
           :locations [{:line 1, :column 9}],
           :query-path [:rate_game],
           :arguments {:member_id "1410", :game_id "9999", :rating "4"}}]}
```

Although the `rate-game` field resolver just returned a simple map (with keys `:message` and `:status`), Lacinia has enhanced the map identifying the location (within the query document), the query path (which indicates which operation or nested field was involved), and the arguments passed to the field resolver function.

In Lacinia, there's a difference between a resolver error, from using `resolve-as`, and an overall failure parsing or executing the query. If the `rating` argument is omitted from the query, we can see a significant difference:

```
(q "mutation { rate_game(member_id: \"1410\", game_id: \"9999\") { name rating_
↪summary { count average }}}")
=>
{:errors [{:message "Exception applying arguments to field `rate_game': Not all non-
↪nullable arguments have supplied values.",
           :query-path [],
           :locations [{:line 1, :column 9}],
           :field :rate_game,
           :missing-arguments [:rating]}]}
```

[2] In June 2018 update to the GraphQL specification, the format of error maps in the result map changed; when the tutorial upgrades to a version 0.29.0 of Lacinia, or later, the structure of the maps in the `:errors` key will change somewhat.

Here, the result map contains *only* the `:errors` key; the `:data` key is missing. A similar error would occur if the type of value provided to field argument is unacceptible:

```
(q "mutation { rate_game(member_id: \"1410\", game_id: \"9999\", rating: \"Great!\")
↪{ name rating_summary { count average }}}")
=>
{:errors [{:message "Exception applying arguments to field `rate_game': For argument␣
↪`rating', scalar value is not parsable as type `Int'.",
           :query-path [],
           :locations [{:line 1, :column 9}],
           :field :rate_game,
           :argument :rating,
           :value "Great!",
           :type-name :Int}]}
```

2.12 External Database, Phase 1

We've gone pretty far with our application so far, but it's time to make that big leap, and convert things over to an actual database. We'll be running PostgreSQL in a Docker container.[1]

We're definitely going to be taking two steps backward before taking further steps forward, but the majority of the changes will be in the `clojure-game-geek.db` namespace; the majority of the application, including the field resolvers, will be unaffected.

2.12.1 Dependency Changes

Listing 29: project.clj

```
(defproject clojure-game-geek "0.1.0-SNAPSHOT"
  :description "A tiny BoardGameGeek clone written in Clojure with Lacinia"
  :url "https://github.com/walmartlabs/clojure-game-geek"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.9.0"]
                 [com.stuartsierra/component "0.3.2"]
                 [com.walmartlabs/lacinia "0.30.0"]
                 [com.walmartlabs/lacinia-pedestal "0.10.0"]
                 [org.clojure/java.jdbc "0.7.8"]
                 [org.postgresql/postgresql "42.2.5.jre7"]
                 [com.mchange/c3p0 "0.9.5.2"]
                 [io.aviso/logging "0.3.1"]])
```

We're bringing in the very latest versions of lacinia and lacinia-pedestal (something we'll likely do almost every chapter). Since these are now based on Clojure 1.9, it's a fine time to upgrade to that.

We're also adding several new dependencies for accessing a PostgreSQL database:

- The Clojure `java.jdbc` library

[1] A Docker container is the Inception of computers; a container is essentially a light-weight virtual machine that runs inside your computer.
To the PostgreSQL server running inside the container, it will appear as if the entire computer is running Linux, just as if Linux and PostgreSQL were installed on a bare-metal computer.
Docker images are smaller and less demanding than full operating system virtual machines. In fact frequently you will run several interconnected containers together.
Docker includes infrastructure for downloading the images from a central repository. Ultimately, it's faster and easier to get PostgreSQL running inside a container that to install the database onto your computer.

- The PostgreSQL driver that plugs into the library
- A java library, c3p0, that is used for connection pooling

2.12.2 Database Initialization

We've added a number of scripts to project.

First, a file used to start PostgreSQL:

Listing 30: docker-compose.yml

```
version: '3'
services:
  db:
    ports:
    - 25432:5432
    image: postgres:10.2-alpine
```

This file is used with the docker-compose command to set up one or more containers. We only define a single container right now.

The image key identifies the name of the image to download from hub.docker.com.

The port mapping is part of the magic of Docker . . . the PostgreSQL server, inside the container, will listen to requests on its normal port: 5432, but our code, running on the host operation system, can reach the server as port 25432 on localhost.

The docker-up.sh script is used to start the container:

Listing 31: bin/docker-up.sh

```
#!/usr/bin/env bash

docker-compose -p cgg up -d
```

There's also a bin/docker-down.sh script to shut down the container, and a bin/psql.sh to launch a SQL command prompt for the cggdb database.

After starting the container, it is necessary to create the cggdb database and populate it with initial data, using the setup-db.sh script:

Listing 32: bin/setup-db.sh

```
#!/usr/bin/env bash

docker exec -i --user postgres cgg_db_1 createdb cggdb

docker exec -i --user postgres cgg_db_1 psql cggdb -a  <<__END
create user cgg_role password 'lacinia';
__END

docker exec -i cgg_db_1 psql -Ucgg_role cggdb -a <<__END
drop table if exists designer_to_game;
drop table if exists game_rating;
drop table if exists member;
drop table if exists board_game;
```

(continues on next page)

(continued from previous page)

```
drop table if exists designer;

CREATE OR REPLACE FUNCTION mantain_updated_at()
RETURNS TRIGGER AS \$\$
BEGIN
   NEW.updated_at = now();
   RETURN NEW;
END;
\$\$ language 'plpgsql';

create table member (
  member_id int generated by default as identity primary key,
  name text not null,
  created_at timestamp not null default current_timestamp,
  updated_at timestamp not null default current_timestamp);

create trigger member_updated_at before update
on member for each row execute procedure
mantain_updated_at();

create table board_game (
  game_id int generated by default as identity primary key,
  name text not null,
  summary text,
  min_players integer,
  max_players integer,
  created_at timestamp not null default current_timestamp,
  updated_at timestamp not null default current_timestamp);

create trigger board_game_updated_at before update
on board_game for each row execute procedure
mantain_updated_at();

create table designer (
  designer_id int generated by default as identity primary key,
  name text not null,
  uri text,
  created_at timestamp not null default current_timestamp,
  updated_at timestamp not null default current_timestamp);

create trigger designer_updated_at before update
on designer for each row execute procedure
mantain_updated_at();

create table game_rating (
  game_id int references board_game(game_id),
  member_id int references member(member_id),
  rating integer not null,
  created_at timestamp not null default current_timestamp,
  updated_at timestamp not null default current_timestamp);

create trigger game_rating_updated_at before update
on game_rating for each row execute procedure
mantain_updated_at();

create table designer_to_game (
  designer_id int  references designer(designer_id),
```

(continues on next page)

(continued from previous page)

```
  game_id int  references board_game(game_id),
  primary key (designer_id, game_id));

insert into board_game (game_id, name, summary, min_players, max_players) values
  (1234, 'Zertz', 'Two player abstract with forced moves and shrinking board', 2, 2),
  (1235, 'Dominion', 'Created the deck-building genre; zillions of expansions', 2,␣
↪null),
  (1236, 'Tiny Epic Galaxies', 'Fast dice-based sci-fi space game with a bit of chaos
↪', 1, 4),
  (1237, '7 Wonders: Duel', 'Tense, quick card game of developing civilizations', 2,␣
↪2);

alter table board_game alter column game_id restart with 1300;

insert into member (member_id, name) values
  (37, 'curiousattemptbunny'),
  (1410, 'bleedingedge'),
  (2812, 'missyo');

alter table member alter column member_id restart with 2900;

insert into designer (designer_id, name, uri) values
  (200, 'Kris Burm', 'http://www.gipf.com/project_gipf/burm/burm.html'),
  (201, 'Antoine Bauza', 'http://www.antoinebauza.fr/'),
  (202, 'Bruno Cathala', 'http://www.brunocathala.com/'),
  (203, 'Scott Almes', null),
  (204, 'Donald X. Vaccarino', null);

alter table designer alter column designer_id restart with 300;

insert into designer_to_game (designer_id, game_id) values
  (200, 1234),
  (201, 1237),
  (204, 1235),
  (203, 1236),
  (202, 1237);

insert into game_rating (game_id, member_id, rating) values
  (1234, 37, 3),
  (1234, 1410, 5),
  (1236, 1410, 4),
  (1237, 1410, 4),
  (1237, 2812, 4),
  (1237, 37, 5);
__END
```

The DDL for the `cggdb` database includes a pair of timestamp columns, `created_at` and `updated_at`, in most tables. Defaults and database triggers ensure that these are maintained by PostgreSQL.

2.12.3 Primary Keys

There's a problem with the data model we've used in prior chapters: the primary keys.

We've been using simple numeric strings as primary keys, because it was convenient. Literally, we just made up those values. But eventually, we're going to be writing data to the database, including new Board Games, new Publishers, and new Members.

With the change to using PostgreSQL, we've switched to using numeric primary keys. Not only are these more space efficient, but we have set up PostgreSQL to allocate them automatically. This is great news once we have multiple Clojure Game Geek servers running, as it ensures that primary keys are truly unique, enforced by the database. We'll circle back to this issue when we add mutations to create new entities.

In the meantime, the schema has changed; the id fields have changed type from type `ID` (which, in GraphQL, is a kind of opaque string) to type `Int` (which is a 32 bit, signed integer).

Listing 33: resources/cgg-schema.edn

```
{:objects
 {:BoardGame
  {:description "A physical or virtual board game."
   :fields
   {:game_id {:type (non-null Int)}
    :name {:type (non-null String)}
    :rating_summary {:type (non-null :GameRatingSummary)
                     :resolve :BoardGame/rating-summary}
    :summary {:type String
              :description "A one-line summary of the game."}
    :description {:type String
                  :description "A long-form description of the game."}
    :designers {:type (non-null (list :Designer))
                :description "Designers who contributed to the game."
                :resolve :BoardGame/designers}
    :min_players {:type Int
                  :description "The minimum number of players the game supports."}
    :max_players {:type Int
                  :description "The maximum number of players the game supports."}
    :play_time {:type Int
                :description "Play time, in minutes, for a typical game."}}}

  :GameRatingSummary
  {:description "Summary of ratings for a single game."
   :fields
   {:count {:type (non-null Int)
            :description "Number of ratings provided for the game.  Ratings are 1 to␣
↪5 stars."}
    :average {:type (non-null Float)
              :description "The average value of all ratings, or 0 if never rated."}}}

  :Member
  {:description "A member of Clojure Game Geek.  Members can rate games."
   :fields
   {:member_id {:type (non-null Int)}
    :member_name {:type (non-null String)
                  :description "Unique name of member."}
    :ratings {:type (list :GameRating)
              :description "List of games and ratings provided by this member."
              :resolve :Member/ratings}}}

  :GameRating
  {:description "A member's rating of a particular game."
   :fields
   {:game {:type (non-null :BoardGame)
           :description "The Game rated by the member."
           :resolve :GameRating/game}
    :rating {:type (non-null Int)
```

(continues on next page)

(continued from previous page)

```
             :description "The rating as 1 to 5 stars."}}}

  :Designer
  {:description "A person who may have contributed to a board game design."
   :fields
   {:designer_id {:type (non-null Int)}
    :name {:type (non-null String)}
    :url {:type String
          :description "Home page URL, if known."}
    :games {:type (non-null (list :BoardGame))
            :description "Games designed by this designer."
            :resolve :Designer/games}}}}

 :queries
 {:game_by_id
  {:type :BoardGame
   :description "Select a BoardGame by its unique id, if it exists."
   :args
   {:id {:type (non-null Int)}}
   :resolve :query/game-by-id}

  :member_by_id
  {:type :Member
   :description "Select a ClojureGameGeek Member by their unique id, if it exists."
   :args
   {:id {:type (non-null Int)}}
   :resolve :query/member-by-id}}

 :mutations
 {:rate_game
  {:type :BoardGame
   :description "Establishes a rating of a board game, by a Member.

   On success (the game and member both exist), selects the BoardGame.
   Otherwise, selects nil and an error."
   :args
   {:game_id {:type (non-null Int)}
    :member_id {:type (non-null Int)}
    :rating {:type (non-null Int)
             :description "Game rating as a number between 1 and 5."}}
   :resolve :mutation/rate-game}}}
```

In addition, the `id` field on the BoardGame, Member, and Publisher objects has been renamed: to `game_id`, `member_id`, and `publisher_id`, respectfully. This aligns the field names with the database column names.

As Clojure developers, we generally follow the *kebab case* convention of using dashes in names. GraphQL, JSON, and most databases use *snake case*, with underscores. Snake case keywords in Clojure look slightly odd, but are 100% valid.

There's nothing that prevents you from reading database data and converting the column names to kebab case . . . but you'll just have to undo that somehow in the GraphQL schema, as kebab case is not valid for GraphQL names.

Much better to have as consistent a representation of the data as possible, spanning the database, the GraphQL schema, the Clojure data access code, and the over-the-wire JSON format . . . and not buy yourself any extra work that has no tangible benefits.

2.12.4 org.clojure/java.jdbc

This library is the standard approach to accessing a database from Clojure code. `java.jdbc` can access, in a uniform manner, any database for which there is a Java JDBC driver.

The `clojure.java.jdbc` namespace contains a number of functions for acessing a database, including functions for executing arbitrary queries, and specialized functions for peforming inserts, updates, and deletes.

For all of those functions, the first parameter is a *database spec*, a map of data used to connect to the database. In a trivial case, this identifies the Java JDBC driver class, and provides extra information to build a JDBC URL, including details such as the database host, the user and password, and the name of the database.

In practice, opening up a new connection for each operation is unacceptible so we'll jump right in with a database connection pooling library, `C3P0`.

2.12.5 clojure-game-geek.db

In prior chapters, the `:db` component was just a wrapper around an Atom; starting here, we're going to update it to be a wrapper around a connection to the PostgreSQL database running in the Docker container.

Our goal in this chapter is to update just one basic query to use the database, the query that retrieves a board game by its unique id. We'll make just the changes necessary for that one query before moving on.

Listing 34: src/clojure_game_geek/db.clj

```clojure
(ns clojure-game-geek.db
  (:require
    [com.stuartsierra.component :as component]
    [clojure.java.jdbc :as jdbc])
  (:import (com.mchange.v2.c3p0 ComboPooledDataSource)))

(defn ^:private pooled-data-source
  [host dbname user password port]
  {:datasource
   (doto (ComboPooledDataSource.)
     (.setDriverClass "org.postgresql.Driver" )
     (.setJdbcUrl (str "jdbc:postgresql://" host ":" port "/" dbname))
     (.setUser user)
     (.setPassword password))})

(defrecord ClojureGameGeekDb [ds]

  component/Lifecycle

  (start [this]
    (assoc this
           :ds (pooled-data-source "localhost" "cggdb" "cgg_role" "lacinia" 25432)))

  (stop [this]
    (-> ds :datasource .close)
    (assoc this :ds nil)))

(defn new-db
  []
  {:db (map->ClojureGameGeekDb {})})
```

(continues on next page)

(continued from previous page)

```
(defn find-game-by-id
  [component game-id]
  (first
    (jdbc/query (:ds component)
                ["select game_id, name, summary, min_players, max_players, created_at,
↪ updated_at
               from board_game where game_id = ?" game-id])))

(defn find-member-by-id
  [component member-id]
  (->> component
       :db
       deref
       :members
       (filter #(= member-id (:id %)))
       first))

(defn list-designers-for-game
  [component game-id]
  (let [designers (:designers (find-game-by-id component game-id))]
    (->> component
         :db
         deref
         :designers
         (filter #(contains? designers (:id %))))))

(defn list-games-for-designer
  [component designer-id]
  (->> component
       :db
       deref
       :games
       (filter #(-> % :designers (contains? designer-id)))))

(defn list-ratings-for-game
  [component game-id]
  (->> component
       :db
       deref
       :ratings
       (filter #(= game-id (:game_id %)))))

(defn list-ratings-for-member
  [component member-id]
  (->> component
       :db
       deref
       :ratings
       (filter #(= member-id (:member_id %)))))

(defn ^:private apply-game-rating
  [game-ratings game-id member-id rating]
  (->> game-ratings
       (remove #(and (= game-id (:game_id %))
                     (= member-id (:member_id %))))
       (cons {:game_id game-id
              :member_id member-id
```

(continues on next page)

(continued from previous page)

```
               :rating rating})))

(defn upsert-game-rating
  "Adds a new game rating, or changes the value of an existing game rating."
  [db game-id member-id rating]
  (-> db
      :db
      (swap! update :ratings apply-game-rating game-id member-id rating)))
```

The requires for the `db` namespace have changed; we're using the `clojure.java.jdbc` namespace to connect to the database and execute queries, and also making use of the `ComboPooledDataSource` class, which allows for pooled connections.

The ClojureGameGeekDb record has changed; it now has a `ds` (data source) field, and that is the connection to the PostgreSQL database. The `start` method now opens the connection pool to the database, and the `stop` method shuts down the connection pool.

For the meantime, we've hardwired the connection details (hostname, username, password, and port) to our Docker container. A later chapter will discuss approaches to configuration. Also note that we're connecting to port `25432` on `localhost`; Docker will forward that port to the container port `5432`.

That leaves the revised implementation of the `find-game-by-id` function; the only data access function rewritten to use the database connection. It simply constructs and executes the SQL query.

With `clojure.java.jdbc` the query is a vector consisting of a SQL query string followed by zero or more query parameters. Each `?` character in the query string corresponds to a query parameter, based on position.

The `clojure.java.jdbc/query` function returns a seq of matching rows. By default, each selected row is converted into a Clojure map, and the column names are converted from strings into keywords.

For an operation like this one, which returns at most one map, we use `first`.

If no rows match, then the seq will be empty, and `first` will return nil. That's a perfectly good way to identify that the provided Board Game id was not valid.

2.12.6 At the REPL

Starting a new REPL, we can give the new code and schema a test:

```
(start)
=> :started
(q "{ game_by_id(id: 1234) { game_id name summary min_players max_players }}")
=>
{:data {:game_by_id {:game_id 1234,
                     :name "Zertz",
                     :summary "Two player abstract with forced moves and shrinking␣
↪board",
                     :min_players 2,
                     :max_players 2}}}")
```

Great! That works!

Notice how everything fits together: the column names in the database (`game_id`, `summary`, etc.) became keywords (`:game_id`, `:summary`, etc.) in a map; meanwhile the GraphQL field names did the same conversion and everything meets together in the middle, with GraphQL fields selecting those same keys from the map.

Meanwhile all the other `clojure-game-geek.db` namespace functions, expecting to operate against a map inside an Atom, are now broken. We'll fix them in the next couple of chapters.

2.12.7 User Namespace Improvements

We've made some tiny changes to the `user` namespace:

Listing 35: dev-resources/user.clj

```clojure
(ns user
  (:require
    [com.walmartlabs.lacinia :as lacinia]
    [clojure.java.browse :refer [browse-url]]
    [clojure-game-geek.system :as system]
    [clojure.walk :as walk]
    [com.stuartsierra.component :as component])
  (:import (clojure.lang IPersistentMap)))

(defn simplify
  "Converts all ordered maps nested within the map into standard hash maps, and
   sequences into vectors, which makes for easier constants in the tests, and␣
↪eliminates ordering problems."
  [m]
  (walk/postwalk
    (fn [node]
      (cond
        (instance? IPersistentMap node)
        (into {} node)

        (seq? node)
        (vec node)

        :else
        node))
    m))

(defonce system nil)

(defn q
  [query-string]
  (-> system
      :schema-provider
      :schema
      (lacinia/execute query-string nil nil)
      simplify))

(defn start
  []
  (alter-var-root #'system (fn [_]
                             (-> (system/new-system)
                                 component/start-system)))
  (browse-url "http://localhost:8888/")
  :started)

(defn stop
  []
  (when (some? system)
    (component/stop-system system)
    (alter-var-root #'system (constantly nil)))
  :stopped)
```

(continues on next page)

(continued from previous page)

```
(comment
  (start)
  (stop)
  )
```

To make loading and reloading the `user` namespace easier, we've changed the `system` Var to be a `defonce`. This means that even if the code for the namespace is reloaded, the `system` Var will maintain its value from before the code was reloaded.

A common cycle is to make code changes, `stop`, then `start` the system.

We've moved the code that contructs a new system into the `start` function, and changed the `stop` function to return the `system` Var to nil after stopping the system, if a system is in fact running.

Lastly, there's a comment containing expressions to start and stop the system. This is great for REPL oriented development, we can use the Cursive *send form before caret to REPL* command (Shift-Ctrl E)[2] to make it easier to quickly and accurately execute those commands.

2.12.8 Next Up

We've been sloppy about one aspect of our application: we've entirely been testing at the REPL. It's time to write some tests, then convert the rest of the `db` namespace.

2.13 Testing, Phase 1

Before we get much further, we are very far along for code that has no tests. Let's fix that.

First, we need to reorganize a couple of things, to make testing easier.

2.13.1 HTTP Port

Let's save ourselves some frustration: when we run our tests, we can't know if there is a REPL-started system running or not. There's no problem with two complete system maps running at the same time, and even hitting the same database, all within a single process . . . that's why we like Component, as it helps us avoid unecessary globals.

Unfortunately, we still have one conflict: the HTTP port for inbound requests. Only one of the systems can bind to the default 8888 port, so let's make sure our tests use a different port.

Listing 36: src/clojure_game_geek/server.clj

```
(ns clojure-game-geek.server
  (:require [com.stuartsierra.component :as component]
            [com.walmartlabs.lacinia.pedestal :as lp]
            [io.pedestal.http :as http]))

(defrecord Server [schema-provider server port]

  component/Lifecycle
  (start [this]
    (assoc this :server (-> schema-provider
                            :schema
```

(continues on next page)

[2] The author uses Cursive, but Emacs and other editors all have similar functionality.

(continued from previous page)

```
                              (lp/service-map {:graphiql true
                                               :port port})
                              http/create-server
                              http/start)))

  (stop [this]
    (http/stop server)
    (assoc this :server nil)))

(defn new-server
  []
  {:server (component/using (map->Server {:port 8888})
                            [:schema-provider])})
```

We've added a bit of configuration for the `:server` component, the port to bind to. This will make it possible for our test code to use a different port.

2.13.2 Simplify Utility

To keep our tests simple, we'll want to use the `simplify` utility function discussed earlier. Here, we're creating a new namespace for test utilities, and moving the `simplify` function from the `user` namespace to the `test-utils` namespace:

Listing 37: dev-resources/clojure_game_geek/test_utils.clj

```
(ns clojure-game-geek.test-utils
  (:require
    [clojure.walk :as walk])
  (:import
    (clojure.lang IPersistentMap)))

(defn simplify
  "Converts all ordered maps nested within the map into standard hash maps, and
   sequences into vectors, which makes for easier constants in the tests, and␣
↪eliminates ordering problems."
  [m]
  (walk/postwalk
    (fn [node]
      (cond
        (instance? IPersistentMap node)
        (into {} node)

        (seq? node)
        (vec node)

        :else
        node))
    m))
```

This is located in the `dev-resource` folder, so that that Leiningen won't treat it as a namespace containing tests to execute.

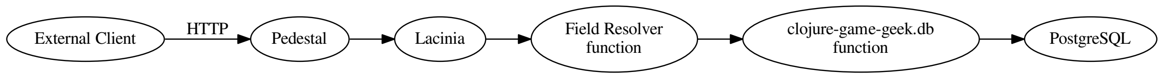
Over time, we're likely to add a number of little tools here to make tests more clear and concise.

2.13.3 Integration or Unit? Yes

When it comes to testing, your first thought should be at what level of granularity testing should occur. *Unit testing* is generally testing the smallest possible bit of code; in Clojure terms, testing a single function, ideally isolated from everything else.

Integration testing is testing at a higher level, testing how several elements of the system work together.

Our application is layered as follows:



In theory, we could test each layer separately; that is, we could test the `clojure-game-geek.db` functions against a database (or even, some mockup of a database), then test the field resolver functions against the `db` functions, etc.

In practice, building a Lacinia application is an exercise in integration; the individual bits of code are often quite small and simple, but there can be issues with how these bits of code interact.

I prefer a modest amount of integration testing using a portion of the full stack.

There's no point in testing a block of database code, only to discover that the results don't work with the field resolver functions calling that code. Likewise, for nominal success cases, there's no point in testing the raw database code if the exact same code will be exercised when testing the field resolver functions.

There's still a place for more focused testing, especially testing o failure scenarios and other edge cases.

Likewise, as we build up more code in our application outside of Lacinia, such as request authentication and authorization, we may want to exercise our code by sending HTTP requests in from the tests.

For our first test, we'll do some integration testing; our tests will start at the Lacinia step from the diagram above, and work all the way down to the database instance (running in our Docker container).

To that mind, we want to start up the schema connected to field resolvers, the `db` namespace, and the database itself. The easiest way to do this start up a new system, and extract the pieces we need from the running system map.

2.13.4 First Test

Our first test will replicate a bit of the manual testing we've already done in the REPL: reading an existing board game by its primary key.

Listing 38: test/clojure_game_geek/system_tests.clj

```
(ns clojure-game-geek.system-tests
  (:require
    [clojure.test :refer [deftest is]]
    [clojure-game-geek.system :as system]
    [clojure-game-geek.test-utils :refer [simplify]]
    [com.stuartsierra.component :as component]
    [com.walmartlabs.lacinia :as lacinia]))

(defn ^:private test-system
  "Creates a new system suitable for testing, and ensures that
  the HTTP port won't conflict with a default running system."
  []
  (-> (system/new-system)
```

(continues on next page)

(continued from previous page)

```
      (assoc-in [:server :port] 8989)))

(defn ^:private q
  "Extracts the compiled schema and executes a query."
  [system query variables]
  (-> system
      (get-in [:schema-provider :schema])
      (lacinia/execute query variables nil)
      simplify))

(deftest can-read-board-game
  (let [system (component/start-system (test-system))
        results (q system
                   "{ game_by_id(id: 1234) { name summary min_players max_players␣
↪play_time }}"
                   nil)]
    (is (= {:data {:game_by_id {:max_players 2
                                :min_players 2
                                :name "Zertz"
                                :play_time nil
                                :summary "Two player abstract with forced moves and␣
↪shrinking board"}}}
           results))
    (component/stop-system system)))
```

We're making use of the standard `clojure.test` library.

The `test-system` function builds a standard system, but overrides the HTTP port, as dicussed above.

We use that function to create and start a system for our first test. This first test is a bit verbose; later we'll refactor some of the code out of it, to make writing additional tests easier.

Because we control the initial test data[1] we know what at least a couple of rows in our database look like.

It's quite easy to craft a tiny GraphQL query and execute it; that will flow through Lacinia, to our field resolvers, to the database access code, and ultimately to the database, just like in the diagram.

2.13.5 Running the Tests

There's a number of ways to run Clojure tests.

From the command line, `lein test`:

```
~/workspaces/github/clojure-game-geek > lein test

lein test clojure-game-geek.system-tests

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

But who wants to do that all the time?

Clojure startup time is somewhat slow, as before your tests can run, large numbers of Java classes must be loaded, and signifcant amounts of Clojure code, both from our application and from any libraries, must be read, parsed, and compiled.

[1] An improved approach might be to create a fresh database namespace for each test, or each test namespace, and create and populate the tables with fresh test data each time. This might be very important when attempting to run these tests inside a Continuous Integration server.

Fortunately, Clojure was created with a REPL-oriented development workflow in mind. This is a fast-feedback cycle, where you can run tests, diagnose failures, make code corrections, and re-run the tests in a matter of seconds. Generally, the slowest part of the loop is the part that executes inside your grey matter.

Because the Clojure code base is already loaded and running, even a change that affects many namespaces can be reloaded in milliseconds.

If you are using an IDE, you will be able to run tests directly in a running REPL. In Cursive, `Ctrl-Shift-T` runs all tests in the current namespace, and `Ctrl-Alt-Cmd-T` runs just the test under the cursor. Cursive is even smart enough to properly reload all modified namespaces before executing the tests.

Similar commands exist for whichever editor you are using. Being able to load code and run tests in a fraction of a second is incredibly liberating if you are used to a more typical grind of starting a new process just to run tests[2] .

2.13.6 Database Issues

These tests assume the database is running locally, and has been initialized.

What if it's not? It might look like this:

```
lein test clojure-game-geek.system-tests
WARN  com.mchange.v2.resourcepool.BasicResourcePool - com.mchange.v2.resourcepool.
↪BasicResourcePool$ScatteredAcquireTask@614dbaad -- Acquisition Attempt Failed!!!␣
↪Clearing pending acquires. While trying to acquire a needed new resource, we failed␣
↪to succeed more than the maximum number of allowed acquisition attempts (30). Last␣
↪acquisition attempt exception:
org.postgresql.util.PSQLException: Connection to localhost:25432 refused. Check that␣
↪the hostname and port are correct and that the postmaster is accepting TCP/IP␣
↪connections.
        at org.postgresql.core.v3.ConnectionFactoryImpl.
↪openConnectionImpl(ConnectionFactoryImpl.java:280)
        at org.postgresql.core.ConnectionFactory.openConnection(ConnectionFactory.
↪java:49)
        at org.postgresql.jdbc.PgConnection.<init>(PgConnection.java:195)
        at org.postgresql.Driver.makeConnection(Driver.java:454)
        at org.postgresql.Driver.connect(Driver.java:256)
        at com.mchange.v2.c3p0.DriverManagerDataSource.
↪getConnection(DriverManagerDataSource.java:175)
        at com.mchange.v2.c3p0.WrapperConnectionPoolDataSource.
↪getPooledConnection(WrapperConnectionPoolDataSource.java:220)
        at com.mchange.v2.c3p0.WrapperConnectionPoolDataSource.
↪getPooledConnection(WrapperConnectionPoolDataSource.java:206)
        at com.mchange.v2.c3p0.impl.C3P0PooledConnectionPool
↪$1PooledConnectionResourcePoolManager.acquireResource(C3P0PooledConnectionPool.
↪java:20
...

Ran 1 tests containing 1 assertions.
0 failures, 1 errors.
Tests failed.
```

Because of the connection pooling, this actually takes quite some time to fail, and produces hundreds (!) of lines of exception output.

If you see a huge swath of tests failing, the first thing to do is double check external dependencies, such as the database running inside the Docker container.

[2] Downside: you'll probably read a lot less Twitter while developing.

2.13.7 Conclusion

We've created just one test, and managed to get it to run. That's a great start. Next up, we'll flesh out our tests, fix the many outdated functions in the `clojure-game-geek.db` namespace, and do some refactoring to ensure that our tests are concise, readable, and efficient.

2.14 External Database, Phase 2

Let's get the rest of the functions in the `clojure-game-geek.db` namespace working again and add tests for them. We'll do a little refactoring as well, to make both the production code and the tests clearer and simpler.

2.14.1 Logging

It's always a good idea to know exactly what SQL queries are executing in your application; you'll never figure out what's slowing down your application if you don't know what queries are even executing.

Listing 39: src/clojure_game_geek/db.clj

```clojure
(ns clojure-game-geek.db
  (:require
    [com.stuartsierra.component :as component]
    [io.pedestal.log :as log]
    [clojure.java.jdbc :as jdbc]
    [clojure.string :as str])
  (:import (com.mchange.v2.c3p0 ComboPooledDataSource)))

(defn ^:private pooled-data-source
  [host dbname user password port]
  {:datasource
   (doto (ComboPooledDataSource.)
     (.setDriverClass "org.postgresql.Driver")
     (.setJdbcUrl (str "jdbc:postgresql://" host ":" port "/" dbname))
     (.setUser user)
     (.setPassword password))})

(defrecord ClojureGameGeekDb [ds]

  component/Lifecycle

  (start [this]
    (assoc this
           :ds (pooled-data-source "localhost" "cggdb" "cgg_role" "lacinia" 25432)))

  (stop [this]
    (-> ds :datasource .close)
    (assoc this :ds nil)))

(defn new-db
  []
  {:db (map->ClojureGameGeekDb {})})


(defn ^:private query
  [component statement]
```

(continues on next page)

(continued from previous page)

```
  (let [[sql & params] statement]
    (log/debug :sql (str/replace sql #"\s+" " ")
               :params params))
  (jdbc/query (:ds component) statement))
```

We've introduced our own version of `clojure.java.jdbc/query` with two differences:

- It logs the SQL and parameters it will execute
- It accepts a component, and extracts the database specification from the component

Because of how we format the SQL in our code, it is useful to convert the embedded newlines and indentation into single spaces.

A bit about logging: In a typical Java, or even Clojure, application the focus on logging is on a textural message for the user to read. Different developers approach this in different ways ... everything from the inscrutably cryptic to the overly verbose. Yes, across that spectrum, there always an assumption that some user is reading the log.

The `io.pedestal/pedestal.log` library introduces a different idea: logs as a stream of data ... a sequence of maps. That's what we see in the call to `log/debug`: just keys and values that are interesting.

When logged, it may look like:

```
DEBUG clojure-game-geek.db - {:sql "select game_id, name, summary, min_players, max_
↪players, created_at, updated_at from board_game where game_id = $1",
  :params (1234), :line 38}
```

That's the debug level and namespace, and the map of keys and values (`io.pedestal.log` adds the `:line` key).

The useful and interesting details are present and unambiguously formatted, since it is not formatted specifically for a user to read.

This can be a very powerful concept; these logs can even be read back into memory, converted back into data, and operated on with all the `map`, `reduce`, and `filter` power that Clojure provides.[1]

After years of sweating the details on formatting (and capitalizing, and quoting, and punctuating) human-readible error messages, it is a joy to just throw whatever data is useful into the log, and not care about all those human oriented formatting details.

This is, of course, all possible because all data in Clojure can be printed out nicely and even read back in again. By comparison, data values or other objects in Java only have useful debugging output if their class provides an override of the `toString()` method.

When it comes time to execute a query, not much has changed except that it isn't necessary to extract the databased spec from the componment:

Listing 40: src/clojure_game_geek/db.clj

```
(defn find-game-by-id
  [component game-id]
  (first
    (query component
           ["select game_id, name, summary, min_players, max_players, created_at,␣
↪updated_at
             from board_game where game_id = ?" game-id])))
```

[1] I've used this on another project where a bug manifested only at a large scale of operations; by hooking into Logback and capturing the logged maps, it was possible to quickly filter through megabytes of output to the find the clues that revealed how the bug occured.

2.14.2 logback-test.xml

We can enable logging, just for testing purposes, in our `logback-test.xml`:

Listing 41: dev-resources/logback-test.xml

```
<configuration scan="true" scanPeriod="1 seconds">

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%-5level %logger - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="warn">
    <appender-ref ref="STDOUT"/>
  </root>

  <logger name="clojure-game-geek.db" level="DEBUG"/>

</configuration>
```

Nicely, Logback will pick up this change to the configuration file without a restart.

2.14.3 Re-running tests

If we switch back to the `clojure-game-geek.system-tests` namespace and re-run the tests, the debug output will be mixed into the test tool output:

```
Loading src/clojure_game_geek/db.clj... done
Loading test/clojure_game_geek/system_tests.clj... done
Running tests in clojure-game-geek.system-tests
DEBUG clojure-game-geek.db - {:sql "select game_id, name, summary, min_players, max_
↪players, created_at, updated_at from board_game where game_id = $1", :params (1234),
↪ :line 42}
Ran 1 test containing 1 assertion.
No failures.
```

2.14.4 More code updates

The remaining functions in `clojure-game-geek.db` can be rewritten to make use of `query` and operate on the real database:

Listing 42: src/clojure_game_geek/db.clj

```
(defn find-member-by-id
  [component member-id]
  (first
    (query component
           ["select member_id, name, created_at, updated_at
             from member
             where member_id = $1" member-id])))

(defn list-designers-for-game
  [component game-id]
```

(continues on next page)

(continued from previous page)

```
  (query component
         ["select d.designer_id, d.name, d.uri, d.created_at, d.updated_at
           from designer d
           inner join designer_to_game j on (d.designer_id = j.designer_id)
           where j.game_id = $1
           order by d.name" game-id]))

(defn list-games-for-designer
  [component designer-id]
  (query component
         ["select g.game_id, g.name, g.summary, g.min_players, g.max_players, g.
↪created_at, g.updated_at
           from board_game g
           inner join designer_to_game j on (g.game_id = j.game_id)
           where j.designer_id = $1
           order by g.name" designer-id]))

(defn list-ratings-for-game
  [component game-id]
  (query component
         ["select game_id, member_id, rating, created_at, updated_at
           from game_rating
           where game_id = $1" game-id]))

(defn list-ratings-for-member
  [component member-id]
  (query component
         ["select game_id, member_id, rating, created_at, updated_at
           from game_rating
           where member_id = $1" member-id]))

(defn upsert-game-rating
  "Adds a new game rating, or changes the value of an existing game rating.

  Returns nil"
  [component game-id member-id rating]
  (query component
         ["insert into game_rating (game_id, member_id, rating)
           values ($1, $2, $3)
           on conflict (game_id, member_id) do update set rating = $3"
          game-id member-id rating])

  nil)
```

The majority of this is quite straight-forward, except for `upsert-game-rating`, which makes use of PostgreSQL language extensions to handle a conflict (where a rating already exists for a particular game and member) - the insert is converted to an update.

In the next chapter, we'll focus on testing the code we've just added.

2.15 Tutorial Wrapup

That's as far as we've made it with the tutorial so far.

Expect many more updates, and soon!

Caution: This tutorial is a work-in-progress. It will likely be weeks or months of part-time work. PR's welcome!

Once the overall structure of the tutorial comes together, we'll make a pass over the pages to add more cross-reference links and so forth. We're aware that each chapter ends a bit suddenly.

Our goal with this tutorial is to build up the essentials of a full application implemented in Lacinia, starting from nothing.

Unlike many of the snippets used elsewhere in the Lacinia documentation, this will be something you can fork and experiment with yourself.

Along the way, we hope you'll learn quite a bit about not just Lacinia and GraphQL, but about building Clojure applications in general.

You can pull down the full source for the tutorial from GitHub: https://github.com/walmartlabs/clojure-game-geek

CHAPTER 3

Fields

Fields are the basic building block of GraphQL data.

Objects and *interfaces* are composed of fields. Queries and mutations are a special kind of field.

Fields are functions. Or, more specifically, fields are a kind of operation that begins with some data, adds in other details (such as field arguments provided in the query), and produces new data that can be incorporated into the overall result.

3.1 Field Definition

A field definition occurs in the schema to describe the type and other details of a field. A field definition is a map with specific keys.

3.2 Field Type

The main key in a field definition is `:type`, which is required. This is the type of value that may be returned by the field resolver, and is specified in terms of the type DSL.

3.3 Types DSL

Types are the essence of fields; they can represent scalar values (simple values, such as string or numbers), composite objects with their own fields, or lists of either scalars or objects.

In the schema, a type can be:

- A keyword corresponding to an object, interface, enum, or union
- A scalar type (built in, or schema defined)
- A non-nillable version of any of the above: `(non-null X)`

- A list of any of the above: `(list X)`

The built-in scalar types:

GraphQL Spec

Read about scalar types.

- String
- Float
- Int
- Boolean
- ID

Conventions

By convention, the built-in scalar types are identified as symbols, such as `{:type String}`. Other types (objects, interfaces, enums, and unions) are identified as keywords, `{:type (list :character)}`. It actually makes no difference; internally everything is converted to keywords in the compiled schema.

3.4 Field Resolver

The `:resolve` key identifies the field resolver function, used to provide the actual data.

This data, the *resolved value*, is never directly returned to the client; this is because in GraphQL, the client query identifies which fields from the resolved value are selected (and often, renamed) to form the result value.

`:resolve` is optional; when not provided it is assumed that the containing field's resolved value is a map containing a key exactly matching the field's name.

A field resolver function may be provided on a field inside an *object definition*, or inside a *query definition* or *mutation definition*. No field resolver function should be provided in the fields of an *interface definition*: if provided in an interface definition, the resolve will be silently ignored.

Field Resolvers

Please refer to the *full description of field resolvers*.

The field's resolver is passed the resolved value of the **containing** field, object, query, or mutation.

The return value may be a scalar type, or a structured type, as defined by the field's `:type`.

For composite (non-scalar) types, the client query **must** include a nested set of fields to be returned in the result map. The query is a tree, and the leaves of that tree will always be simple scalar values.

3.5 Arguments

A field may define arguments using the `:args` key; this is a map from argument name to an argument definition.

A field uses arguments to modify what data, and in what order, is to be returned. For example, arguments could set boundaries on a query based on date or price, or determine sort order.

Argument definitions define a value for `:type`, and may optionally provide a `:description`. Arguments do **not** have resolvers, as they represent explicit data from the client passed to the field.

Arguments may also have a `:default-value`. The default value is supplied to the field resolver when the request does not itself supply a value for the argument.

An argument that is not specified in the query, and does not have a default value, will be omitted from the argument map passed to the *field resolver*.

3.6 Description

A field may include a `:description` key; the value is a string exposed through *Introspection*.

3.7 Deprecation

A field may include a `:deprecation` key; this identifies that the field is *deprecated*.

CHAPTER 4

Objects

A schema object defines a single type of data that may be queried. This is often a mapping from data obtained from a database or other external store and exposed by GraphQL.

GraphQL Spec

Read about server-side types.

GraphQL supports objects, interfaces, unions, and enums.

A simple object defines a set of fields, each with a specific type. Object definitions are under the `:objects` key of the schema.

```
{:objects
 {:product
  {:fields {:id {:type ID}
            :name {:type String}
            :sku {:type String}
            :keywords {:type (list String)}}}}}
```

This defines a schema containing only a single schema object[1], *product*, with four fields:

- id - an identifier
- name - a string
- sku - a string
- keyword - a list of strings

[1] A schema that fails to define either queries or mutations is useful only as an example.

4.1 Field Definitions

An object definition contains a `:fields` key, whose value is a map from field name to *field definition*. Field names are keywords.

4.2 Interface Implementations

An object may implement zero or more *interfaces*. This is described using the `:implements` key, whose value is a list of keywords identifying interfaces.

Objects can only implement interfaces: there's no concept of inheritance from other objects.

An object definition must include all the fields from all implemented interfaces; failure to do so will cause an exception to be thrown when the schema is compiled.

In some cases, a field defined in an object may be more specific than a field from an inherited interface; for example, the field type in the interface may itself be an interface; the field type in the object must be that exact interface *or* an object that implements that interface.

In our Star Wars themed example schema, we see that the `:character` interface defines the `:friends` field as type `(list :character)`. So, in the generic case, the friends of a character can be either Humans or Droids.

Perhaps in a darker version of the Star Wars universe, Humans can not be friends with Droids. In that case, the `:friends` field of the `:human` object would be type `(list :human)` rather than the more egalitarian `(list :character)`. This appears to be a type conflict, as the type of the `:friends` field differs between `:human` and `:character`

In fact, this does not violate type constraints, because a Human is always a Character.

4.3 Object Description

An object definition may include a `:description` key; the value is a string exposed through *Introspection*.

When an object implement an interface, it may omit the `:description` of inherited fields, and on arguments of inherited fields to inherit the description from the interface.

CHAPTER 5

Interfaces

GraphQL supports the notion of *interfaces*, collections of fields and their arguments.

GraphQL Spec

Read about interfaces.

To keep things simple, interfaces can not extend other interfaces. Likewise, objects can implement multiple interfaces, but can not extend other objects.

Interfaces are valid types, they can be specified as the return type of a query, mutation, or as the type of a field.

```
{:interfaces
 {:named
  {:fields {:name {:type String}}}}

 :objects
 {:person
  {:implements [:named]
   :fields {:name {:type String}
            :age {:type Int}}}

  :business
  {:implements [:named]
   :fields {:name {:type String}
            :employee_count {:type Int}}}}}
```

An interface definition may include a `:description` key; the value is a string exposed through *Introspection*.

The description on an interface field, or on an argument of an interface field, will be inherited by the object field (or argument) unless overriden. This helps to elimiate duplication of documentation between an interface and the object implementing the interface.

The *object definition* must include all the fields of all extended interfaces.

Tip: When a field or operation type is an interface, the field resolver may return any of a number of different concrete object types, and Lacinia has no way to determine which; this information must be *explicitly provided*.

CHAPTER 6

Enums

GraphQL Spec

Read about enums.

GraphQL supports enumerated types, types whose value is limited to a explicit list.

```
{:enums
 {:episode
  {:description "The episodes of the original Star Wars trilogy."
   :values [:NEWHOPE :EMPIRE :JEDI]}}}
```

It is allowed to define enum values as either strings, keywords, or symbols. Internally, the enum values are converted to keywords.

Enum values must be unique, otherwise an exception is thrown when compiling the schema.

Enum values must be GraphQL Names: they may contain only letters, numbers, and underscores.

Enums *are* case sensitive; by convention they are in all upper-case.

When an enum type is used as an argument, the value provided to the field resolver function will be a keyword, regardless of whether the enum values were defined using strings, keywords, or symbols.

Field resolvers are required to return a keyword, and that keyword must match one of the values in the enum.

As with many other elements in GraphQL, a description may be provided for the enum (for use with *Introspection*).

To provide a description for individual enum values, a different form must be used:

```
{:enums
 {:episode
  {:description "The episodes of the original Star Wars trilogy."
   :values [{:enum-value :NEWHOPE :description "The first one you saw."}
            {:enum-value :EMPIRE :description "The good one."}
            {:enum-value :JEDI :description "The one with the killer teddy bears."}]}}
↪}
```

(continues on next page)

(continued from previous page)

The `:description` key is optional.

You may include the `:deprecation` key used to mark a single value as *deprecated*.

You may mix-and-match the two forms.

CHAPTER 7

Unions

A union type is a type that may be any of a list of possible objects.

GraphQL Spec

Read about unions.

A union is a type defined in terms of different objects:

```
{:objects
 {:person
  {:fields {:name {:type String}
            :age {:type Int}}}

  :photo
  {:fields {:imageURL {:type String}
            :title {:type String}
            :height {:type Int}
            :width {:type Int}}}}

 :unions
 {:search_result
  {:members [:person :photo]}}

 :queries
 {:search
  {:type (list :search_result)
   :args {:term String}}}}
```

A union definition must include a `:members` key, a sequence of object types.

The above example identifies the `:search-result` type to be either a `:person` (with fields `:name` and `:age`), or a `:photo` (with fields `:imageURL`, `:title`, `:height`, and `:width`).

Unions must define at least one type; each member type must be an object type (they may not reference scalar types, interfaces, or other unions).

When a client makes a union request, they must use the fragment spread syntax to identify what is to be returned based on the runtime type of object:

```
{ search (term:"ewok") {
  ... on person { name }
  ... on photo { imageURL title }
}}
```

This breaks down what will be returned in the result map based on the type of the value produced by the `:search` query. Sometimes there will be a `:name` key in the result, and other times an `:image-url` and `:title` key. This may vary result by result even within a single request:

```
{:data
 {:search
  [{:name "Nik-Nik"}
   {:imageURL "http://www.telegraph.co.uk/content/dam/film/ewok-xlarge.jpg"
    :title "an Ewok in The Return of the Jedi"}
   ]}}
```

Tip: When a field or operation type is a union, the field resolver may return any of a number of different concrete object types, and Lacinia has no way to determine which; this information must be *explicitly provided*.

CHAPTER 8

Queries

GraphQL Spec

Read about operations.

Queries are responsible for generating the initial resolved values that will be picked apart to form the result map.

Other than that, queries are just the same as any other field. Queries have a type, and accept arguments.

Queries are defined using the `:queries` key of the schema.

```
{:queries
 {:hero
  {:type (non-null :character)
   :args {:episode {:type :episode}}}

  :human
  {:type (non-null :human)
   :args {:id {:type String
               :default-value "1001"}}}}}
```

Queries may also be defined as fields of the *root query object*.

The *field resolver* for a query is passed nil as the the value (the third parameter). Outside of this, the query field resolver is the same as any field resolver anywhere else.

In the GraphQL specification, it is noted that queries are idempotent; if the query document includes multiple queries, they are allowed to execute in *parallel*.

CHAPTER 9

Mutations

Mutations parallel *queries*, except that the root field resolvers may make changes to underlying data in addition to exposing data.

The *field resolver* for a mutation will, as with a query, be passed nil as its value argument (the third argument). A mutation is expected to perform some state changing operation, then return a value that indicates the new state; this value will be recursively resolved and selected, just as with a query.

Mutations are defined in the schema using the top-level `:mutations` key.

Mutations may also be defined as fields of the *root mutation object*.

When a single query includes more than one mutation, the mutations *must* execute in the client-specified order. This is different from queries, which allow for each root query to run in *parallel*.

Typically, mutations are only allowed when the incoming request is explicitly an HTTP POST. However, that is beyond the scope of Lacinia (it doesn't know about the HTTP request, just the query string extracted from the HTTP request).

CHAPTER 10

Subscriptions

Subscriptions are GraphQL's approach to server-side push. The description is a bit abstract, as the specification keeps all options open on how subscriptions are to be implemented.

GraphQL Spec

Read about subscriptions.

With subscriptions, a client can establish a long-lived connection to a server, and will receive new data on the connection as it becomes available to the server.

Common use cases for subscriptions are updating a conversation page as new messages are added, updating a dashboard as interesting events about a system occur, or monitoring the progress of some long-lived process.

10.1 Overview

The specification discusses a *source stream* and a *response stream*.

Lacinia implements the source stream as a callback function. The response stream is largely the responsibility of the web tier.

- Lacinia invokes a streamer function once, to initialize the subscription stream.
- The streamer is provided with a source stream callback function; as new values are available they are passed to this callback.

  Typically, the streamer will create a thread, core.async process, or other long-lived construct to feed values to the source stream.
- Whenever the source stream callback is passed a value, Lacinia will execute the subscription as a query, which will generate a new response (with the standard `:data` and/or `:errors` keys).
- The response will be converted as necessary and streamed to the client, forming the response stream.

- The streamer must return a function that will be invoked to perform cleanup. This cleanup function typically stops whatever process was started earlier.

Subscriptions are operations, like queries or mutations. They are defined using the top-level `:subscriptions` key in the schema, or as fields of the *root subscription object*.

10.2 Streamer

The streamer is responsible for initiating and managing the the source stream.

The streamer is provided as the `:stream` key in the subscription definition.

```
{:objects
 {:LogEvent
  {:fields
   {:severity {:type String}
    :message {:type String}}}}

 :subscriptions
 {:logs
  {:type :LogEvent
   :args {:severity {:type String}}
   :stream :stream-logs}}}
```

Streamers parallel *field resolvers*, and a function, com.walmartlabs.lacinia.util/attach-streamers, is provided to replace keywords in the schema with actual functions.

A streamer is passed three values:

- The application context
- The field arguments
- The source stream callback

The first two are the same as a field resolver; the third is a function that accepts a single value.

The streamer should perform whatever operations are necessary for it to set up the stream of values; typically this is registering as a listener for updates to some form of publish/subscribe system.

As new values are published, the streamer must pass those values to the source stream callback.

Further, the streamer must return a function to clean up the stream when the subscription is terminated.

```
(defn log-message-streamer
  [context args source-stream]
  ;; Create an object for the subscription.
  (let [subscription (create-log-subscription)]
    (on-publish subscription
      (fn [log-event]
        (-> log-event :payload source-stream)))
    ;; Return a function to cleanup the subscription
    #(stop-log-subscription subscription)))
```

A lot of this example is hypothetical; it presumes `create-log-subscription` will return an value that can be used with `on-publish` and `stop-log-subscription`. A real implementation might use Clojure core.async, subscribe to a JMS queue, or an almost unbounded number of other options.

Regardless, the streamer provides the stream of source values, by making successive calls to the provided source stream callback function, and it provides a way to cleanup the subscription, by returning a cleanup function.

The subscription stays active until either the client closes the connection, or until `nil` is passed to the source stream callback.

In either case, the cleanup callback will then be invoked.

10.2.1 Timing

The source stream callback will return immediately. It must return nil.

The provided value will be used to generate a GraphQL result map, which will be streamed to the client. Typically, the result map will be generated asynchronously, on another thread.

Implementations of the source stream callback may set different guarantees on when or if values in the source stream are converted to responses in the response stream.

Likewise, when the subscription is closed (by either the client or by the streamer itself), the callback will be invoked asynchronously.

10.3 Resolver

Unlike a query or mutation, the *field resolver* for a subscription always starts with a specific value, provided by the streamer, via the source stream callback.

Because of this, the resolver is optional: if not provided, a default resolver is used, one that simply returns the source stream value.

However, it still makes sense to implement a resolver in some cases.

Both the resolver and the streamer receive the same map of arguments: it is reasonable that some may be used by the streamer (for example, to filter which values go into the source stream), and some by the resolver (to control the selections on the source value).

CHAPTER 11

Deprecation

Schemas grow and change over time. GraphQL values backwards compatibility quite highly, so changes to a schema are typically additive: introducing novel fields, types, and enum values.

However, when the implementation of a field is just *wrong*, it can be kept for compatibility, but deprecated.

Both *fields* and *enums* can include a `:deprecated` key. Remember that queries, mutations, and subscriptions are (under the covers) just fields, so they can be deprecated as well.

The value for the key can either be `true`, or a string description of the reason the field is deprecated. Typically, the description indicates an alternative field to use instead.

Deprecation does *not* affect execution of the field in any way; the deprecation flag and reason simply shows up in *introspection*.

When using the *Schema Definition Language*, elements may be marked with the `@deprecated` *directive*.

CHAPTER 12

Directives

Directives provide a way to describe additional options to the GraphQL executor.

Directives is a GraphQL term; in practice, directives are much like meta data in Clojure, or annotations in Java.

GraphQL Spec

Read about directives here and here.

Directives allow Lacinia to change the incoming query based on additional criteria. For example, we can use directives to include or skip a field if certain criteria are met.

Currently Lacinia supports just the two standard query directives: `@skip` and `@include`, but future versions may include more.

Warning: Directive support is currently in transition towards some support for custom directives.

12.1 Directives in Schema IDL

When using *GraphQL SDL Parsing*, the *directive* keyword allows new directives to be defined. Directive definitions can be defined for executable elements (such as a field selection in a query document), or for type system elements (such as an object or field definition in the schema).

Directives may also be defined in an EDN schema; the root `:directive-defs` element is a map of directives types to directive definition. A directive defintion defines the types of any arguments, as well as a set of locations.

```
{:directive-defs
 {:access
  {:locations #{:field-definition}
   :args {:role {:type (non-null String)}}}}}
```

(continues on next page)

(continued from previous page)

```
:queries
{:ultimate_answer
 {:type String
  :directives [{:directive-type :access
                :directive-args {:role "deep-thought"}}]}}}
```

This defines a field definition directive, `@access`, and applies it to the `ultimate_answer` query field.

12.2 Directive Validation

Directives are validated:

- Directives may have arguments, and the argument types must be rooted in known *scalar* types.
- Directives may be placed on schema elements (objects and input objects, unions, enums and enum values, fields, etc.). Directives placed on an element are verified to be applicable to the location.

> **Warning:** Directive support is evolving quickly; full support for directives, including argument type validation, is forthcoming, as is an API to identify schema and executable directives.
>
> The goal of the current stage is to support parsing of SDL schemas that include directive definitions and directives on elements.
>
> *Introspection* hasn't caught up to to these changes; custom directives are not identified, nor are directives on elements.

12.3 @deprecated directive

The `@deprecated` directive is supported. This enables *Deprecation* of object fields and enum values.

CHAPTER 13

Field Resolvers

Field resolvers are how Lacinia goes beyond data modelling to actually providing access to data.

GraphQL Spec

Read about value resolution.

Field resolvers are attached to fields, but also to top-level query and mutation operations. It is only inside field resolvers that a Lacinia application can connect to a database or an external system: field resolvers are where the data actually *comes* from.

In essence, the top-level operations perform the initial work in a request, getting the root object (or collection of objects).

Field resolvers in nested fields are responsible for extracting and transforming that root data. In some cases, a field resolver may need to perform additional queries against a back-end data store.

13.1 Overview

Each operation (query, mutation, or subscription) will have a root field resolver. Every field inside the operation or other object will have a field resolver: if an explicit one is not provided, Lacinia creates a default one.

A field resolver's responsibility is to *resolve* a value; in the simplest case, this is accomplished by just returning the value.

As you might guess, the processing of queries into result map data is quite recursive. The initial operation's field resolver is passed nil as the container resolved value.

The root field resolver will return a map[1] ; as directed by the client's query, the fields of this object will be selected and the top-level object passed to to the field resolvers for the fields in the selection.

[1] Or, in practice, a sequence of maps. In theory, an operation type could be a scalar, but use cases for this are rare.

This continues down, where at each layer of nested fields in the query, the containing field's resolved value is passed to each field's resolver function, along with the global context, and the arguments specific to that field.

The rules of field resolvers:

- A operation will resolve to a map of keys and values (or resolve to a sequence of such maps). The fields requested in the client's query will be used to resolve nested selections.
- Each field is passed its containing field's resolved value. It then returns a resolved value, which itself may be passed to its sub-fields.

Tip: It is possible to *preview nested selections* in a field resolver, which can be used to implement some important optimizations.

Meanwhile, the selected data from the resolved value is added to the result map.

If the value is a scalar type, it is added as-is.

Otherwise, the value is a structured type, and the query **must** provide nested selections.

13.1.1 Field Resolver Arguments

A field resolver is passed three arguments:

- The application context.
- The field's arguments.
- The containing field's resolved value.

Application Context

The application context is a map passed to the field resolver. It allows some global state to be passed down into field resolvers; the context is initially provided to com.walmartlabs.lacinia/execute The initial application context may even be nil.

Many resolvers can simply ignore the context.

Warning: Lacinia will frequently add its own keys to the context; these will be namespaced keywords. Please do not attempt to make use of these keys unless they are explicitly documented. Undocumented keys are not part of the Lacinia API and are subject to change without notice.

Field Arguments

This is a map of arguments provided in the query. The arguments map has keyword keys; the value types are as determined by definition of the field argument.

If the argument value is expressed as a query variable, the variable will be resolved to a simple value when the field resolver is invoked.

Container's Resolved Value

As mentioned above, the execution of a query is highly recursive. The operation, as specified in the query document, executes first; its resolver is passed nil for the container resolved value.

The operation's resolved value is passed to the field resolver for each field nested in the operation.

For scalar types, the field resolver can simply return the selected value.

For structured types, the field resolver returns a resolved value; the query *must* contain nested selections. These selections will trigger further fields, whose resolvers will be passed the resolved value.

For example, you might have a `:lineItem` query of type `:LineItem`, and LineItem might include a field, `:product` of type `:Product`. A query `{lineItem(id:"12345") { product }}` is not valid: it is not possible to return a Product directly, you **must** select specific fields within Product: `{lineItem(id:"12345") { product { name upc price }}}`.

Tip: Generally, we expect the individual values to be Clojure maps (or records). Lacinia supports *other types*, though that creates a bit of a burden on the developer to provide the necessary resolvers.

13.1.2 Resolving Collections

When an operation or field resolves as a collection, things are only slightly different.

The nested selections are applied to **each** resolved value in the collection.

13.1.3 Default Field Resolver

In the majority of cases, there is a direct mapping from a field name (in the schema) to a key of the resolved value.

When the `:resolve` key for a field is not specified, a default resolver is provided automatically; this default resolver simply expects the container resolved value to be a map containing a key that exactly matches the field name.

It is even possible to customize this default field resolver, as an option passed to com.walmartlabs.lacinia.schema/compile.

13.2 Attaching Resolvers

Schemas start as EDN files, which has the advantage that symbols do not have to be quoted (useful when using the `list` and `non-null` qualifiers on types). However, EDN is data, not code, which makes it nonsensical to defined field resolvers directly in the schema.

One option is to use `assoc-in` to attach resolvers after reading the EDN, but before invoking com.walmartlabs.lacinia.schema/compile. This can become quite cumbersome in practice.

Instead, the standard approach is to put keyword placeholders in the EDN file, and then use com.walmartlabs.lacinia.util/attach-resolvers, which walks the schema tree and makes the changes, replacing the keywords with the actual functions.

```
(ns org.example.schema
  (:require
    [clojure.edn :as edn]
    [clojure.java.io :as io]
```

(continues on next page)

(continued from previous page)

```
    [com.walmartlabs.lacinia.schema :as schema]
    [com.walmartlabs.lacinia.util :as util]
    [org.example.db :as db]))

(defn star-wars-schema
  []
  (-> (io/resource "star-wars-schema.edn")
      slurp
      edn/read-string
      (util/attach-resolvers {:hero db/resolve-hero
                              :human db/resolve-human
                              :droid db/resolve-droid
                              :friends db/resolve-friends})
      schema/compile))
```

The `attach-resolvers` step occurs **before** the schema is compiled.

13.2.1 Resolver Factories

There are often cases where many fields will need very similar field resolvers. A second resolver option exist for this case, where the schema references a *field resolver factory* rather than a field resolver itself.

In the schema, the value for the `:resolve` key is a vector of a keyword and then additional arguments:

```
{:queries
 {:hello {:type String
          :resolve [:literal "Hello World"]}}}
```

In the code, you must provide the field resolver factory:

```
(ns org.example.schema
  (:require
    [clojure.edn :as edn]
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.schema :as schema]
    [com.walmartlabs.lacinia.util :as util]))

(defn ^:private literal-factory
  [literal-value]
  (fn [context args value]
    literal-value))

(defn hello-schema
  []
  (-> (io/resource "hello.edn")
    slurp
    edn/read-string
    (util/attach-resolvers {:literal literal-factory})
    schema/compile))
```

The `attach-resolvers` function will see the `[:literal "Hello World"]` in the schema, and will invoke `literal-factory`, passing it the `"Hello World"` argument. `literal-factory` is responsible for returning the actual field resolver.

A field resolver factory may have any number of arguments.

Common uses for field resolver factories:

- Mapping GraphQL field names to Clojure hyphenated names
- Converting or formatting a raw value into a selected value
- Accessing a deeply nested value in a structure

13.3 Explicit Types

For structured types, Lacinia needs to know what type of data is returned by the field resolver, so that it can, as necessary, process query fragments.

When the type of field is a concrete object type, Lacinia automatically tags the value with the schema type.

When the type of a field is an interface or union, it is necessary for the field resolver to explicitly tag the value with its object type.

13.3.1 Using tag-with-type

The function com.walmartlabs.lacinia.schema/tag-with-type exists for this purpose. The tag value is a keyword matching an object definition.

When a field returns a list of an interface, or a list of a union, then each individual resolved value must be tagged with its concrete type. It is allowed and expected that different values in the collection will have different concrete types.

Generally, type tagging is just metadata added to a map (or Clojure record type). However, Lacinia supports tagging of arbitrary objects that don't support Clojure metadata . . . but `tag-with-type` will return a wrapper type in that case. When using Java types, make sure that `tag-with-type` is the last thing a field resolver does.

13.3.2 Using record types

As an alternative to `type-with-tag`, it is possible to associate an object with a Java class; typically this is a record type created using `defrecord`.

The `:tag` key of the object definition must be set to the the class name (as a symbol).

```
{:objects
 {:business
  {:fields
   {:id {:type :ID}
    :name {:type :String}}
   :tag com.example.data.Business}

  :employee
  {:fields
   {:id {:type :ID}
    :employer {:type :business}
    :given_name {:type :String}
    :family_name {:type :String}}
   :tag com.example.data.Employee}}

 :unions
 {:searchable
  {:members [:business :employee]}}

 :queries
```

(continues on next page)

(continued from previous page)

```
{:businesses
 {:type (list :business)
  :resolve :query/businesses}

 :search
 {:type (list :searchable)
  :resolve :query/search}}}
```

This only works if the field resolver functions return the corresponding record types, rather than ordinary Clojure maps. In the above example, the field resolvers would need to invoke the `map->Business` or `map->Employee` constructor functions as appropriate.

Tip: The `:tag` value is a Java class name, not a namespaced Clojure name. That means no slash character, and dashes in the namespace must be converted to underscores.

13.3.3 Container type

When a field resolver is invoked, the context value for key `:com.walmartlabs.lacinia/container-type-name` will be the name of the concrete type (a keyword) for the resolved value passed into the resolver. This will be nil for top-level operations.

When the type of the containing field is a union or interface, this value will be the specific concrete object type for the actual resolved value.

13.4 Exceptions

Field resolvers are responsible for catching any exceptions that occur.

Uncaught exceptions will bubble up out of Lacinia code entirely.

This is not desirable: better to return a partial result along with errors.

Field resolvers should catch exceptions and use *ResolverResults* to communicate errors back to Lacinia for inclusion in the `:errors` key of the result.

Failure to catch exceptions is even more damaging when using *async field resolvers*.

13.5 Using ResolverResult

In the simplest case, a field resolver will do its job, resolving a value, simply by returning the value.

However, there are several other scenarios:

- There may be errors associated with the field which must be communicated back to Lacinia (for the top-level `:errors` key in the result map)
- A field resolver may want to introduce *changes into the application context* as a way of communicating to deeply nested field resolvers
- A field resolver may operate *asynchronously*, and want to return a promise for data that will be available in the future

GraphQL Spec

Read about errors.

Field resolvers should not throw exceptions; instead, if there is a problem generating the resolved value, they should use the com.walmartlabs.lacinia.resolve/resolve-as function to return a ResolverResult value.

When using `resolve-as`, you may pass the error map as the second parameter (which is optional). You may pass a single error map, or a seq of error maps. This first parameter is the resolved value, which may be `nil`.

Why not just throw an exception?

Exceptions are a terrible way to deal with control flow issues, even in the presence of actual failures. More importantly, the ResolverResult approach allows more than a single error, and works well with Lacinia's *asynchronous processing features*.

Errors will be exposed as the top-level `:errors` key of the execution result.

Error maps must contain at a minimum a `:message` key with a value of type String.

You may specify other keys and values as you wish, but these values will be part of the ultimate result map, so they should be both concise and safe for the transport medium. Generally, this means not to include values that can't be converted into JSON values.

In the result map, error maps are transformed; they contain the `:message` key, as well as `:locations`, `:path`, and (sometimes) `:extensions`.

```
{:data {:hero {:friends [nil nil nil]}}
 :errors [{:message "Non-nullable field was null."
           :locations [{:line 1
                        :column 20}]
           :path [:hero :friends :arch_enemy]}]}
```

The `:locations` key identifies where in the query document, as a line and column address, the error occured. It's value is an array (normally, a single value) of location maps; each location map has `:line` and `:column` keys.

The `:path` associates the error with a location in the result data; this is seq of the names of fields (or aliases for fields). Some elements of the path may be numeric indices into sequences, for fields of type list. These indices are zero based.

Any additional keys in the error map are collected into the `:extensions` key (which is only present when the error map has such keys).

The order in which errors appear in the `:errors` key of the result map is not specified; however, Lacinia does remove duplicate errors.

13.5.1 Tagging Resolvers

If you write a function that *always* returns a ResolverResult, you should set the tag of the function to be com.walmartlabs.lacinia.resolve/ResolverResult. Doing so enables an optimization inside Lacinia - it can skip the code that checks to see if the function did in fact return a ResolverResult and wrap it in a ResolverResult if not.

Unfortunately, because of how Clojure handles function meta-data, you need to write your function as follows:

```
(require '[com.walmartlabs.lacinia.resolve :refer [resolve-as ResolverResult]])

(def tagged-resolver
  ^ResolverResult (fn [context args value]
                    (resolve-as ...)))
```

This places the type tag on the function, not on the symbol (as normally happens with `defn`).

It doesn't matter whether the function invokes `resolve-as` or `resolve-promise`, but returning nil or a bare value from a field resolver tagged with ResolverResult will cause runtime exceptions, so be careful.

13.6 FieldResolver Protocol

In the majority of cases, a field resolver is simply a function that accepts the three parameters: context, args, and value.

However, when structuring large systems using Component (or any similar approach), this can be inconvienient, as it does not make it possible to structure field resolvers as components.

The com.walmartlabs.lacinia.resolve/FieldResolver` protocol addresses this: it defines a single method, ``resolve-value`. This method is the analog of an ordinary field resolver function.

The support for this protocol is baked directly into com.walmartlabs.lacinia.schema/compile.

Just like field resolver functions, FieldResolver instances can resolve a value directly by returning it, or can return a ResolverResult.

13.7 Asynchronous Field Resolvers

Lacinia supports asynchronous field resolvers: resolvers that run in parallel within a single request.

This can be very desirable: different fields within the same query may operate on different databases or other backend data sources, for example.

Alternately, a single request may invoke multiple top-level operations which, again, can execute in parallel.

It's very easy to convert a normal synchronous field resolver into an asynchronous field resolver: Instead of returning a normal value, an asynchronous field resolver returns a special kind of *ResolverResult*, a `ResolverResultPromise`.

Such a promise is created by the :api:`resolve/resolve-promise function.

The field resolver function returns immediately, but will typically perform some work in a background thread. When the resolved value is ready, the `deliver!` method can be invoked on the promise.

```
(require
  '[com.walmartlabs.lacinia.resolve :as resolve])

(defn ^:private get-user
  [connection user-id]
  ...)

(defn resolve-user
  [context args _]
  (let [{:keys [id]} args
        {:keys [connection]} context
        result (resolve/resolve-promise)]
```

(continues on next page)

(continued from previous page)

```
    (.start (Thread.
              #(try
                 (resolve/deliver! result (get-user connection id))
                 (catch Throwable t
                   (resolve/deliver! result nil
                     {:message (str "Exception: " (.getMessage t))}))))

    result))
```

The promise is created and returned from the field resolver function. In addition, as a side effect, a thread is started to perform some work. When the work is complete, the `deliver!` method on the promise will inform Lacinia, at which point Lacinia can start to execute selections on the resolved value (in this example, the user data).

On normal queries, Lacinia will execute as much as it can in parallel. This is controlled by how many of your field resolvers return a promise rather than a direct result.

Despite the order of execution, Lacinia ensures that the order of keys in the result map matches the order in the query.

GraphQL Spec

Read about execution.

For mutations, the top-level operations execute serially. That is, Lacinia will execute one top-level operation entirely before starting the next top-level operation.

13.7.1 Timeouts

Lacinia does not enforce any timeouts on the field resolver functions, or the promises they return. If a field resolver fails to `deliver!` to a promise, then Lacinia will block, indefinitely.

It's quite reasonable for a field resolver to enforce some kind of timeout on its own, and deliver nil and an error message when a timeout occurs.

13.7.2 Exceptions

Uncaught exceptions in an asynchonous resolver are especially problematic, as it means that ResolverResultPromises are never delivered.

In the example above, any thrown exception is converted to an *error map*.

Warning: Not catching exceptions will lead to promises that are never delivered and that will cause Lacinia to block indefinitely.

13.7.3 Thread Pools

By default, calls to `deliver!` invoke the callback (provided to `on-deliver!`) in the same thread. This is not always desirable; for example, when using Clojure core.async, this can result in considerable processing occuring within a thread from the dispatch thread pool (the one used for `go` blocks). There are typically only eight threads in that pool, so a callback that does a lot of processing (or blocks due to I/O operations) can result in a damaging impact on overall server throughput.

To address this, an optional executor can be provided, via the dynamic com.walmartlabs.lacinia.resolve/*callback-executor* var. When a ResolverResultPromise is delivered, the executor (if non-nil) will be used to execute the callback; Java thread pools implement this interface.

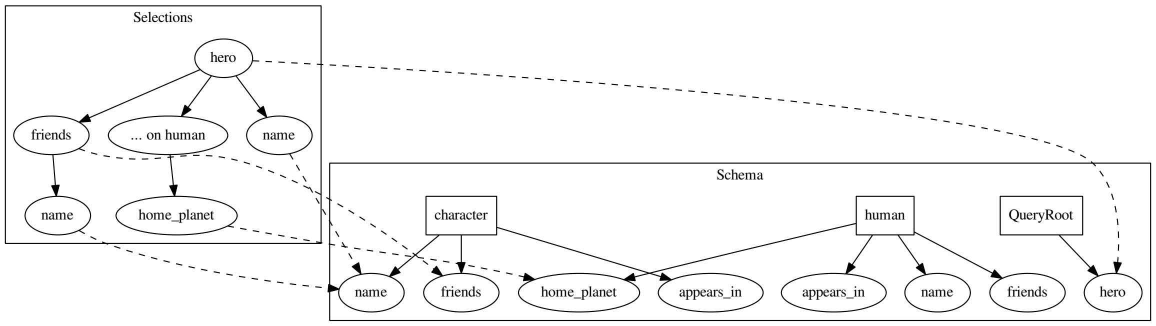
13.8 Nested Selections

To review: executing a GraphQL query is highly recursive, where a field resolver will be responsible for obtaining data from an external resource or data store, and nested fields will refine the root data and make selections.

Consider a simple GraphQL query:

```
{
  hero
  {
    name
    friends { name }
    ... on human {
      home_planet
    }
  }
}
```

The query can be visualized as a tree of selections:



Nodes in the selections tree relate to fields in the schema. Remember that the type of query `hero` is the interface type `character` (and so can be either a `human` or a `droid`).

In execution order, resolution occurs top to bottom, so the `hero` selection occurs first, then (potentially *in parallel*) `friends`, `home_planet`, and (hero) `name`. These last two are leaf nodes, because they are scalar values. The list of `characters` (from the `friends` field) then has its `name` field selected. The result map then constructs from bottom to top.

13.8.1 Previewing Selections

A field resolver can "preview" what fields will be selected below it in the selections tree. This is a tool frequently used to optimize data retrieval operations.

As an example, lets assume a starting configuration where the `hero` field resolver fetches just the basic data for a hero (`id`, `name`, `home_planet`, etc.) and the `friends` resolver does a second query against the database to fetch the list of friends.

That's two database queries. Perhaps we can optimize things by getting rid of the `friends` resolver, and doing a join to fetch the hero and friends at the same time. The `hero` resolver can just ensures there's a `:friends` key in the map (with the fetched friend values), and the default field resolver for the `friends` field will access it.

That's simpler, but costly when `friends` is not part of the query.

The function com.walmartlabs.lacinia.executor/selects-field? can help here:

```
(require
  '[com.walmartlabs.lacinia.executor :as executor])

(defn resolve-hero
  [context args _]
  (if (executor/selects-field? context :character/friends)
    (fetch-hero-with-friends args)
    (fetch-hero args)))
```

Here, inside the application context (provided to the `resolve-hero` function) is information about the selections, and `selects-field?` can determine if a particular field appears *anywhere* below `hero` in the selection tree.

`selects-field?` identifies fields even inside nested or named fragments, `(executor/selects-field? context :human/home_planet)` would return true.

It is also possible to get *all* the fields that will be selected, using `selections-seq`. This a lazy, breadth-first navigation of all fields in the selection tree.

In the sequence of field names, any fragments are collapsed into their containing fields.

This level of detail may be insufficient, in which case the function `selections-tree` can be used.

This function builds a recursive structure that identifies the entire tree structure. For the above query, it would return the following structure:

```
{:character/friends {:selections
                      {:character/name nil}}
 :human/home_planet nil
 :character/name nil}
```

This shows, for example, that `:character/name` is used in two different ways (inside the `hero` query itself, and within the `friends` field).

For fields with arguments, an `:args` key is present, with the exact values which will be supplied to the the nested field's resolver.

Fields with neither arguments nor sub-selections are represented as nil.

13.9 Application Context

The application context passed to your field resolvers is normally set by the initial call to com.walmartlabs.lacinia/execute. Lacinia uses the context for its own book-keeping (the keys it places into the map are namespaced to avoid collisions) but otherwise the same map is passed to all field resolvers.

In specific cases, it is useful to allow a field resolver to modify the application context, with the change exposed just to the fields nested below, but to any depth.

For example, in this query:

```
{
  products(search: "fuzzy") {
    category {
      name
      product {
        upc
        name
        highlighted_name
      }
    }
}
```

Here, the search term is provided to the `products` field, but is again needed by the `highlighted_name` field, to highlight the parts of the name that match the search term.

The resolver for the `products` field can communicate this information "down tree" to the resolver for the `highlighted_name` field, by using the com.walmartlabs.lacinia.resolve/with-context function.

```
(require '[com.walmartlabs.lacina.resolve :as resolve])

(defn resolve-products
  [_ args _]
  (let [search-term (:search args)]
    (-> (perform-product-search args)
        (resolve/with-context {::search-term search-term}))))

(defn resolve-highlighted-name
  [context _ product]
  (let [{:keys [::search-term]} context]
    (-> product :name (add-highlight search-term))))
```

The map provided to `with-context` will be merged into the application context before any nested resolvers are invoked. In this way, the new key, `::search-term`, is only present in the context for field resolvers below `products` field.

Some field resolvers returns lists of values; the entire list can be wrapped in this way **OR** individual values within the list may be wrapped.

Tip: Remember that query execution is top to bottom, then the final result map is assembled from the leaves back up to the roots.

When using lacinia-pedestal, the default behavior is to capture the Ring request map and supply it in the application context under the `:request` key.

13.10 Extensions

Lacinia makes it possible to add extension data to the result map from within field resolvers. This extension data is exposed as the `:extensions` key on the result map (alongside the more familiar `:data` and `:errors` keys).

Extensions key?

GraphQL supports a third result key, `extensions`, as described in the spec. It exists just for this kind of extra information in the response.

The GraphQL specification allow for extensions but leaves them entirely up to the application to define and use - there's no validation at all. One example of extension data is *timing information* that Lacinia can optionally include in the result.

More general extension data is introduced using modifier functions defined in the `com.walmartlabs.lacinia.resolve` namespace.

13.10.1 Extension Data

The `with-extensions` function is used to introduce data into the result. `with-extension` is provided with a function, and additional optional arguments, and is used to modify the extension data to a new state.

A hypothetical example; perhaps you would like to identify the total amount of time spent performing database access, and return a total in the `:total-db-access-ms` extension key.

You might instrument each resolver that accesses the database to calculate the elapsed time.

```clojure
(require '[com.walmartlabs.lacinia.resolve :refer [with-extensions]])

(defn resolve-user
  [context args value]
  (let [start-ms (System/currentTimeMillis)
        user-data (get-user-data (:id args))
        elapsed-ms (- (System/currentTimeMillis) start-ms)]
     (with-extensions user-data
       update :total-db-access-ms (fnil + 0) elapsed-ms)))
```

The call to `with-extensions` adds the elapsed time for this call to the key (the `fnil` function lets the + function treat nil as 0).

This data is exposed in the final result map:

```clojure
{:data
 {:user
  {:id "thx1138"
   :name "Thex"}}
 :extensions
 {:total-db-access-ms 3}}
```

You should be aware that field resolvers may run in an unpredictable order, especially when *asynchronous field resolvers* are involved. Complex update logic may be problematic if one field resolver expects to modify extension data introduced by a different field resolver. Sticking with `assoc-in` or `update-in` is a good bet.

13.10.2 Warnings

The `with-error` modifier function adds one or more errors to a result; in general, errors are serious - Lacinia adds errors when it can't parse the GraphQL document, for example.

Lacinia adds a less dramatic level, a warning, via the `with-warning` modifier. `with-warning` adds an error map to the `:warnings` map stored in the `:extensions` key.

What a client does with errors and warnings is up to the application. In general, errors should indicate a failure of the overall request, and an interactive client might display an alert dialog related to the request and response.

An interactive client may present warnings differently, perhaps adding an icon to the user view to indicate that there were non-fatal issues executing the query.

13.11 Resolver Timing

Caution: This feature is experimental, and subject to change. Feedback is encouraged!

When scaling a service, it is invaluable to know where queries are spending their execution time; Lacinia can help you here; when enabled, Lacinia will collect start and finish timestamps, and elapsed time, for each resolver function that is invoked during execution of the query.

Timing collection is enabled using the key, `:com.walmartlabs.lacinia/enable-timing?`, in the application context:

```
(require '[com.walmartlabs.lacinia :as lacinia])

(lacinia/execute
  star-wars-schema
  "{
  luke: human(id: \"1000\") { friends { name }}
  leia: human(id: \"1003\") { name }
}" nil
  {::lacinia/enable-timing? true})
=>
{:data {:luke {:friends [{:name "Han Solo"} {:name "Leia Organa"} {:name "C-3PO"}
↪{:name "R2-D2"}]},
        :leia {:name "Leia Organa"}},
 :extensions
 {:timing
  {:human
   {:execution/timings [{:start 1493245557504, :finish 1493245557504, :elapsed 0}
                        {:start 1493245557507, :finish 1493245557507, :elapsed 0}],
    :friends
    {:execution/timings [{:start 1493245557505, :finish 1493245557505, :elapsed 0}]}}}
↪}}
```

Extensions key?

GraphQL supports a third result key, `extensions`, as described in the spec. It exists just for this kind of extra information in the response.

Timings are returned in a tree structure below the `:extensions` key of the result. The tree structure reflects the structure of the *query* (not the schema).

We can see here that the `human` query operation's resolver was invoked twice, and the `friends` field's resolver was invoked just once. Only explicitly added resolve functions are timed; default resolve functions are trivial and not included.

For each invocation, values are identified for the start and finish time (in standard format, milliseconds since the epoch), and elapsed time (also in milliseconds).

Since our sample data is all in-memory, execution is instantaneous, so the elapsed time is 0. In real applications, making requests to a database or accessing other resources, there would be some amount of elapsed time.

When the field resolvers are *asynchronous*, you'll often see start and finish times for each invocation overlap. The finish time is calculated not when the resolver function returns, but when it produces a value (that is, when the ResolverResult is realized).

13.12 Examples

13.12.1 Formatting a Date

This is a common scenario: you have a Date or Timestamp value and it needs to be in a specific format in the result map.

In this example, the field resolver will extract the key from the container's resolved value, and format it:

```
(fn [context args resolved-value]
    (->> resolved-value
         :updated-at
         (format "%tm-%<td-%<tY")))
```

This example is tied to a specific key (`:updated-at`) and a specific format. A *resolver factory* could be used to make this a more general pattern.

13.12.2 Accessing a Java Instance Method

In some cases, you may find yourself exposing JavaBeans as schema objects. This fights against the grain of Lacinia, which expects schema objects to be Clojure maps.

It would be tedious to write a custom field resolver function for each and every Java instance method that needs to be invoked. Instead, we can use a *resolver factory*.

```
(defn ^:private make-accessor
  [method-sym]
  (let [method-name (name method-sym)
        arg-types (make-array Class 0)
        args (make-array Object 0)]
    (fn [value]
      (let [c (class value)
            method (.getMethod c method-name arg-types)]
        (.invoke method value args)))))

(defn resolve-method
  [method-sym]
  (let [f (make-accessor method-sym)]
    (fn [_ _ value]
      (f value))))
```

This won't be the most efficient approach, since it has to lookup a method on each use and then invoke that method using Java reflection, but may be suitable for light use, or as the basis for a more efficient implementation.

Note: More examples forthcoming.

CHAPTER 14

Input Objects

GraphQL Spec

Read about input objects.

In some cases, it is desirable for a query to include arguments with more complex data than a single value. A typical example would be passing a bundle of values as part of a *mutation* operation (rather than an unmanageable number of individual field arguments).

Input objects are defined like ordinary objects, with a number of restrictions:

- Field types are limited to scalars, enums, and input objects (or list and non-null wrappers around those types)
- There are no field resolvers for input objects; these are values passed in their entirety from the client to the server
- Input objects do not implement interfaces

Input objects are defined as keys of the top-level `:input-objects` key in the schema.

CHAPTER 15

Custom Scalars

Defining custom scalars may allow users to better model their domain.

GraphQL Spec

Read about custom scalars.

To define a custom scalar, you must provide implementations, in your schema, for two transforming callback functions:

parse parses query arguments and coerces them into their scalar types according to the schema.

serialize serializes a scalar to a value that will be in the result of the query or mutation.

In other words, a scalar is serialized to another type, typically a string, as part of executing a query and generating results. In some cases, such as field arguments, the reverse may be true: the client will provide the serialized version of a value, and the parse operation will convert it back to the appropriate type.

Both a parse function and a serialize function must be defined for each scalar type. These callback functions are passed a value and peform necessary coercions and validations.

Neither callback is ever passed `nil`.

Dates are a common example of this, as dates are not supported directly in JSON, but are typically encoded as some form of string.

Here is an example that defines and uses a custom `:Date` scalar type:

```
(require
  '[clojure.spec.alpha :as s]
  '[com.walmartlabs.lacinia :as g]
  '[com.walmartlabs.lacinia.schema :as schema])

(import
  java.text.SimpleDateFormat
  java.util.Date)
```

(continues on next page)

(continued from previous page)

```
(def date-formatter
  "Used by custom scalar :Date."
  (SimpleDateFormat. "yyyy-MM-dd"))

(def schema
  (schema/compile
    {:scalars
     {:Date
      {:parse #(when (string? %)
                  (try
                    (.parse date-formatter %)
                    (catch Throwable _
                      nil)))
       :serialize #(try
                     (.format date-formatter %)
                     (catch Throwable _
                       nil))}}

     :queries
     {:today
      {:type :Date
       :resolve (fn [ctx args v] (Date.))}
      :echo
      {:type :Date
       :args {:input {:type :Date}}
       :resolve (fn [ctx {:keys [input]} v] input)}}}))

(g/execute schema "{today}" nil nil)
=> {:data {:today "2018-11-21"}}
(g/execute schema "{ echo(input: \"2018-11-22\") }" nil nil)
=> {:data {:echo "2018-11-22"}}
(g/execute schema "{ echo(input: \"thanksgiving\") }" nil nil)
=>
{:errors [{:message "Exception applying arguments to field `echo': For argument `input
↪', unable to convert \"thanksgiving\" to scalar type `Date'.",
           :locations [{:line 1, :column 3}],
           :extensions {:field :echo, :argument :input, :value "thanksgiving", :type-
↪name :Date}}]}
```

Warning: This is just an simplified example used to illustrate the broad strokes. It is not thread safe, because the `SimpleDateFormat` class is not thread safe.

15.1 Parsing

The parse callback is provided a value the originates in either the GraphQL query document, or in the variables map.

The values passed to the callback may be strings, numbers, or even maps (with keyword keys). It is expected that the parse function will do any necessary conversions and validations, or indicate an invalid value.

15.2 Serializing

Serializing is often the same as parsing (in fact, it is not uncommon to use one function for both roles).

The serialize callback is passed whatever value was selected from a field and cooerces it to an appropriate value for the response (typically, either a string, or another value that can be encoded into JSON).

15.3 Handling Invalid Values

Especially when parsing an input string into a value, there can be problems, especially included invalid data sent in the request.

The parse and serialize callback functions should **not** throw an exception; instead, to indicate a parsing or serializing problem, they may return nil, or create a coercion failure result by invoking the function com.walmartlabs.lacinia.schema/coercion-failure.

Returning nil results in a generic error message (as in the example above). By creating a coercion failure result, a specific error message can be provided, as well as additional data.

In either case, the parse or serialization failure will result in an error being added to the result map.

15.4 Scalars and Variables

GraphQL Spec

Read about variables.

When using variables, the scalar parser will be provided not with a string per-se, but with a Clojure value: a native Long, Double, or Boolean. In this case, the parser is, not so much parsing, as validating and transforming.

For example, the built-in `Int` parser handles strings and all kinds of numbers (including non-integers). It also ensures that `Int` values are, as identified in the specification, limited to signed 32 bit numbers.

15.5 Attaching Scalar Transformers

As with field resolvers, the pair of transformers for each scalar have no place in an EDN file as they are functions. Instead, the transformers can be attached after reading the schema from an EDN file, using the function com.walmartlabs.lacinia.util/attach-scalar-transformers.

CHAPTER 16

Root Object Names

Top-level query, mutation, and subscription operations are represented in Lacinia as fields on special objects. These objects are the “operation root objects”. The default names of these objects are `QueryRoot`, `MutationRoot`, and `SubscriptionRoot`.

When compiling an input schema, these objects will be created if they do not already exist.

The `:roots` key of the input schema is used to override the default names. Inside `:roots`, you may specify keys `:query`, `:mutation`, or `:subscription`, with the value being the name of the corresponding object.

If the objects already exist, then any fields on the objects are automatically available operations. In the larger GraphQL world (beyond Lacinia), this is the typical way of defining operations.

Beyond that, the operations from the `:queries`, etc. map of the input schema will be merged into the fields of the corresponding root object.

Name collisions are not allowed; a schema compile exception is thrown if an operation (via `:queries`, etc.) conflicts with a field of the corresponding root object.

16.1 Unions

It is allowed for the root type to be a union. This can be a handy way to organize many different operations.

In this case, Lacinia creates a new object and merges the fields of the members of the union, along with any operations from the input schema.

Again, the field names must not conflict.

The new object becomes the root operation object.

16.2 Internal Names

In some cases, you may see error messages that refer to fields in the `:__Queries`, `:__Mutations`, or `:__Subscriptions` objects. Internally, Lacinia constructs these objects, building `:__Queries` from the

`:queries` map (and so forth). The fields may then be merged from these internal objects into the actual operation root objects.

CHAPTER 17

Introspection

Introspection is a key part of GraphQL: the schema is self-describing.

GraphQL Spec

Read about introspection.

Introspection data is derived directly from the schema. Often, a `:description` key is added to the schema to provide additional help.

Introspection is necessary to support the in-browser graphiql IDE.

Introspection can also be leveraged by smart clients, presumably custom in-browser or mobile applications, to help deal with schema evolution. A smart client can use introspection to determine if a particular field exists before including that field in a query request; this can help defuse the process of introducing a new field (in the server) at the same time as a new client that needs to make use of that field.

CHAPTER 18

clojure.spec

Lacinia makes use of clojure.spec; specifically, the arguments to com.walmartlabs.lacinia.schema/compile and com.walmartlabs.lacinia.parser.schema/parse-schema are *always* validated with spec.

This is useful, especially for `compile`, and the data structure passed in for compilation is complex and deeply nested.

However, the exceptions thrown by clojure.spec can be challenging to read.

The use of Expound is recommended; it does a much better job of formatting that wealth of data for a person to read.

For example, it omits all the extraneous detail, making it much easier to find where the problem exists:

```
-- Spec failed --------------------
  {:objects
   {:Henry
    {:fields
     {:higgins
      {:type ...,
       :resolve ...,
       :deprecated 7.0}}}}}
                   ^^^
should satisfy
  true?
or
  string?
```

Further, Lacinia includes an extra namespace, not loaded by default: `com.walmartlabs.lacinia.expound`. This namespace simply defines spec messages for some of the trickier specs defined by Lacinia.

CHAPTER 19

GraphQL SDL Parsing

GraphQL Spec

Read about the Schema Definition Language.

As noted in the *overview*, Lacinia schemas are represented as Clojure data. However, Lacinia also contains a facility to transform schemas written in the GraphQL Schema Definition Language into the form usable by Lacinia. This is exposed by the function com.walmartlabs.lacinia.parser.schema/parse-schema.

The Lacinia schema definition includes things which are not available in the SDL, such as resolvers, subscription streamers, custom scalar parsers/serializers and documentation. To add these, `parse-schema` has two arguments: a string containing the SDL schema definition, and a map of resolvers, streamers, scalar functions and documentation to attach to the schema:

```
{:resolvers {:field-name resolver-fn}
 :streamers {:field-name stream-fn}
 :scalars {:scalar-name {:parse parse-spec
                         :serialize serialize-spec}}
 :documentation {:type-name doc-str
                 :type-name/field-name doc-str
                 :type-name/field-name.arg-name doc-str}}
```

19.1 Example

Listing 1: *schema.txt*

```
enum episode {
  NEWHOPE
  EMPIRE
  JEDI
```

(continues on next page)

(continued from previous page)

```
}

type Character {
  name: String!
  episodes: [episode]
}

input CharacterArg {
  name: String!
  episodes: [episode]!
}

type Query {
  find_all_in_episode(episode: episode!) : [Character]
}

type Mutation {
  add_character(character: CharacterArg!) : Boolean
}

schema {
  query: Query
  mutation: Mutation
}
```

```
(parse-schema (slurp (clojure.java.io/resource "schema.txt"))
              {:resolvers {:Query {:find_all_in_episode :find-all-in-episode}
                                   :Mutation {:add_character :add-character}}
               :documentation {:Character "A Star Wars character"
                               :Character/name "Character name"
                               :Query/find_all_in_episode "Find all characters in the␣
↪given episode"
                               :Query/find_all_in_episode.episode "Episode for which␣
↪to find characters."}})
```

Listing 2: *Return value of parse-schema*

```
{:objects
 {:Character {:fields
              {:name {:type (non-null String)
                      :description "Character name"}
               :episodes {:type (list :episode)}}
              :description "A Star Wars character"}
  :Query {:fields
          {:find_all_in_episode {:type (list :Character)
                                 :args
                                 {:episode
                                  {:type (non-null :episode)
                                   :description "Episode for which to find characters.
↪"}}
                                 :resolve :find-all-in-episode
                                 :description "Find all characters in the given␣
↪episode"}}}
  :Mutation {:fields
             {:add_character
              {:type Boolean
```

(continues on next page)

(continued from previous page)

```
              :args {:character {:type (non-null :CharacterArg)}}
              :resolve :add-character}}}}

:input-objects
{:CharacterArg {:fields
                {:name {:type (non-null String)}
                 :episodes {:type (non-null (list :episode))}}}}

:enums
{:episode {:values [{:enum-value :NEWHOPE}
                    {:enum-value :EMPIRE}
                    {:enum-value :JEDI}]}}

:roots
{:query :Query
 :mutation :Mutation}}
```

The `:documentation` key uses a naming convention on the keys which become paths into the Lacinia input schema. `:Character/name` applies to the `name` field of the `Character` object. `:Query/find_all_in_episode.episode` applies to the `episode` argument, inside the `find_all_in_episode` field of the `Query` object.

Tip: Attaching documentation this way is less necessary since release 0.29.0, which added support for embedded schema documentation.

Alternately, the documentation map can be parsed from a Markdown file using com.walmartlabs.lacinia.parser.docs/parse-docs.

The same key structure can be used to document input objects and interfaces.

Unions may be documented, but do not contain fields.

Enums may be documented, as well as Enum values (e.g., `:Episode/JEDI`).

As is normal with Schema Definition Language, the available queries, mutations, and subscriptions (not shown in this example) are defined on ordinary schema objects, and the `schema` element identifies which objects are used for which purposes.

The `:roots` map inside the Lacinia schema is equivalent to the `schema` element in the SDL.

Warning: Schema extensions are defined in the GraphQL specification, but not yet implemented.

CHAPTER 20

Sample Projects

boardgamegeek-graphql-proxy Howard Lewis Ship created this simple proxy to expose part of the BoardGameGeek database as GraphQL, using Lacinia.

It was used for examples in his Clojure/West 2017 talk: Power to the (Mobile) People: Clojure and GraphQL.

leaderboard-api A simple API to track details about games and high scores. Built on top of Compojure and PostgreSQL. See this blog post by the author.

CHAPTER 21

Clojure 1.9

Lacinia targets Clojure 1.9, and makes specific use of `clojure.spec`.

To use Lacinia with Clojure 1.8, modify your `project.clj` to include `clojure-future-spec`:

```
:dependencies [[org.clojure/clojure "1.8.0"]
               [com.walmartlabs/lacinia "x.y.z"]
               [clojure-future-spec ""]
               ...]
```

CHAPTER 22

Other Resources

- First Release Announcement

  A quick introduction to Lacinia, and goes into detail about the problems it solves for us at Walmart.
- A talk at Clojure/West 2017: *Power to the (Mobile) People: Clojure and GraphQL* is available on YouTube.
- A good introductory Blog Post by James Borden.
- A talk at Clojure/Conj 2017: *The Power of Lacinia & Hystrix in Production* is available on You Tube.
- Blog Post: The Case for Clojure and GraphQL: Replacing Django.
- Blog Post: Through the Looking Graph: Experiences adopting GraphQL on a Clojure/script project.

CHAPTER 23

Contributing to Lacinia

We hope to build a community around Lacinia and extensions and enhancements to it.

23.1 Licensing

Lacinia is licensed under the terms of the Apache Software License 2.0. Any contributions made by the community, including patches, pull requests, or any content provided in an issue, represents a transfer of intellectual property to Walmartlabs, for the sole purpose of maintaining and improving Lacinia.

23.2 Process

Our process is light: once a pull request (PR) comes in, core committers will review the the code and provide feedback.

CLA

When submitting a PR, you will need to sign the Contributor License Agreement (CLA) before the PR can be *reviewed or merged*. The CLA is quite straightforward, and simply assigns copyright and patent rights to Walmart for any contributed changes to Lacinia.

After at least two core committers provide the traditional feedback LGTM (looks good to me), we'll merge to master.

It is the submitter's responsibility to keep a PR up to date when it has conflicts with master.

Please do not change the version number in `project.clj`; the core committers will handle version number changes. Generally, we advance the version number immediately after a release.

Please close PRs that are not ready to merge, or need other refinements, and re-open when ready.

As Lacinia's community grows, we'll extend this process as needed . . . but we want to keep it light.

23.3 Issue Tracking

We currently use the GitHub issue tracker for Lacinia. We expect that to be sufficient for the meantime.

23.4 Coding Conventions

Please follow the existing code base.

We prefer `defn ^:private` to `defn-`.

Keep as much as possible private; the more public API there is, the more there is to support release on release.

Occasionally it is not reasonable to refactor a common implementation function out of a public namespace to a private namespace, in order to share the function between namespaces. In that case, add the `:no-doc` metadata. This will prevent that var from appearing in the generated API documentation, and signify that the function is intended to be private (and therefore, subject to change without notice).

Where possible, apply `:added` metadata on newly created namespaces or newly added functions (or vars).

When a new namespace is introduced, only the namespace needs the `:added` metadata, not the individual functions (or vars).

We indent with spaces, and follow default indentation patterns.

We value documentation. Lacinia docstrings are formatted with Markdown. Tests can also be great documentation.

23.5 Tests

We are test driven. We expect patches to include tests.

We may reject patches or pull requests that arrive without tests.

23.6 Documentation

Patches that change behavior and invalidate existing documentation will be rejected. Such patches should also update the documentation.

Ideally, patches that introduce new functionality will also include documentation changes.

Documentation is generated using Sphinx, which is not difficult to set up, but does require Python.

23.7 Backwards Compatibility

We respect backwards compatibility.

We make it a top priority to ensure that anyone who writes an application using Lacinia will be free from upgrade headaches, even as Lacinia gains more features.

We have, so far, expressly *not* documented the internal structure of compiled schema or parsed query, so that we can be free to be fluid in making future improvements.

CHAPTER 24

Using this library

This library aims to maintain feature parity to that of the official reference JavaScript implementation and be fully compliant with the GraphQL specification.

Lacinia can be plugged into any Clojure HTTP pipeline. The companion library lacinia-pedestal provides full HTTP support, including *GraphQL subscriptions*, for Pedestal.

24.1 Overview

A GraphQL server starts with a schema of exposed types.

This GraphQL schema is described as an EDN data structure:

```
{:enums
 {:episode
  {:description "The episodes of the original Star Wars trilogy."
   :values [:NEWHOPE :EMPIRE :JEDI]}}

 :interfaces
 {:character
  {:fields {:id {:type String}
            :name {:type String}
            :appears_in {:type (list :episode)}
            :friends {:type (list :character)}}}}

 :objects
 {:droid
  {:implements [:character]
   :fields {:id {:type String}
            :name {:type String}
            :appears_in {:type (list :episode)}
            :friends {:type (list :character)
                      :resolve :friends}
            :primary_function {:type (list String)}}}
```

(continues on next page)

(continued from previous page)

```
  :human
  {:implements [:character]
   :fields {:id {:type String}
            :name {:type String}
            :appears_in {:type (list :episode)}
            :friends {:type (list :character)
                      :resolve :friends}
            :home_planet {:type String}}}}

 :queries
 {:hero {:type (non-null :character)
         :args {:episode {:type :episode}}
         :resolve :hero}

  :human {:type (non-null :human)
          :args {:id {:type String
                      :default-value "1001"}}
          :resolve :human}

  :droid {:type :droid
          :args {:id {:type String
                      :default-value "2001"}}
          :resolve :droid}}}
```

The schema defines all the data that could possibly be queried by a client.

To make this schema useful, *field resolvers* must be added to it. These functions are responsible for doing the real work (querying databases, communicating with other servers, and so forth). These are attached to the schema after it is read from an EDN file, using the placeholder keywords in the schema, such as `:resolve :droid`.

The client uses the GraphQL query language to specify exactly what data should be returned in the result map:

```
{
  hero {
    id
    name
    friends {
      name
    }
  }
}
```

This translates to "run the hero query; return the default hero's id and name, and friends; just return the name of each friend."

Lacinia will return this as Clojure data:

```
{:data
 {:hero
  {:id "2001"
   :name "R2-D2"
   :friends [{:name "Luke Sykwalker"},
             {:name "Han Solo"},
             {:name "Leia Organa"}]}}}
```

This is because R2-D2 is, of course, considered the hero of the Star Wars trilogy.

This Clojure data can be trivially converted into JSON or other formats when Lacinia is used as part of an HTTP server application.

A key takeaway: GraphQL is a contract between a client and a server; it doesn't know or care where the data comes from; that's the province of the field resolvers. That's great news: it means Lacinia is equally adept at pulling data out of a single database as it is at integrating and organizing data from multiple backend systems.