com.walmartlabs/lacinia-pedestal Documentation

Release 0.10.1

Walmartlabs

Sep 11, 2021

Contents

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

This library, Lacinia-Pedestal, extends Lacinia to the web, building on the Pedestal framework. It provides Pedestal routes and interceptors to allow GraphQL schemas to be exposed as a web endpoint, as well as providing the GraphIQL client

CHAPTER 1

Overview

Warning: The newer `com.walmartlabs.lacinia.pedestal2 namespace` is recommended over the now-deprecated `com.walmartlabs.lacinia.pedestal` namespace. The new namespace was introduced to maintain backwards compatibility, but be aware of which you use, as the default URLs served differ with each namespace. This guide assumes you are using the latest `pedestal2` namespace and its defaults.

You start with a schema file, `resources/hello-world-schema.edn`, in this example:

```
{:queries
 {:hello
  {:type String}}}
```

From there there are three steps:

- Load and compile the schema
- Create a Pedestal service around the schema
- Start the Pedestal service

```
(ns demo.server
  (:require
    [clojure.edn :as edn]
    [clojure.java.io :as io]
    [com.walmartlabs.lacinia.pedestal2 :as p2]
    [com.walmartlabs.lacinia.schema :as schema]
    [com.walmartlabs.lacinia.util :as util]
    [io.pedestal.http :as http]))

(defn ^:private resolve-hello
  [context args value]
  "Hello, Clojurians!")

(defn ^:private hello-schema
```

(continues on next page)

(continued from previous page)

```
  []
  (-> (io/resource "hello-world-schema.edn")
      slurp
      edn/read-string
      (util/inject-resolvers {:queries/hello resolve-hello})
      schema/compile))

(def service (-> (hello-schema)
                 (p2/default-service nil)
                 http/create-server
                 http/start))
```

At the end of this, an instance of Jetty is launched on port 8888.

The GraphQL endpoint will be at `http://localhost:8888/api` and the GraphIQL client will be at `http://localhost:8888/ide`.

The options map provided to `default-service` allow a number of features of Lacinia-Pedestal to be configured or customized, though the intent of `default-service` is to just be initial scaffolding - it should be replaced with application-specific code.

CHAPTER 2

Request Format

Clients may send either a HTTP GET or HTTP POST request to execute a query.

In both cases, the request path will (by default) be `/api`.

2.1 POST application/json

When using POST with the `application/json` content type, the body of the request may contain the following keys:

`query` Required: The GraphQL query document, as a string.

`variables` Optional: A JSON object of variables (as defined and referenced in the query document).

`operationName` Optional: The name of the specific operation to execute, when the query document defines more than one named operation.

This is the standard and expected request format, and the only one directly supported with the `com.walmartlabs.lacinia.pedestal2` namespace.

2.2 GET (deprecated)

The GraphQL query document must be provided as query parameter `query`.

This is only supported with the `com.walmartlabs.lacinia.pedestal` namespace; `GET` is not supported out-of-the-box with `pedestal2`.

2.3 POST application/graphql (deprecated)

Warning: This format is fully supported, but represents a legacy format used internally at Wal-Mart, prior to the GraphQL community identifying an over-the-wire format. The `POST application/json` format is preferred.

The body of the request should be the GraphQL query document.

If the query document defines variables, they must be specified as the `variables` query parameter, as a string-ified JSON object.

This is only supported with the `com.walmartlabs.lacinia.pedestal` namespace; it is not supported out-of-the-box with `pedestal2`.

CHAPTER 3

Response

The response to a GraphQL request is a `application/json` object.

The response will include a `errors` key if there are fatal or non-fatal errors.

Fatal errors are those that occur before the query is executed; these represents parse failures or validation errors.

3.1 Response Format

The response format is always `application/json`.

The response body is the result map from executing the query (e.g., has `data` and/or `errors` keys).

In cases where there is a problem parsing or preparing the query, the response will still be in the regular format, e.g.:

```
{"errors": [{"message": "Request body is empty."}]}
```

GraphQL supports a third key, `extensions`, but does not define what content goes there; it is for application-specific extensions.

3.2 HTTP Status

The normal HTTP status is `200 OK`.

If there was a fatal error (such as a query parse error), the HTTP status will be `400 Bad Request`.

3.3 Status Conversion

Field resolvers may return error maps. If an error map contains a `:status` key, this value will be used as the overall HTTP status of the response.

When multiple error maps contains `:status`, the numerically largest value is used.

The `:status` key is removed from all error maps before the response is streamed to the client.

CHAPTER 4

Async

By default, Lacinia-Pedestal blocks the Pedestal request thread while executing the query. The default pedestal interceptor stack includes a synchronous query execution handler, `com.walmartlabs.lacinia.pedestal2/query-executor-handler`.

Lacinia also provides an asynchronous query execution handler: `com.walmartlabs.lacinia.pedestal2/async-query-executor-handler`.

When used in the interceptor stack, execution starts on a Pedestal request processing thread, but (at the discretion of individual field resolver functions) may continue on other threads.

Further, the return value from the asychronous handler is a channel, forcing Pedestal to switch to async mode.

Lacinia-Pedestal does not impose any restrictions on the number of requests it will attempt to process concurrently; normally, this is gated by the number of Pedestal request processing threads available.

When using the asynchronous query execution handler, you should provide application-specific interceptors to rate limit requests, or risk saturating your server.

CHAPTER 5

Subscriptions

Subscriptions are a way for a client to request notifications about arbitrary events defined by the server; this parallels how a query exposes arbitrary data defined by the server.

The essential support for GraphQL subscriptions is in the main Lacinia library.

Lacinia-Pedestal's subscription support is designed to be compatible with Apollo GraphQL, a popular library in the JavaScript domain[1]. Like Apollo, Lacinia-Pedestal uses WebSockets to create a durable connection between the client and the server.

5.1 Overview

A client (typically, a web browser or mobile phone) will establish a connection to the server, and convert it to a full-duplex WebSocket connection.

This single WebSocket connection will be multiplexed to handle any number of subscription requests from the client.

When a subscription is requested, a streamer defined in the GraphQL schema is invoked. A *streamer* is similar to a field resolver; it has two responsibilities:

- Do whatever setup is necessary, then as new events are available, provide data to a source stream callback function.
- Return a cleanup function that shuts down whatever was previously set up.

Most commonly, a streamer will subscribe to some external feed such as a JMS or Kafka queue, or perhaps a core.async pub or channel.

When a streamer passes nil to the callback, a clean shutdown of the subscription occurs; the client is sent a completion message. The completion message informs the client that the stream of events has completed, and that it should not attempt to reconnect.

[1] Apollo defines a particular contract for how the client and server communicate; this includes heartbeats, and an explicit way for the server to signal to the client that the subscription has completed.
The Apollo project also provides clients in several languages.

The definition of "completed" here is entirely up to the application. For example, a field argument could specify the maximum number of values to stream, and the streamer can pass nil after sufficient values are streamed.

The cleanup function is invoked when the client closes the subscription, when the connection from the client is lost due to a network partition, or when the streamer passes nil to the callback.

5.2 Configuration

When using `com.walmartlabs.lacinia.pedestal2/default-service`, subscriptions are always enabled, but `default-service` is always intended to be replaced in a live application.

The underlying function `com.walmartlabs.lacinia.pedestal2/enable-subscriptions` does the work of enabling subscriptions; the function is passed subscription options:

The following keys are commonly used:

`:subscriptions-path` Path to use in subscriptions WebSocket requests; defaults to `/ws`,

`:keep-alive-ms` The interval at which keep-alive messages are sent to the client; defaults to 30 seconds.

`:subscription-interceptors` A seq of interceptors used when processing GraphQL query, mutation, or subscription requests via the WebSocket connection. This is used when overriding the default interceptors.

Further options are described by listener-fn-factory.

5.3 Connection Parameters

When the client creates a connection, it may pass a payload in the `connection_init` message; this is the connection parameters, and is made available to the streamer and resolver in the context under the `:com.walmartlabs.lacinia/connection-parameters` key.

5.4 Endpoint

Subscriptions are processed on a second endpoint; normal requests continue to be sent to `/api`, but subscription requests must use `/ws`.

The `/ws` endpoint does not handle ordinary requests; instead it is used only to establish the WebSocket connection. From there, the client sends WebSocket text messages to initiate a subscription, and the server sends WebSocket text messages for subscription updates and keep alive messages.

Subscription requests are not allowed in `/api` path.

5.5 GraphiQL

GraphiQL, when enabled, is configured with subscriptions enabled; this means that GraphiQL can send `subscription` queries.

CHAPTER 6

Interceptors

com.walmartlabs.lacinia.pedestal2 defines Pedestal interceptors and supporting code.

The inject function (added in 0.7.0) adds (or replaces) an interceptor to a vector of interceptors.

6.1 Example

```
(ns server
  (:require
    [com.stuartsierra.component :as component]
    [com.walmartlabs.lacinia.pedestal2 :as p2]
    [com.walmartlabs.lacinia.pedestal :refer [inject]]
    [io.pedestal.interceptor :refer [interceptor]
    [io.pedestal.http :as http]))

(defn ^:private extract-user-info
  [request]
  ;; This is very application-specific ...
  )

(def ^:private user-info-interceptor
  (interceptor
   {:name ::user-info
    :enter (fn [context]
             (let [{:keys [request]} context
                   user-info (extract-user-info request)]
               (assoc-in context [:request :lacinia-app-context :user-info] user-
↪info)))}))

(defn ^:private interceptors
  [schema]
  (-> (p2/default-interceptors schema nil)
      (inject user-info-interceptor :after ::p2/inject-app-context)))
```

(continues on next page)

(continued from previous page)

```
(defn ^:private create-server
  [compiled-schema port]
  (let [interceptors (interceptors compiled-schema)
        routes #{["/api" :post interceptors :route-name ::api]}]
    (-> {:env :dev
         ::http/routes routes
         ::http/port port
         ::http/type :jetty
         ::http/join? false}
        http/create-server
        http/start)))

(defrecord Server [schema-source server port]

  component/Lifecycle

  (start [this]
    (let [compiled-schema (:schema schema-source)
          server' (create-server compiled-schema port)]
      (assoc this :server server')))

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

There's a lot to process in this more worked example:

- We're using Component to organize our code and dependencies.
- The schema is provided by a source component (in the next listing), injected as a dependency into the `Server` component.
- We're building our Pedestal service explicitly, rather than using `default-service`.

The interceptor is responsible for putting the user info *into* the request, and then it's simple to get that data inside a resolver function:

```
(ns schema-source
  (:require
    [com.walmartlabs.lacinia.schema :as schema]
    [com.walmartlabs.lacinia.util :as util]
    [com.stuartsierra.component :as component]
    [clojure.edn :as edn]
    [clojure.java.io :as io]))

(defn ^:private resolve-user
  [context _args _value]
  (let [{:keys [user-info]} context]
    ;; Use user-info to get the data from somewhere ...
    ))

(defrecord SchemaSource []

  component/Lifecycle

  (start [this]
    (assoc this :schema (-> (io/resource "schema.edn")
```

(continues on next page)

(continued from previous page)

```
                              slurp
                              edn/read-string
                              (util/inject-resolvers {:queries/user resolve-user})
                              schema/compile)))

  (stop [this]
    (dissoc this :schema)))
```

Again, it's a little sketchy because we don't know what the `user-info` data is, how its stored in the request, or what is done with it ... but the `:user-info` put in place by the interceptor is a snap to gain access to in any resolver function.

Tip: The `inject` function is useful for making one or two small additions to the default interceptors, but any more than that will likely lead to confusion about what order the items in the interceptor pipeline are in; better to dupliciate the code from `com.walmartlabs.lacinia.pedestal2/default-interceptors` instead.

CHAPTER 7

Request Tracing

Lacinia includes support for request tracing, which identifies how much time Lacinia spends parsing, validating, and processing the overall request.

By default, this is enabled by passing the header value `lacinia-tracing` set to `true` (or any non-blank string).

Further, the default is for the GraphiQL IDE to provide this value; queries using the IDE will trigger tracing behavior (often resulting in very, very large responses).

You will typically want to disable tracing in production, by removing the `:com.walmartlabs.lacinia.pedestal2/enable-tracing` *interceptor*.

Index

Symbols

O

Q

V