

Lab5 Egress: Part 1 - REST vs GraphQL

Lorin Speybrouck

Remarks

When making the first task I found it frustrating that it was not obvious what a API route parameters where. For this purpose I implemented [Type-safe routing](#) with the ktor Resources plug in, this made it possible to define the name of parameters and query parameters and there type in a class. This is show in the following tasks.

Another tool I missed was Swagger to easily visualise the API routes and test them. For this I added [Ktor OpenAPI Tools](#), this library can automatically generate OpenAPI documentation from code and also provides a route for the Swagger UI. This is also shown in the following task.

When developing the graphql endpoint I also missed a playground with auto generated documentation. For this I used graphiQL using the already used [GraphQL Kotlin](#) library. This is show in task 2.

This all results in the following `base.kt` file

```

// base.kt file
fun Application.setup() {
    install(StatusPages) {
        exception<Throwable>(handler = ::onError)
        status(HttpStatusCode.NotFound, handler = ::onNotFound)
    }
    install(OpenApi) {
        schemas { generator = SchemaGenerator.kotlinux {} }
        autoDocumentResourcesRoutes = true
    }
    install(Resources)
    routing {
        trace {
            application.log.info("Incoming call: {}"),
it.call.request.uri.replace("\n", "\\n"))
        }
        get("/", {
            summary = "Index";
            description = "Hello world route"
            response { HttpStatusCode.OK to { body<String> {} } }
        }) {
            call.respond(HttpStatusCode.Companion.OK, "Hello world")
        }
        route("api.json") {
            openApi()
        }
        route("swagger") {
            swaggerUI("/api.json") { displayRequestDuration = true }
        }
    }
}


```

Tasks

Task 1: Implementing REST

Sources

Result



Responses

Snippets ▾

cURL (bash) cURL (PowerShell) cURL (CMD)

```
curl -X 'GET' \
  http://127.0.0.1:8087/rest/sources' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8087/rest/sources

Server response

Code Details

200

Response body

```
{
  "sources": [
    "ESP32-via-RPI",
    "RPI",
    "esp32",
    "vip_area"
  ]
}
```

Response headers

```
connection: keep-alive
content-length: 102
content-type: application/json
```

Code

```
// Sources.kt
@Serializable
data class SourcesResponse(
    val sources: List<String>
)

@Resource("sources")
class Sources(
)

fun Route.sources() {
    get<Sources>({
        summary="Get all sources"
        description = "Returns an overview of all sources that ever
submitted data"
        response {
            HttpStatusCode.OK to { body<SourcesResponse> {} }
        }
    }) { _ ->
        val records = Influx.query(getSourcesFluxQuery())
        val foundSources = records.mapNotNull { record ->
            record.getValueByKey("source")?.toString()
        }

        call.respond(SourcesResponse(foundSources))
    }
}
```

The sources get request is the simplest query of all REST requests. It gets the data from the database using the sources Flux query and takes all sources values to return them in a SourcesResponse.

Counts

Result

Responses

Snippets ▾

cURL (bash)

cURL (PowerShell)

cURL (CMD)

```
curl -X 'GET' \
  'http://127.0.0.1:8087/rest/counts?source=ESP32-via-RPI' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8087/rest/counts?source=ESP32-via-RPI
```

Server response

Code

Details

200

Response body

```
{
  "events": [
    {
      "timestamp": 1742327845063,
      "value": 1,
      "source": "ESP32-via-RPI"
    },
    {
      "timestamp": 1742327885733,
      "value": 0,
      "source": "ESP32-via-RPI"
    },
    {
      "timestamp": 1742328002599,
      "value": 1,
      "source": "ESP32-via-RPI"
    },
    {
      "timestamp": 1742328063125,
      "value": 0,
      "source": "ESP32-via-RPI"
    },
    {
      "timestamp": 1742328106832,
      "value": 1,
      "source": "ESP32-via-RPI"
    }
  ]
}
```

Response headers

```
connection: keep-alive
content-length: 1254
content-type: application/json
```

Code

```

// Counts.kt
@Serializable
data class EventData(
    val timestamp: Long,
    val value: Int,
    val source: String
)
@Serializable
data class CountsResponse(
    val events: List<EventData>
)

@Resource("counts")
class Counts(
    val start: Long? = null,
    val stop: Long? = null,
    val source: String? = null
)
fun Route.counts() {
    get<Counts>({
        summary = "Get counts over time"
        description = "Returns an overview of the changing count values
over time. Filterable by start and stop timestamps (epochs in ms) and
source tag."
        response {
            HttpStatusCode.OK to { body<CountsResponse> {} }
        }
    }) { counts ->
        val start = if(counts.start != null)
Instant.fromEpochMilliseconds(counts.start) else Defaults.start()
        val stop = if(counts.stop != null)
Instant.fromEpochMilliseconds(counts.stop) else Defaults.stop()
        val source = counts.source

        val records = Influx.query( getCountFluxQuery(start, stop,
source))
        val events = records.map { record ->
            EventData(
                timestamp = record.time?.toEpochMilli() ?: 0,
                value = record.value.toString().toInt(),
                source = record.getValueByKey("source")?.toString() ?:
"unknown"
            )
        }

        call.respond(CountsResponse(events))
    }
}

```

The Counts get request first checks the provided query parameters to see if defaults need to be used. After this it gets its data from the database using the count Flux query. It returns the data in a CountsResponse which contains a list of EventData.

IDs

Result

Responses

Snippets ▾

cURL (bash)

cURL (PowerShell)

cURL (CMD)

```
curl -X 'GET' \
'http://127.0.0.1:8087/rest/ids?stop=1743263951000&source=RPI' \
-H 'accept: */*'
```

Request URL

```
http://127.0.0.1:8087/rest/ids?stop=1743263951000&source=RPI
```

Server response

Code

Details

200

Undocumented

Response body

```
}
{
  "timestamp": 1741814842101,
  "id": "10",
  "source": "RPI"
},
{
  "timestamp": 1741814843159,
  "id": "1",
  "source": "RPI"
},
{
  "timestamp": 1741814844217,
  "id": "0",
  "source": "RPI"
},
{
  "timestamp": 1741814845266,
  "id": "3",
  "source": "RPI"
},
{
  "timestamp": 1741814846313,
  "id": "0",
  "source": "RPI"
},
}
```

Download

Response headers

```
connection: keep-alive
content-length: 16728
content-type: application/json
```

Code

```

// IDs.kt
@Serializable
data class IDsResponse(
    val events: List<IDsEvent>
)
@Serializable
data class IDsEvent(
    val timestamp: Long,
    val id: String,
    val source: String
)

@Resource("ids")
class IDs(
    val start: Long? = null,
    val stop: Long? = null,
    val source: String? = null
)

fun Route.ids() {
    get<IDs>({
        summary="Get scanned ids overview"
        description = "Returns an overview of scanned ids. Filterable by
start and stop timestamps (epochs in ms) and source tag."
        response {
            HttpStatusCode.OK to { body<IDsResponse> {} }
        }
    }) { ids ->
        val start = if(ids.start != null)
Instant.fromEpochMilliseconds(ids.start) else Defaults.start()
        val stop = if(ids.stop != null)
Instant.fromEpochMilliseconds(ids.stop) else Defaults.stop()
        val source = ids.source

        val records = Influx.query(getIDsFluxQuery(start, stop, source))
        val events = records.map { record ->
            IDsEvent(
                timestamp = record.time?.toEpochMilli() ?: 0,
                id = record.getValueByKey("_value")?.toString() ?:
"unknown",
                source = record.getValueByKey("source")?.toString() ?:
"unknown"
            )
        }

        call.respond(IDsResponse(events))
    }
}

```

The IDs query is similar to the Counts query but uses the get IDs Flux query.

Attendance

Result

Responses

Snippets ▾

cURL (bash)

cURL (PowerShell)

cURL (CMD)

```
curl -X 'GET' \
'http://127.0.0.1:8087/rest/attendance?start=1711727951000' \
-H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8087/rest/attendance?start=1711727951000
```

Server response

Code

Details

200

Response body

```
{
  "events": [
    {
      "timestamp": 1741814689970,
      "id": "2",
      "arrival": true,
      "source": "RPI"
    },
    {
      "timestamp": 1741814691028,
      "id": "3",
      "arrival": true,
      "source": "RPI"
    },
    {
      "timestamp": 1741814692084,
      "id": "0",
      "arrival": false,
      "source": "RPI"
    },
    {
      "timestamp": 1741814693136,
      "id": "3",
      "arrival": false,
      "source": "RPI"
    }
  ]
}
```

Response headers

```
connection: keep-alive
content-length: 53180
content-type: application/json
```

Code


```

// Attendance.kt
@Serializable
data class AttendanceResponse(
    val events: List<AttendanceEvent>
)
@Serializable
data class AttendanceEvent(
    val timestamp: Long,
    val id: String,
    val arrival: Boolean,
    val source: String
)

@Resource("attendance")
class Attendance(
    val start: Long? = null,
    val stop: Long? = null,
    val source: String? = null
)

fun Route.attendance() {
    get<Attendance>({
        summary="Get attendance over time"
        description="Returns an overview of the attendance over time.
Filterable by start and stop timestamps (epochs in ms) and source tag."
        response {
            HttpStatusCode.OK to { body<AttendanceResponse> {} }
        }
    }) { attendance ->
        val start = if(attendance.start != null)
Instant.fromEpochMilliseconds(attendance.start) else Defaults.start()
        val stop = if(attendance.stop != null)
Instant.fromEpochMilliseconds(attendance.stop) else Defaults.stop()
        val source = attendance.source

        val records = Influx.query(getAttendanceFluxQuery(start, stop,
source))
        val events = records.map { record ->
AttendanceEvent(
            timestamp = record.time?.toEpochMilli() ?: 0,
            id = record.getValueByKey("_value")?.toString() ?:
"unknown",
            arrival =
record.getValueByKey("arrival")?.toString()?.toBoolean() ?: false,
            source = record.getValueByKey("source")?.toString() ?:
"unknown"
        )
    }
        call.respond(AttendanceResponse(events))
    }
}

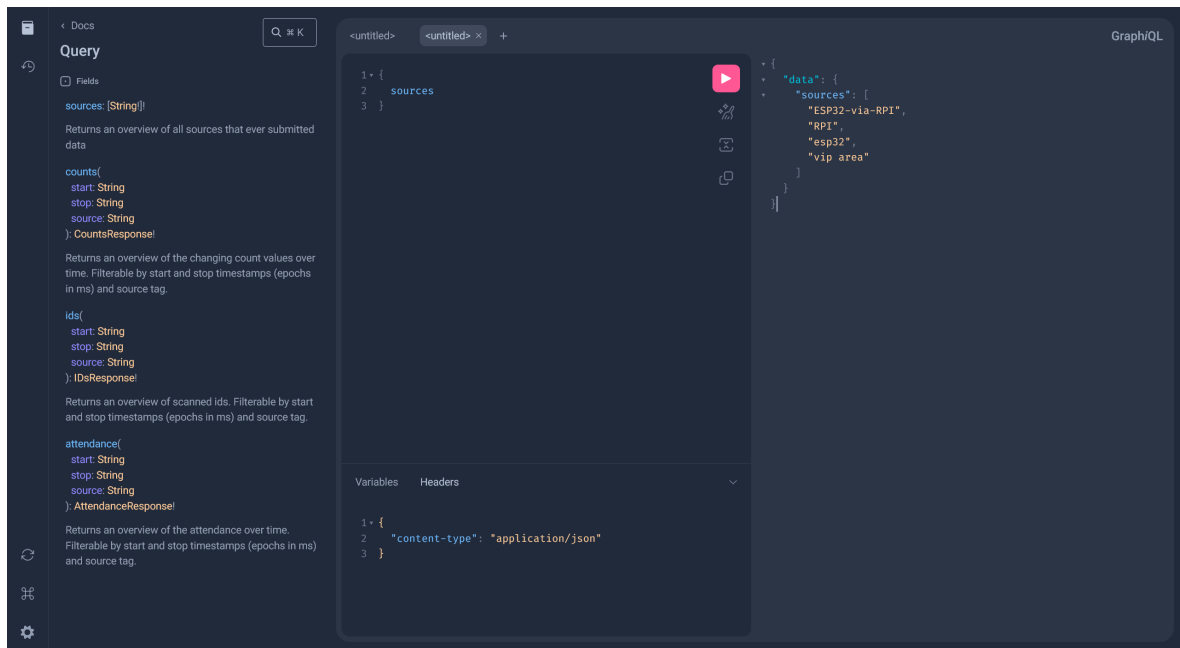
```

The Attendance query is similar to the Counts query but uses the get attendance Flux query.

Task 2: Implementing GraphQL

SourcesQueryService

Result



```
{
  sources
}
```

Code

```
// SourcesQueryService.kt
class SourcesQueryService: Query {
    @GraphQLDescription("Returns an overview of all sources that ever submitted data")
    suspend fun sources(): List<String> {
        val records = Influx.query(getSourcesFluxQuery())
        val foundSources = records.mapNotNull { record ->
            record.getValueByKey("source")?.toString()
        }

        return foundSources
    }
}
```

The SourcesQueryService is adapted from the REST equivalent and only differs by directly returning the found sources.

CountsQueryService

Result

The screenshot shows the GraphQL IDE interface. On the left, the 'Query' tab is active, displaying a query and its schema. The query is:

```
1 {
2   counts(source: "ESP32-via-RPI") {
3     events {
4       source
5       timestamp
6       value
7     }
8   }
9 }
```

The schema on the left includes fields like `sources`, `counts`, `ids`, and `attendance`, each with their respective arguments and return types.

On the right, the 'JSON' tab shows the response:

```
{
  "data": {
    "counts": {
      "events": [
        {
          "source": "ESP32-via-RPI",
          "timestamp": "1742327845063",
          "value": 1
        },
        {
          "source": "ESP32-via-RPI",
          "timestamp": "1742327885733",
          "value": 0
        },
        {
          "source": "ESP32-via-RPI",
          "timestamp": "1742328062599",
          "value": 1
        },
        {
          "source": "ESP32-via-RPI",
          "timestamp": "1742328063125",
          "value": 0
        },
        {
          "source": "ESP32-via-RPI",
          "timestamp": "1742328106832",
          "value": 1
        },
        {
          "source": "ESP32-via-RPI",
          "timestamp": "1742328132499",
          "value": 0
        },
        {
          "source": "ESP32-via-RPI",
          "timestamp": "1742328163163",
          "value": 0
        }
      ]
    }
  }
}
```

```
{
  counts(source: "ESP32-via-RPI") {
    events {
      source
      timestamp
      value
    }
  }
}
```

Code

```

// CountsQueryService.kt
@Serializable
data class EventData(
    val timestamp: String,
    val value: Int,
    val source: String
)
@Serializable
data class CountsResponse(
    val events: List<EventData>
)

class CountsQueryService: Query {
    @GraphQLDescription("Returns an overview of the changing count
values over time. Filterable by start and stop timestamps (epochs in ms)
and source tag.")
    suspend fun counts(
        start: String? = null,
        stop: String? = null,
        source: String? = null
    ) : CountsResponse {
        val startInstant = try {
            if (start != null)
Instant.fromEpochMilliseconds(start.toLong()) else Defaults.start()
        } catch (e: NumberFormatException) {
            throw IllegalArgumentException("Invalid start timestamp:
$start. Expected a numeric value representing epoch time in ms")
        }
        val stopInstant = try {
            if (stop != null)
Instant.fromEpochMilliseconds(stop.toLong()) else Defaults.stop()
        } catch (e: NumberFormatException) {
            throw IllegalArgumentException("Invalid stop timestamp:
$stop. Expected a numeric value representing epoch time in ms")
        }

        val records = Influx.query( getCountFluxQuery(startInstant,
stopInstant, source))
        val events = records.map { record ->
            EventData(
                timestamp = (record.time?.toEpochMilli() ?:
0).toString(),
                value = record.value.toString().toInt(),
                source = record.getValueByKey("source")?.toString() ?:
"unknown"
            )
        }

        return CountsResponse(events)
    }
}

```

Code

```

// IDQueryService.kt
@Serializable
data class IDsResponse(
    val events: List<IDsEvent>
)
@Serializable
data class IDsEvent(
    val timestamp: String,
    val id: String,
    val source: String
)

class IDQueryService: Query {
    @GraphQLDescription("Returns an overview of scanned ids. Filterable
by start and stop timestamps (epochs in ms) and source tag.")
    suspend fun ids(
        start: String? = null,
        stop: String? = null,
        source: String? = null
    ) : IDsResponse {
        val startInstant = try {
            if (start != null)
                Instant.fromEpochMilliseconds(start.toLong()) else Defaults.start()
        } catch (e: NumberFormatException) {
            throw IllegalArgumentException("Invalid start timestamp:
$start. Expected a numeric value representing epoch time in ms")
        }
        val stopInstant = try {
            if (stop != null)
                Instant.fromEpochMilliseconds(stop.toLong()) else Defaults.stop()
        } catch (e: NumberFormatException) {
            throw IllegalArgumentException("Invalid stop timestamp:
$stop. Expected a numeric value representing epoch time in ms")
        }

        val records = Influx.query(getIDsFluxQuery(startInstant,
stopInstant, source))
        val events = records.map { record ->
            IDsEvent(
                timestamp = (record.time?.toEpochMilli() ?:
0).toString(),
                id = record.getValueByKey("_value")?.toString() ?:
"unknown",
                source = record.getValueByKey("source")?.toString() ?:
"unknown"
            )
        }

        return IDsResponse(events)
    }
}

```

The IDQueryService is similar to the CountsQueryService.

AttendanceQueryService

Result

The screenshot shows the GraphQL IDE interface. On the left, the 'Query' panel lists several fields: `sources` (returns an overview of all sources), `counts` (returns an overview of changing count values), `ids` (returns an overview of scanned IDs), and `attendance` (returns an overview of attendance over time). The main editor displays a query:

```
1 {
2   attendance(source: "ESP32-via-RPI") {
3     events {
4       id
5       arrival
6     }
7   }
8 }
```

 Below the query editor, the 'Variables' and 'Headers' tabs are visible, with 'Variables' showing `1 { 2 "content-type": "application/json" 3 }`. On the right, the 'Execute query' button is highlighted, and the resulting JSON data is shown:

```
{
  "data": {
    "attendance": [
      {
        "id": "A021590E",
        "arrival": false
      },
      {
        "id": "A021590E",
        "arrival": true
      },
      {
        "id": "A021590E",
        "arrival": false
      },
      {
        "id": "A021590E",
        "arrival": true
      },
      {
        "id": "A021590E",
        "arrival": false
      },
      {
        "id": "A021590E",
        "arrival": true
      },
      {
        "id": "A021590E",
        "arrival": false
      },
      {
        "id": "A021590E",
        "arrival": true
      }
    ]
  }
}
```

```
{
  attendance(source: "ESP32-via-RPI")
  {
    events {
      id
      arrival
    }
  }
}
```

Code

```

// AttendanceQueryService.kt
@Serializable
data class AttendanceResponse(
    val events: List<AttendanceEvent>
)
@Serializable
data class AttendanceEvent(
    val timestamp: String, // Long is not supported in GraphQL
    val id: String,
    val arrival: Boolean,
    val source: String
)
class AttendanceQueryService: Query {
    @GraphQLDescription("Returns an overview of the attendance over
time. Filterable by start and stop timestamps (epochs in ms) and source
tag.")
    suspend fun attendance(
        start: String? = null,
        stop: String? = null,
        source: String? = null
    ): AttendanceResponse {
        val startInstant = try {
            if (start != null)
Instant.fromEpochMilliseconds(start.toLong()) else Defaults.start()
        } catch (e: NumberFormatException) {
            throw IllegalArgumentException("Invalid start timestamp:
$start. Expected a numeric value representing epoch time in ms")
        }
        val stopInstant = try {
            if (stop != null)
Instant.fromEpochMilliseconds(stop.toLong()) else Defaults.stop()
        } catch (e: NumberFormatException) {
            throw IllegalArgumentException("Invalid stop timestamp:
$stop. Expected a numeric value representing epoch time in ms")
        }

        val records = Influx.query(getAttendanceFluxQuery(startInstant,
stopInstant, source))
        val events = records.map { record ->
            AttendanceEvent(
                timestamp = (record.time?.toEpochMilli() ?:
0).toString(),
                id = record.getValueByKey("_value")?.toString() ?:
"unknown",
                arrival =
record.getValueByKey("arrival")?.toString()?.toBoolean() ?: false,
                source = record.getValueByKey("source")?.toString() ?:
"unknown"
            )
        }
        return AttendanceResponse(events)
    }
}

```


The AttendanceQueryService is similar to the CountsQueryService.

Task 3: Add Egress API to Your Microservices Architecture

The Egress API was successfully deployed on the cloud to this url: [Egress](#). This was done by following the same steps as lab 3.

Optional Task 4: Differences between GraphQL and REST

Unfinished

Task 5: Obtaining forecasts

Code

```
// Model.kt
suspend fun predict(
    inputStart: Instant,
    inputStop: Instant,
    source: String,
    method: String
): Result<List<Pair<Instant, Int>>> {
    val records = Influx.query(getCountFluxQuery(inputStart, inputStop,
    source))
    if (records.isEmpty())
        return Result.success(emptyList())

    val counts = records.map { record -> record.value.toString().toInt()
    }.takeLast(200)
    val timestamps = records.map { record -> record.time?.toEpochMilli()
    ?: 0 }.takeLast(200)

    return try {
        val requestBody = PredictionRequest(timestamps, counts)
        val response = client.post("$url/$method") {
            contentType(ContentType.Application.Json)
            setBody(requestBody)
        }

        val responseBody = response.body<PredictionResponse>()
        val predictions =
        responseBody.timestamps.zip(responseBody.counts) { timestamp, count ->
            Pair(Instant.fromEpochMilliseconds(timestamp), count)
        }

        Result.success(predictions)
    } catch (e: Exception) {
        LOGGER.error("Failed to get predictions: ${e.message}")
        Result.failure(e)
    }
}
```

Forecasts are created using the Counts query. From the results counts and timestamps are extracted. (These are limited because too many results give problems with the JSON serializer). After this a PredictionRequest is created using the timestamps and counts as separate lists. The PredictionResponse is then mapped to a list of pairs of timestamp and count, which are then returned.

Task 6: Scheduling and caching forecasts

Code

```
// ServiceSyncJob.kt
fun start() {
    if (job != null)
        return

    job = scope.launch {
        while (isActive) {
            try {
                val now = Clock.System.now()
                val result = Model.predict(begin, now, source, method)
                result.fold(
                    onSuccess = { predictions ->
                        if (predictions.isNotEmpty())
                            ServiceCache.put(method, source,
                                predictions)
                        LOGGER.info("Stored ${predictions.size}
                                predictions in cache for source: $source, method: $method")
                        begin = now
                    },
                    onFailure = { error ->
                        LOGGER.error("Failed to get predictions:
                                ${error.message}")
                    }
                )
            } catch (e: Exception) {
                if (e is CancellationException) throw e
                LOGGER.error("Error in sync job: ${e.message}", e)
            } finally {
                delay(syncDelay)
            }
        }
    }
}
```

In the ServiceSyncJob a job is created that calls the function created in the previous task. The results of this are put into the service cache, this happens for every . After this the job waits for syncDelay, before repeating.

Task 7: GETting forecasts

Result

Responses

Snippets ▾

cURL (bash) cURL (PowerShell) cURL (CMD)

```
curl -X 'GET' \
  'http://127.0.0.1:8087/rest/forecast?source=vip%20area&method=mean_forecast' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8087/rest/forecast?source=vip%20area&method=mean_forecast
```

Server response

Code Details

200

Response body

```
{
  "source": "vip area",
  "method": "mean_forecast",
  "predictions": [
    {
      "timestamp": 1743689274881,
      "value": 5
    },
    {
      "timestamp": 1743689334881,
      "value": 5
    },
    {
      "timestamp": 1743689394881,
      "value": 5
    },
    {
      "timestamp": 1743689454881,
      "value": 5
    },
    {
      "timestamp": 1743689514881,
      "value": 5
    },
    {
      "timestamp": 1743689574881,
      "value": 5
    }
  ]
}
```

Response headers

```
connection: keep-alive
content-length: 18166
content-type: application/json
```

Code

```

// Forecasts.kt
@Serializable
data class PredictionData(
    val timestamp: Long,
    val value: Int
)

@Serializable
data class ForecastResponse(
    val source: String,
    val method: String,
    val predictions: List<PredictionData>
)

@Resource("forecast")
class Forecast(
    val time: Long? = null,
    val source: String,
    val method: String
)

fun Route.forecast() {
    get<Forecast>({
        summary="Get the forecast for a given source and method"
        description="Returns the best fitting forecast given the
timestamp`time` (epochs in ms) or using default value
Clock.System.now(), according to the provided source value (tag)"
        response {
            HttpStatusCode.OK to { body<ForecastResponse> {} }
        }
    }) { forecast ->
        val timeInstant = if(forecast.time != null)
Instant.fromEpochMilliseconds(forecast.time) else Clock.System.now()
        val source = forecast.source
        val method = forecast.method

        val predictions = ServiceCache.get(method, source, timeInstant)
        if (predictions.isNullOrEmpty()) {
            throw IllegalArgumentException("No predictions available
for the given source: $source and method: $method")
        }
        val formattedPredictions = predictions.map { pair ->
            PredictionData(pair.first.toEpochMilliseconds(),
pair.second)
        }

        call.respond(
            ForecastResponse(
                source = source,
                method = method,
                predictions = formattedPredictions
            )
        )
    }
}

```

In the forecasts get requests the cache, which is filled in the previous task, is queried for an appropriate prediction given the passed query parameters. The prediction is then returned in a ForecastResponse.

Questions

Question 1

Why is it useful to expose the same content (same fields) from both REST and GraphQL queries?

This gives the consumer of the API choice. REST queries contain a lot of information and when a user needs all of this information this will be the fastest way. But if the consumer needs only limited data or needs the data to answer further queries(multiple back and forth calls) GraphQL will be faster, despite its higher overhead.

Question 2

Define the four GraphQL services using the GraphQL Schema Definition Language (SDL). If you need help doing this, here is a good explanation and examples of the SDL.

An advantage of the approach I took is that this is generated automatically.

```

schema {
  query: Query
}

type Query {
  "Returns an overview of the attendance over time. Filterable by start
  and stop timestamps (epochs in ms) and source tag."
  attendance(source: String, start: String, stop: String):
  AttendanceResponse!
  "Returns an overview of the changing count values over time.
  Filterable by start and stop timestamps (epochs in ms) and source tag."
  counts(source: String, start: String, stop: String): CountsResponse!
  "Returns an overview of scanned ids. Filterable by start and stop
  timestamps (epochs in ms) and source tag."
  ids(source: String, start: String, stop: String): IDsResponse!
  "Returns an overview of all sources that ever submitted data"
  sources: [String!]!
}
type AttendanceEvent {
  arrival: Boolean!
  id: String!
  source: String!
  timestamp: String!
}
type AttendanceResponse {
  events: [AttendanceEvent!]!
}
type CountsResponse {
  events: [EventData!]!
}
type EventData {
  source: String!
  timestamp: String!
  value: Int!
}
type IDsEvent {
  id: String!
  source: String!
  timestamp: String!
}
type IDsResponse {
  events: [IDsEvent!]!
}

```

Question 3

The forecasting job system helps with improving the egress' response time when querying forecasting results. Give two more reasons why scheduling and/or caching these requests/results is beneficial.

- Efficiency: running the forecast query takes a lot of system resources(AI model), these resources can be better used for responding to actual api requests

- Consistency: by caching the result multiple users will get the same answer, otherwise they would get different data because a AI model contains randomness