

GHENT UNIVERSITY

CLOUD APPLICATIONS AND MOBILE (E736010)

Lab 5 Egress: Part 1 - REST vs GraphQL

Professor:

Prof. dr. ir. Sofie VAN HOECKE

Assistants:

Dr. Jennifer B. SARTOR

Ing. Tom WINDELS

2024 - 2025



UNIVERSITEIT
GENT

Table of contents

1	Introduction	1
2	Goals	2
3	Preparation: Technologies	3
3.1	Kotlin and IntelliJ IDEA	3
3.2	Ktor	3
3.3	Time	4
3.4	Serialization	4
3.5	GraphQL	4
4	Tutorials	5
4.1	Application structure	5
4.2	Testing	8
5	Tasks	9
5.1	Task 1: Implementing REST	9
5.2	Task 2: Implementing GraphQL	9
5.3	Task 3: Add Egress API to Your Microservices Architecture	10
5.4	Optional Task 4: Differences between GraphQL and REST	10
5.5	Task 5: Obtaining forecasts	10
5.6	Task 6: Scheduling and caching forecasts	11
5.7	Task 7: GETting forecasts	12
6	Question(s)	12
7	Summary	12
8	Material to submit	12

1 Introduction

The previous lab sessions set up the microservice-oriented architecture (see Figure 1). This lab session builds upon this architecture by providing APIs that will offer the sensor data to the outside world. You will be implementing the Egress API microservice in the picture. These APIs will later be consumed by your mobile application.

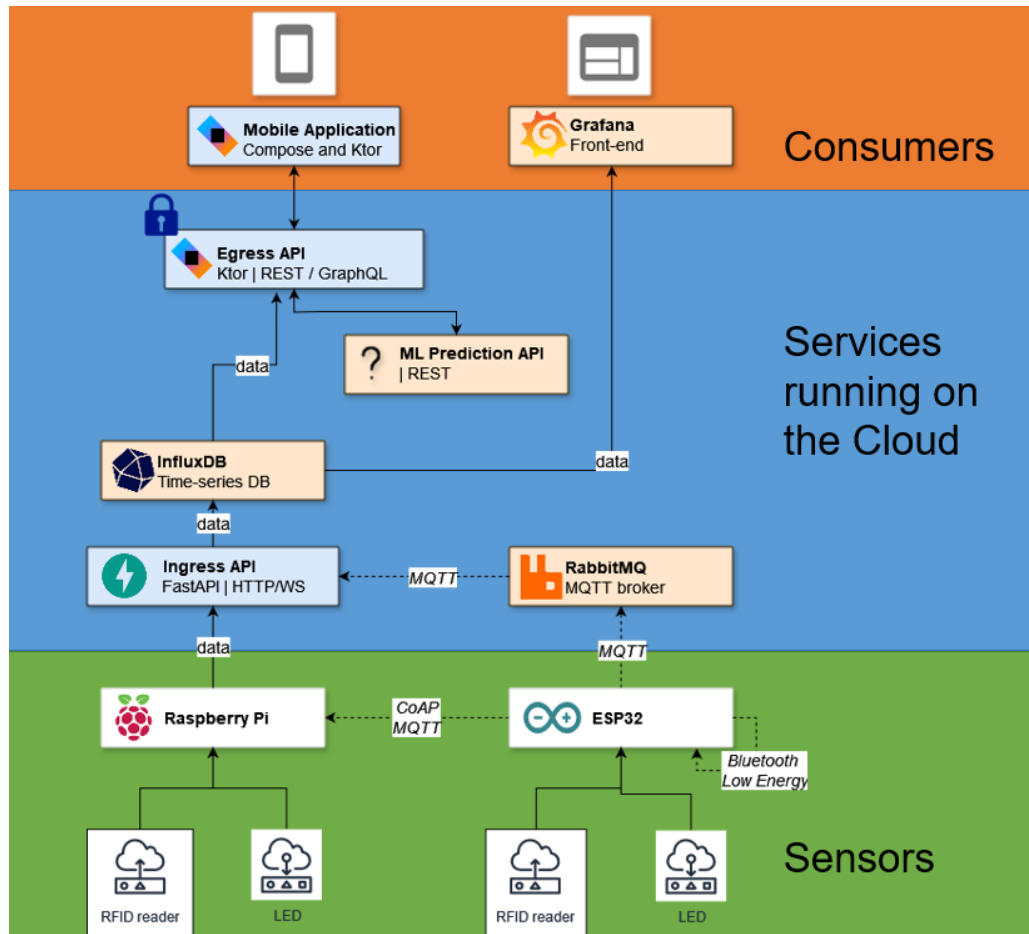


Figure 1: IoT stack architecture

In this assignment, you will have to construct a REST API and a GraphQL API, both of which query your InfluxDB for data and display it to the user, but they use different query languages. Then you will add an egress-api to your microservice architecture, similarly to how you did with the ingress-api. You will deploy your egress-api to the cloud. In an optional task, you can compare the two techniques in regards to how fast they are and the size of the data they return from a query.

Then we have three tasks that will implement forecasting future expected behavior (how your count of people will change) based on the collected sensor data of the past. For this you will communicate with a forecasting API (seen as the ML Prediction API box in Figures 1 and 2), cache the forecasts in your egress, and write an endpoint to expose these forecasts from egress. An overview of parts of the IoT stack that this lab covers can be found in Figure 2. The RPi is grayed out because it is not necessary to send actual data. Using random RFID IDs is fine from now on.

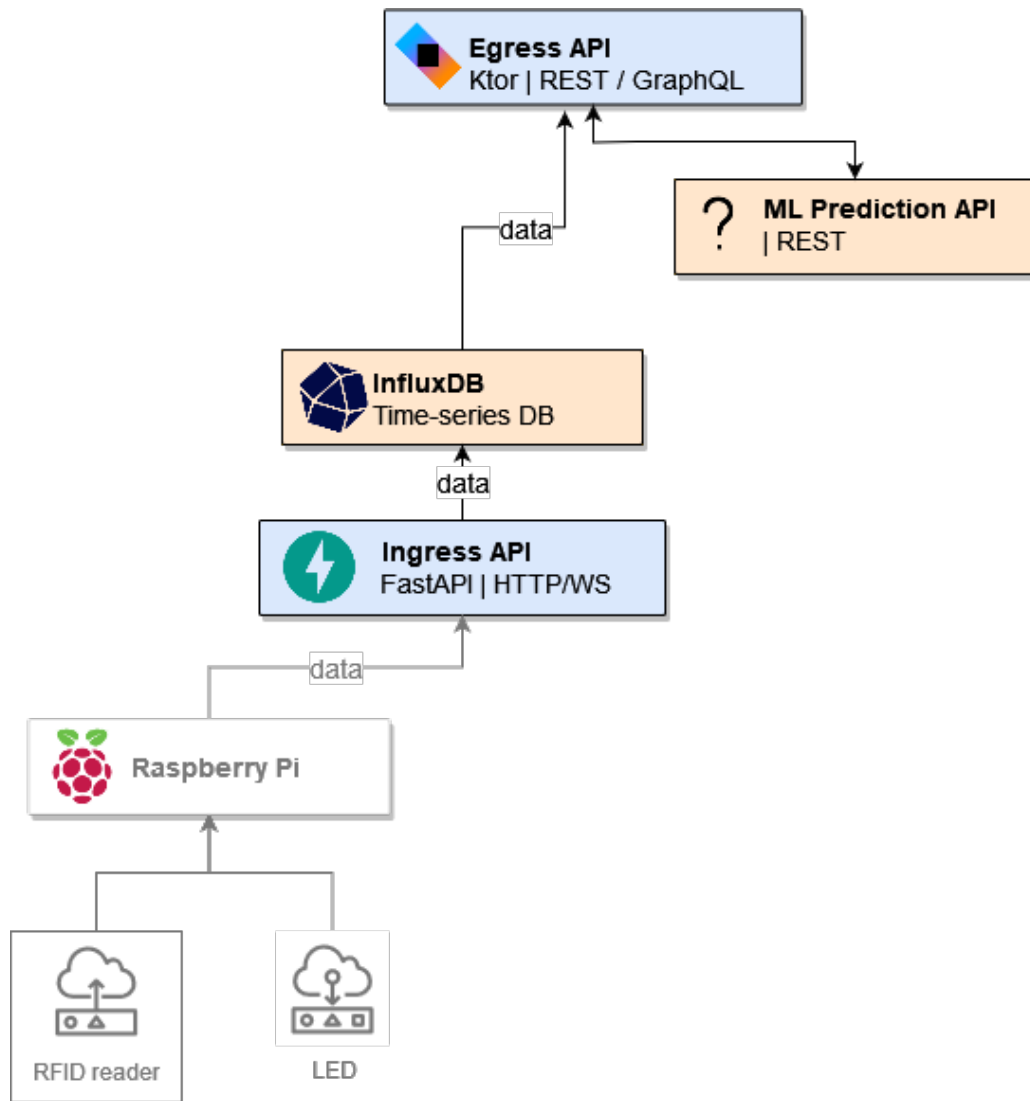


Figure 2: Architecture of this lab session

2 Goals

1. Compose GraphQL queries to get specific data
2. Construct a GraphQL Schema in the GraphQL Schema Definition Language
3. Implement REST and GraphQL APIs
4. Illustrate the differences between REST and GraphQL
5. Interact with a third-party external API
6. Write asynchronous code (coroutines)

3 Preparation: Technologies

3.1 Kotlin and IntelliJ IDEA

Kotlin is a cross-platform, statically typed, general-purpose high-level programming language with type inference. Kotlin is designed to interoperate fully with Java. You have to go through a tutorial to get familiar with Kotlin. Go through this [Kotlin tutorial](#) before you start this lab to get to know the basics. Furthermore, [understanding coroutines](#) is also important in the later tasks, considering making and answering web requests are asynchronous.

The language is created by JetBrains, developers of various IDEs, including PyCharm, Android Studio and IntelliJ IDEA. Both Android Studio and IntelliJ IDEA offer the best Kotlin support as a result, and will be used during the coming lab sessions. For this lab, IntelliJ IDEA will be used to develop the egress application further. For our purposes, the Community Edition suffices, and can be installed following the directions [here](#). Make sure your IDE is up to date!

Make sure a recent JDK is installed and configured. The project requires a JVM version 17 or higher. You can configure the version, as well as download new ones, directly in IntelliJ's settings whilst having the project opened (top left, File - Settings), using the dropdown menu labeled "Gradle JVM".

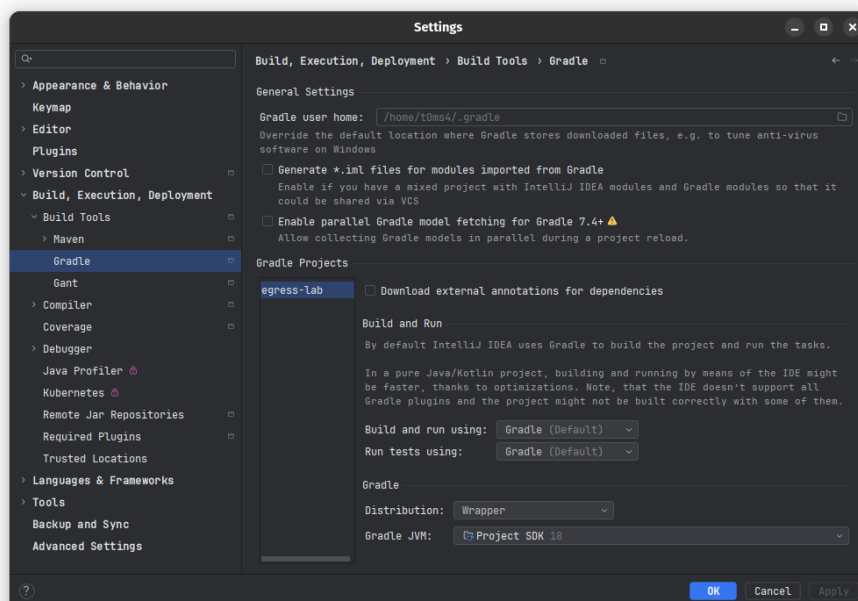


Figure 3: IntelliJ's JVM version settings, at the bottom of the screen. It's recommended to use version 17 and up.

3.2 Ktor

Ktor is a popular framework for building web-based technologies in Kotlin. The framework offers both a client and server implementation, using a similar interface, making it easy to implement logic intended for both project types. The [documentation](#) is a good starting point, considering the Egress API is built using the **server** framework. In later tasks the client

implementation will be used to extend the egress API's functionality.

It is important to be familiar with certain key concepts within Ktor, so make sure you are at least familiar with [routing](#) using the various HTTP verbs (get, post, ...) and the [call object](#) for processing incoming requests before starting with the lab.

3.3 Time

As we're working with time series data, and (most) queries use various time representations throughout the project (epochs, durations, ...), a time library is being used. Here, [kotlinx.datetime](#) is used. This library offers a lot of functionality, with the most notable type used in this application being the `Instant` as it makes interactions with epochs easy. Note that the equivalent Java `datetime` class should **NOT** be used.

3.4 Serialization

Communicating data through web requests requires serialization of this data in a format both the client and server can understand. Here, the REST-side of the egress API uses JSON to represent this data. The conversion process is facilitated using [kotlinx.serialization](#), a powerful serialization library generating all necessary code for the conversion process at compile time. The [introduction found on the project's README](#) should suffice to understand how to use it when creating and receiving JSON responses.

3.5 GraphQL

[GraphQL](#) is a query language for your API. The unique feature of GraphQL is that it allows you to ask for specific data you are interested in, compared to REST. The server composes a response based on your request. With REST, the URL to which you are sending the data determines which data will be sent back. GraphQL queries can fetch a lot of related data in one request, instead of making several roundtrips as one would need in a classic REST architecture.

Try GraphQL in this [demo environment](#) (GraphiQL) to get accustomed to the new language. Click on `Documentation Explorer`, the book at the top left, to reveal the structure of the available data. GraphQL queries typically start with a `"{"` character. Lines that start with a `#` are ignored.

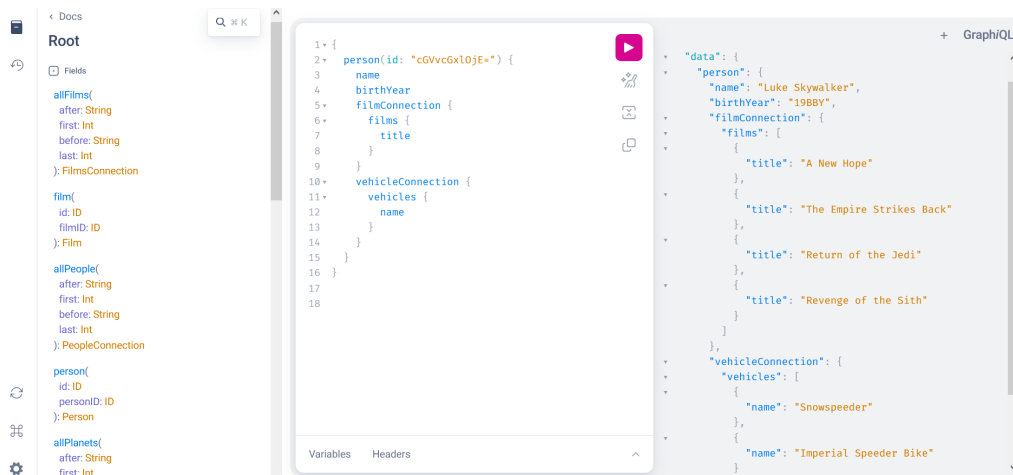


Figure 4: Try GraphQL in a demo environment: GraphiQL.

4 Tutorials

4.1 Application structure

The egress API found on the online learning platform is constructed in a similar fashion as the ingress API, but it's written in Kotlin. In this section we describe the code to bring you up to speed before you have to implement on top of it. **We provide you with Influx integration** in `services/Influx.kt`, which is responsible for setting up the necessary details to talk to our InfluxDB in the cloud. The configuration for the InfluxDB connection has to be filled in with your own credentials. This can be done in its properties (resource) file: `resources/influx2.properties`.

Note that this file reads what is already declared in environment variables. To set up environment variables in IntelliJ associated with your project, you can click on the name of your test or project at the top of the screen, next to the green arrow, and select `Edit Configurations...`, as shown in Figure 5. Then you come to a screen as shown in Figure 6, where you can click to the right of `Environment variables` to enter the values that certain environment variables should take. DO NOT enter the URL or InfluxDB token directly in the properties file, as this file is pushed to a public DockerHub platform where anyone can see these values, and potentially abuse them. You will have to enter the environment variable values for each separate test you want to execute (and tests will be explained more below).

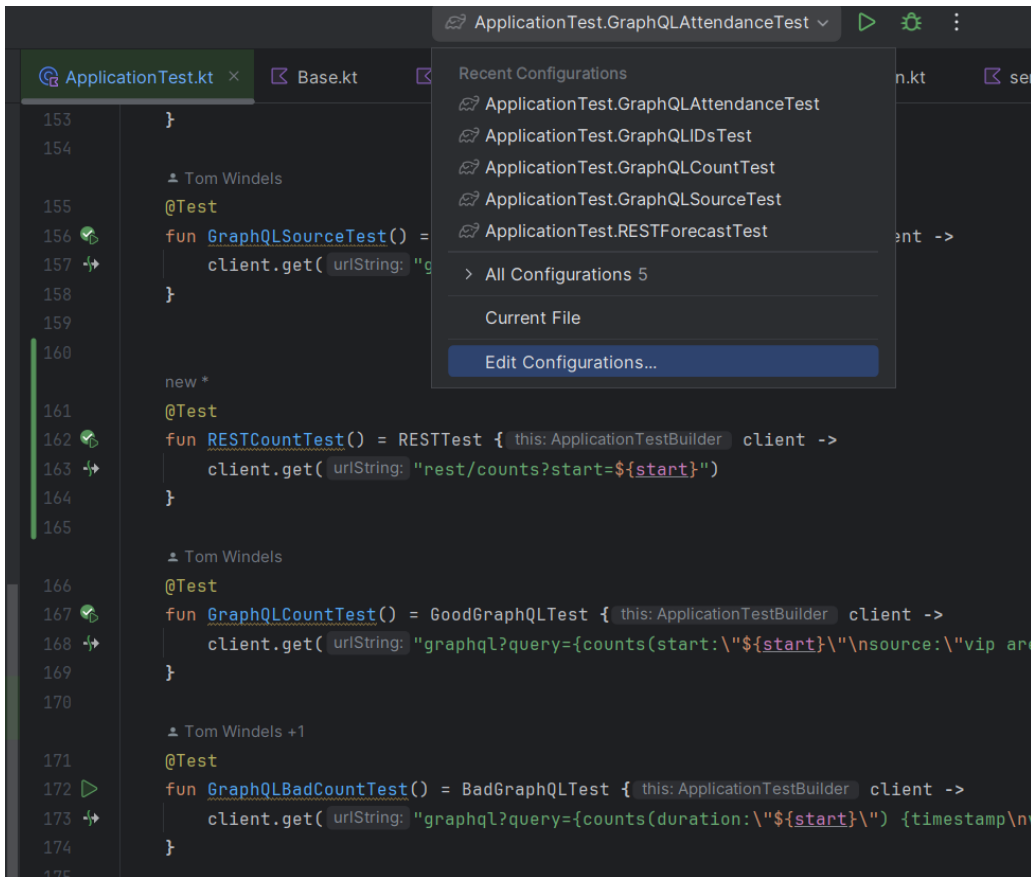


Figure 5: Select Edit Configurations from the project drop-down menu.

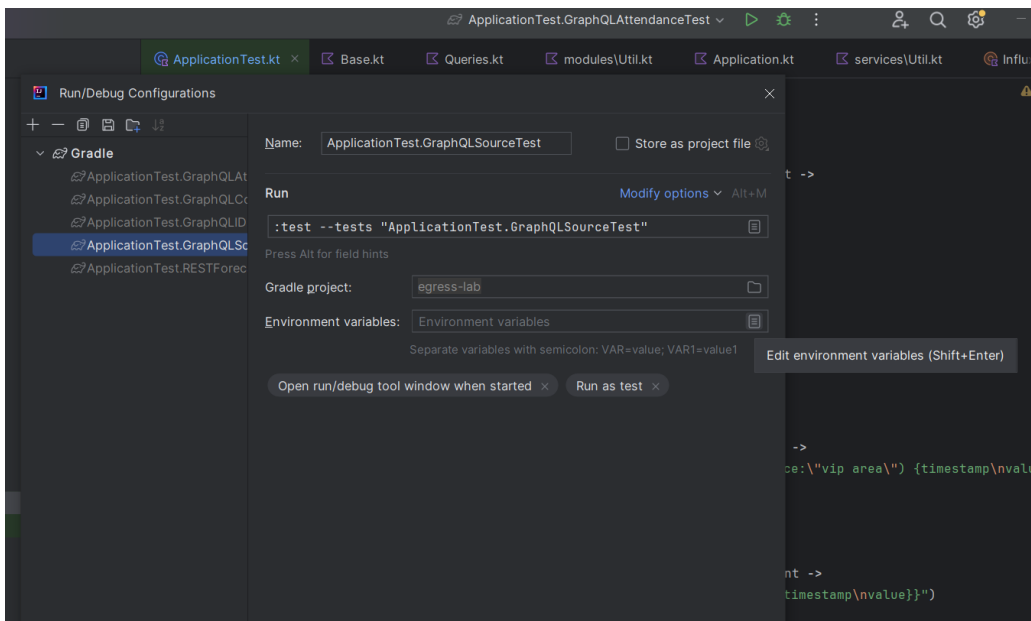


Figure 6: Within edit configurations, you can enter environment variable values in your project.

The egress code base has two modules, namely the REST and GraphQL modules. Each of our modules has a `Module.kt` file that sets up the various endpoints and their routing details. Both REST and GraphQL implementations have separate files for the individual endpoints/queries that you have to implement. The REST module creates different GET routes, while the GraphQL module creates different Query services.

In the folder `egress/src/main/kotlin/server/modules/rest` you will find a number of `.kt` files, each of which will define a different REST endpoint, except for `Module.kt`, which sets up the routing. You will have to implement these endpoints, but we provide the Flux queries for you in `kotlin/server/modules/Queries.kt`.

In Listing 1 we can see one example, the `rest/Counts.kt` file, registering a GET endpoint `counts`. This endpoint has to validate and use the parameters found in the incoming request to respond with the appropriate data. These behaviours are detailed in every endpoint's associated comment. All necessary queries to respond to the various requests are already provided in `modules/Queries.kt`. Transforming the input parameters to values usable with these queries, executing the query on the configured `Influx` instance and returning the result are things you will have to implement yourself. It is very important that the format of what you return matches the specification in the comment of the given function.

```
/** Returns an overview of the changing count values during the provided
 * timestamps `start` and `stop` (epochs in ms) or using default values
 * `Defaults.{start(),stop()}`, optionally filtered by a provided `source`
 * value (tag), and returns them in a JSON structure with layout
 * {
 *     "events": [
 *         {
 *             "timestamp": 0,
 *             "value": 0,
 *             "source": "location"
 *         }, ...
 *     ]
 * }
 */
fun Route.counts() {
    get("counts") {
        TODO()
    }
}
```

Listing 1: `Counts.kt` to retrieve changes to the count using a GET request

In Listing 2 we can see the `modules/graphql/CountsQueryService.kt` file, with class `CountsQueryService` extending the `Query` type, representing a GraphQL service using the library from Expedia Group. This class has one function, `counts`, which takes three Strings as arguments. Similar to their REST counterparts, the comment associated with this function details the expected behaviour of the implementation. Transforming the input parameters to values usable with these queries, executing the query on the configured `Influx` instance and returning the result are all things you will have to implement. Again, the return format is important and will be checked closely.

```

class CountsQueryService: Query {
    /**
     * Returns an overview of the count changes during the provided timestamps
     * [start] and [stop] (epochs in ms) or using default values
     * `Defaults.{start(),stop()}`, optionally filtered by a provided [source]
     * value (tag), and returns them as a list of events with fields for
     * timestamp (ms, string), value (int) and source (string)
     */
    suspend fun counts(
        start: String? = null,
        stop: String? = null,
        source: String? = null
    ) /* TODO: custom return type */ {
        TODO()
    }
}

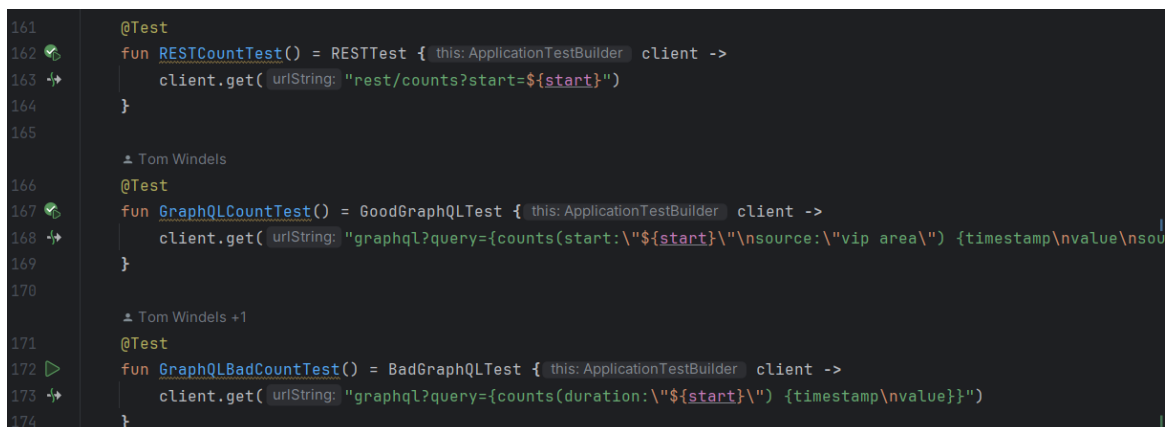
```

Listing 2: CountsQueryService.kt to retrieve changes to the count using a GraphQL query

4.2 Testing

To make validating your implementations easier, testing can be done locally. Tests have been provided for you, making it possible to analyze the responses generated by your implementation directly from the IDE. These tests can be found in `src/test/kotlin/ApplicationTest.kt`. That file adds mocked data to your Influx instance, creates a temporary instance of your egress application and executes the call specific to the test you're running. The tests use a read-write token so they can write mocked testing data to InfluxDB, so make sure `resources/influx2.properties` includes a read-write token. To run each test individually, click on the green arrow to the left of the test in the side bar.

For example, Figure 7 shows three functions, one to test REST's counts endpoint and two to test GraphQL's counts endpoint. A GET operation is performed on an endpoint with some parameters, such as `start` or more complex GraphQL-queries. The test wrappers (`RESTTest {...}`, `GoodGraphQLTest {...}` and `BadGraphQLTest {...}`) execute these GET operations, validating the status code and printing the response body to the console, allowing you to manually review the responses.



```

161  @Test
162  fun RESTCountTest() = RESTTest { this: ApplicationTestBuilder, client ->
163      client.get( urlString: "rest/counts?start=${start}")
164  }
165
166  @Test
167  fun GraphQLCountTest() = GoodGraphQLTest { this: ApplicationTestBuilder, client ->
168      client.get( urlString: "graphql?query={counts(start:\\"${start}\\\"\\nsource:\\"vip area\\") {timestamp\\nvalue\\nsource\\nvalue}}")
169  }
170
171  @Test
172  fun GraphQLBadCountTest() = BadGraphQLTest { this: ApplicationTestBuilder, client ->
173      client.get( urlString: "graphql?query={counts(duration:\\"${start}\\") {timestamp\\nvalue}}")
174  }

```

Figure 7: Excerpt from ApplicationTest.kt to test the count endpoint for both REST and GraphQL. Notice green arrows to execute tests to the left.

Do note that these tests only validate the response code of the tests, and print out the received response. You are responsible for properly analyzing the response itself.

An alternative approach to testing is through the use of [Postman](#). You can run the Egress server application directly on your machine, or host your application in the cloud, and query the various endpoints you have implemented from there. This approach allows you to see the various responses you return using more structured formatting, allowing you to carefully analyze the exact JSON response.

5 Tasks

Don't forget to take screenshots or videos of each task after you get it implemented and working. Your report of Lab5 will only be due after Part 2 of the lab is completed; however, we will check that you have working REST and GraphQL endpoints (tasks 1 and 2), and that your egress is deployed to the cloud (task 3) during the second practical session of this lab.

5.1 Task 1: Implementing REST

In order to visualize the data, an API is needed that exposes this data and only selects the appropriate data to show using queryable endpoints. For this task, the four REST endpoints have to be implemented: `sources`, `counts`, `ids` and `attendance`. `forecast` can be ignored for now. The data each endpoint should return is described in the corresponding comments. Information on how to create responses adhering to these structures can be found in the serialization documentation linked above. Except for `sources`, all endpoints take incoming query parameters as input. These parameters should be respected according to their comments as well. Make sure to use the `kotlinx.datetime` class.

There are REST tests available, querying the endpoints using mocked Influx data, allowing you to validate that the responses adhere to the required layout before continuing. You have to manually validate that the responses you return adhere to the layouts defined in the various endpoint descriptions.

5.2 Task 2: Implementing GraphQL

While the REST endpoints are now implemented, their GraphQL counterparts are still non-functional. These are called `SourcesQueryService`, `CountsQueryService`, `IDQueryService` and `AttendanceQueryService`. The same behaviour as their REST-endpoint alternatives found in Section 5.1 has to be implemented, this time in the form of GraphQL services adhering to the GraphQL convention. The queries and their expected results are described in the corresponding comments.

Before the GraphQL-endpoints become available to consumers of our API, its module has to be enabled. In the application's config file, `src/main/resources/application.yaml`, the GraphQL module has to be enabled again by uncommenting it.

There are GraphQL tests available, querying the services using mocked Influx data, allowing

you to validate that the responses adhere to the required layout before continuing.

5.3 Task 3: Add Egress API to Your Microservices Architecture

In lab 3 (Microservices), you were asked how you would add an egress application that exposes the time-series data from InfluxDB to the cloud, which you will now do. Add the egress application to the architecture such that you can deploy it on the Kubernetes cluster in the cloud (port 8087), similar to ingress. The helm folder already contains most necessary configuration steps, but still requires modification for your specific setup in `helm/values.yaml`. These changes are similar to the ones you made when setting up ingress. Ensure InfluxDB's configuration file (`src/main/resources/influx2.properties`) is properly configured to use the same organisation used with ingress. However, don't change values that are set up to point to environment variables such as:

```
influx2.url=${INFLUX_URL}
```

Pointing these variables to existing environment variables allows us to keep the token and URL secure and easily modifiable when running in the cloud.

To test if your application is working, check if the egress service can be reached in the cloud via a hostname similar to that of the ingress microservice.

```
http://egress.<username>.cloudandmobile.ilabt.imec.be
```

Around the 3rd of April, I will check that you have deployed your egress with working REST and GraphQL endpoints to our cloud infrastructure. This will count towards your final points for this assignment.

5.4 Optional Task 4: Differences between GraphQL and REST

Do not start with this task unless you have done all required tasks! Solving this task correctly is meant for students that want to distinguish themselves.

Despite implementing the same functionality, there certainly are differences between the techniques implemented in Task 1 and Task 2. Time how long it takes to execute a query with both REST and GraphQL and check the response length. This can be done by writing a separate client script that executes the queries on the same endpoint(s) for REST and GraphQL and compares the responses time-wise and length-wise. Illustrate the difference in response length by querying one specific field. Compare your results with what you would expect the differences to be between REST and GraphQL, and explain.

5.5 Task 5: Obtaining forecasts

We don't only want to see past behavior of people scanning RFID tags, arriving and leaving. We would like to see a prediction of future traffic based on past behavior. We will use a 3rd

party machine learning API that takes past counts of RFID tags, and predicts future scan behavior. The final three tasks cover a) accessing this 3rd party API, passing it count behavior obtained from InfluxDB, b) caching those results in our egress, and c) exposing the cached results to egress users. It will be difficult to test these tasks until all three are implemented.

To give new insights to applications consuming the egress API, results obtained from an external forecasting API will be exposed. Before being able to expose this data through an additional endpoint that you have to write (described in Task7), this data has to be available to the egress application. The forecasting API that you will use is documented [here](#), and forecasts future behavior based on RFID count data that you have collected in the past. The logic to get forecasts has to be implemented in `services/forecasting/Model.kt`, in the function called `predict`. Additional comments above this function describe the semantics of the various inputs and possible outputs for this function.

Tip: the `Model` object already has a `client` instance instantiated, configured for use with REST (JSON) applications. Information on how to use this client to query the forecasting API can be found in the `Ktor (client)` documentation.

5.6 Task 6: Scheduling and caching forecasts

Making calls to the forecasting service based on incoming requests from client applications can be a bad solution: the egress response time would depend on the forecasting service response time. A forecasting job system that caches results has to be implemented to fix this. This system is responsible for creating new predictions over time and appending these results to the `ServiceCache`. This cache is then later used to return the results through the egress API in the next and final task.

The job logic has to be implemented in `services/forecasting/ServiceSyncJob.kt`. There, individual `start` and `stop` methods are yet to be implemented, responsible for starting and stopping the corresponding sync job respectively. Every instance of such a job has its own configuration to adhere to, defined by the following parameters: `source`, `method`, `syncDelay` and `begin`. Note that you will be running a job continuously, not part of a specific scope, so you can use the global scope for your implementation. The actual sync implementation relies on calling `Model.predict()` implemented in the prior task. Results obtained by this call are then passed to the cache, as stated in the `ServiceSyncJob` class' comment.

Tip: the [Kotlin coroutines documentation](#) provides plenty of examples on how to manage jobs, delay execution, etc.

Before testing your job implementation, make sure the configuration for the forecasting logic is correctly configured in `resources/forecasting.properties`, using the forecasting documentation as a reference. The number of jobs being created by the egress server varies depending on this configuration.

5.7 Task 7: GETting forecasts

The data found in the cache is currently not accessible to consumer applications, so a new REST endpoint will be implemented in this task. The starting implementation can be found in `rest/Forecast.kt`. Implementation-wise, this task is very similar to task 2, but now interacts with the `ServiceCache` instead of `InfluxDB`, as the results are already processed by the jobs and prediction methods implemented above. Similarly, a comment detailing the expected query inputs and structure of the response is also present.

Make sure you re-deploy the final version of your egress to the cloud infrastructure after all tasks are working.

6 Question(s)

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

7 Summary

Multiple query language methods were implemented to provide the sensor data from the `InfluxDB` database in a structured way. Two methods, REST and GraphQL, were looked into and their differences were illustrated. The data was prepared in such a way that for the upcoming lab sessions, this data could be visualized in a mobile application. Lastly, asynchronous programming was used to create forecasts of our data periodically, and we exposed these forecasts to the outside world as well.

8 Material to submit

You do not have to submit a report or your code and videos until Part2 of this lab is completed. However, as mentioned, we expect that around the second practical session of this lab your egress is deployed to the cloud with working REST and GraphQL endpoints.