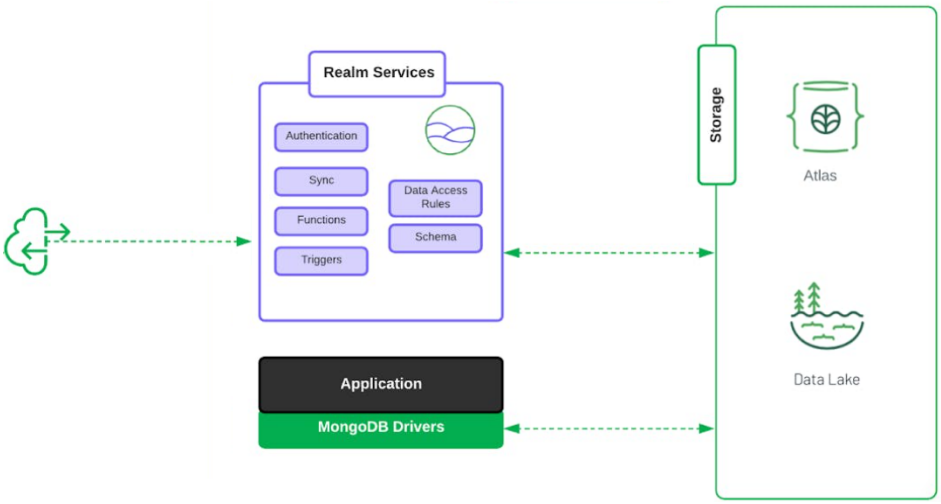
MongoDB Atlas Development



Functions and triggers

There are 3 ways to trigger a function:

- Directly call the function : manually from the console or from another function or from a graphql custom resolver.
- DB trigger
- HTTP endpoint

Directly call the function:

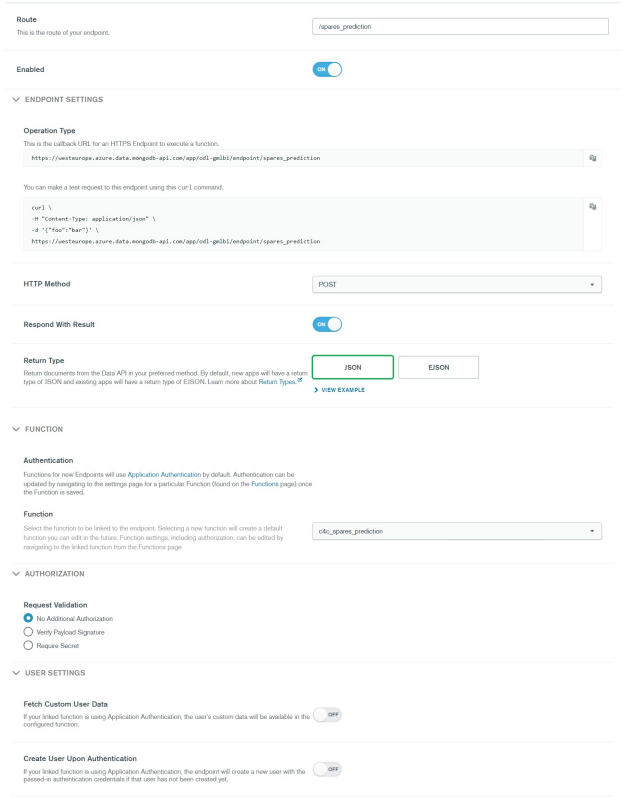
Here is an example of a function that makes an http call to retrieve data from a backend c4c, and then stores the data in a mongodb collection.

This function accepts 3 parameters related to the backend url.

```
exports = async function(base_path,collection,query) {
  const subscription_key = context.values.get("apim-c4c-proxy-key");
  const headers = {
      "Ocp-Apim-Subscription-Key": [subscription_key],
      "Accept": ["Application/json"]
    };
  const response = await context.http.get({
    url: base_path+"/"+collection+"?"+query,
    headers: headers
  });
  data = JSON.parse(response.body.text()).d.results;
  const mongo_collection = context.services.get("mongodb-atlas").db("c4c").collection(collection);
  const data = mongo_collection.insertMany(data)
  .then(result => {
    const { matchedCount, modifiedCount } = result;
    console.log(`Successfully matched ${matchedCount} and modified ${modifiedCount} items.`)
    return result
  })
  .catch(err => console.error(`Failed to update items: ${err}`));
 return data;
};
```

Http endpoints:

To be able to trigger a function through HTTP request, we first need an http endpoint to expose our service



Http endpoint function examples:

Simple example for a function that can accept an http trigger.

The function should accept 2 parameters: request, response.

```
exports = async function (request, response) {
  try {
    // 1. Parse data from the incoming request
    if(request.body === undefined) {
      throw new Error(`Request body was not defined.`)
    }
    const body = JSON.parse(request.body.text());
   return logic(body)
}catch (error) {
  response.setStatusCode(400);
  response.setBody(error.message);
}
    function logic(body) {
      return body;
    }
}
```

Another example enabled for HTTP POST to store an object from the body in a mongodb collection:

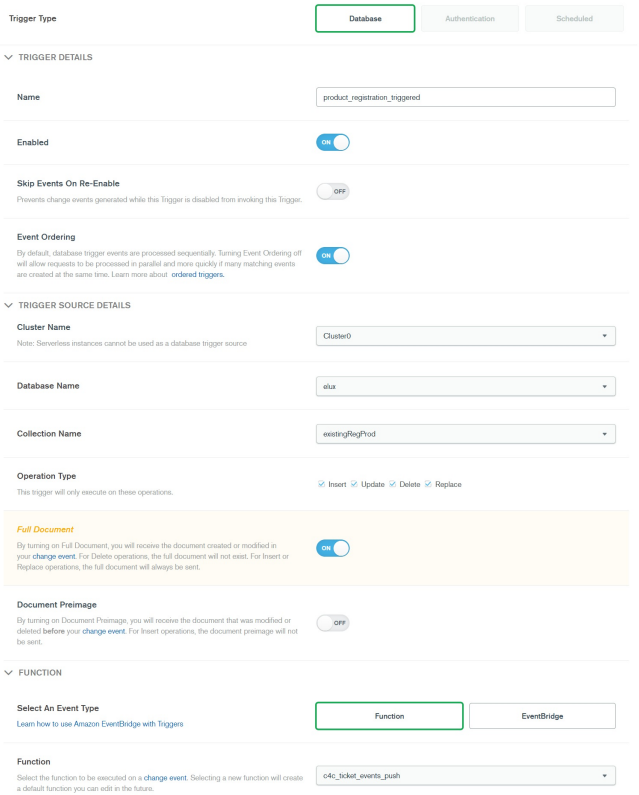
```
exports = async function (request, response) {
  try {
    // 1. Parse data from the incoming request
    if(request.body === undefined) {
      throw new Error(`Request body was not defined.`)
    }
    const body = JSON.parse(request.body.text());
    // 2. Insert 1 doc in collection
    const { insertedId } = await context.services
      .get("mongodb-atlas")
      .db("sbs-extensions")
      .collection("c4c-ticket-events")
      .insertOne(body);
    // 3. Configure the response
    response.setStatusCode(201);
    // tip: You can also use EJSON.stringify instead of JSON.stringify.
    response.setBody(JSON.stringify({
       message: "Successfully saved the request body",
       insertedId,
    }));
  } catch (error) {
    response.setStatusCode(400);
    response.setBody(error.message);
  }
};
```

Triggers:

Apart from the http endpoint feature, there are 3 different types of triggers: Database, Authorization and Scheduled.

DB trigger:

An example configuration for the Database Trigger, and as you see marked in red we have enabled Event Ordering and this will enable an event queue for the function triggers.



Example database trigger function:

And this is an example function that works with the EventChanged triggers, it will post a ticket payload to c4c using the apim proxy.

This function accepts only 1 parameter: changeEvent.

```
exports = async function (changeEvent) {
  // Destructure out fields from the change stream event object
  const { updateDescription, fullDocument } = changeEvent;
  // Prepare request params
  const subscription_key = context.values.get("apim-c4c-proxy-key");
  const url = "https://api-staging.eluxmkt.com/external/sap-integration/c4c/ServiceRequestCollection";
  const headers = {
      "Ocp-Apim-Subscription-Key": [subscription_key],
      "Accept": ["Application/json"],
      "Content-Type": ["Application/json"]
    };
  const body= {"LX_TKT_DGCLMNR":"PL22042183","Name":"Uszkodzenie mechaniczne","RequestInitialReceiptdatetimecontent":"2022-06-04T08:19:14+00:00","LX_TCK_CVRG":"ZP","ActivityServiceIssueCategoryID":"1.1.1.1.17","BuyerPartyID":"5586364","ProcessorPartyID":"6891586","LX_PRD_CH1_KUT":"504","ServiceRequestUserLifeCycleStatusCode":"Z6","ProductID":"910280596","LX_PRD_MI":"EEWA7700R","LX_PRD_BN_KUT":"Electrolux","LX_PRD_SN_KUT":"03456567","LX_PRD_ML_KUT":"00","LX_PRD_PRDT_KUT":"2021-01-01T00:00:00","LX_PRD_RET":"","WarrantyID":"2","SerialID":"2022_2245984","ServiceRequestServicePointLocation":{"ServiceRequestServicePointLocationAddress":{"Street":"KAROLKOWA 30","City":"WARSZAWA","PostalCode":"01-207","Country":"PL"}}};
  // const body= JSON.parse(fullDocument);
  const response = await context.http.post({
```

```
    url: url,
    headers: headers,
    body: body.toString()
  }).then(result => {
     const { matchedCount, modifiedCount } = result;
     console.log(`Successfully matched ${matchedCount} and modified ${modifiedCount} items.`)
     return result
    })
    .catch(err => console.error(`Failed to update items: ${err}`));
  return response;
};
```

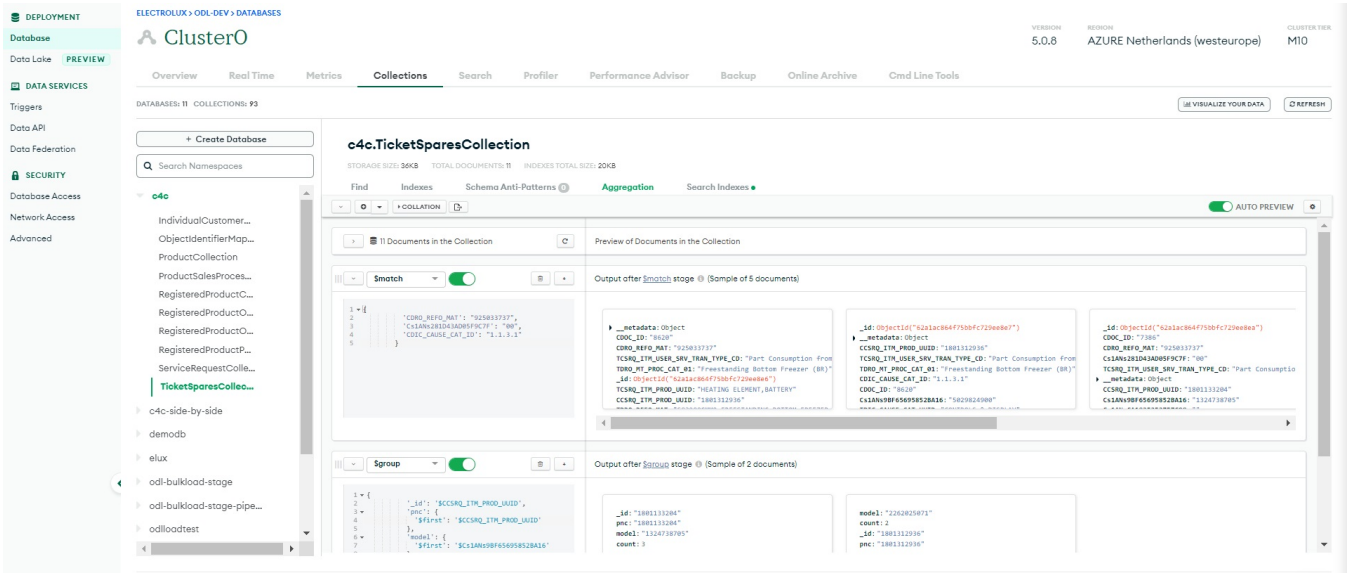
Aggregation pipelines:

Another good feature is the aggregation pipeline

The aggregation pipeline refers to a specific flow of operations that processes, transforms, and returns results. In a pipeline, successive operations are informed by the previous result.

Let’s take a typical pipeline:

Input -> $match -> $group -> $sort -> output



Once the aggregation pipeline ready, you can export it using the export button to any language snippet you need.

Example for JS:

```
[
  {
    '$match': {
      'CDRO_REFO_MAT': '925033737',
      'Cs1ANs281D43AD05F9C7F': '00',
```

```
      'CDIC_CAUSE_CAT_ID': '1.1.3.1'
    }
  }, {
    '$group': {
      '_id': '$CCSRQ_ITM_PROD_UUID',
      'pnc': {
        '$first': '$CCSRQ_ITM_PROD_UUID'
      },
      'model': {
        '$first': '$Cs1ANs9BF65695852BA16'
      },
      'count': {
        '$sum': 1
      }
    }
  }
]
---
```

And here is an example implemented in a nodeJS function:

```
---
exports = async function (request, response) {
  try {
    // 1. Parse data from the incoming request
    if (request.body === undefined) {
      throw new Error(`Request body was not defined.`)
    }
    const body = JSON.parse(request.body.text());
    // 2. Handle the request
    const collection = await context.services
      .get("mongodb-atlas")
      .db("c4c")
      .collection("TicketSparesCollection");
    const pipeline = [
      {
        '$match': {
          'CDRO_REFO_MAT': `${body.pnc}`,
          'Cs1ANs281D43AD05F9C7F': `${body.ml}`,
          'CDIC_CAUSE_CAT_ID': `${body.rfc}`
```

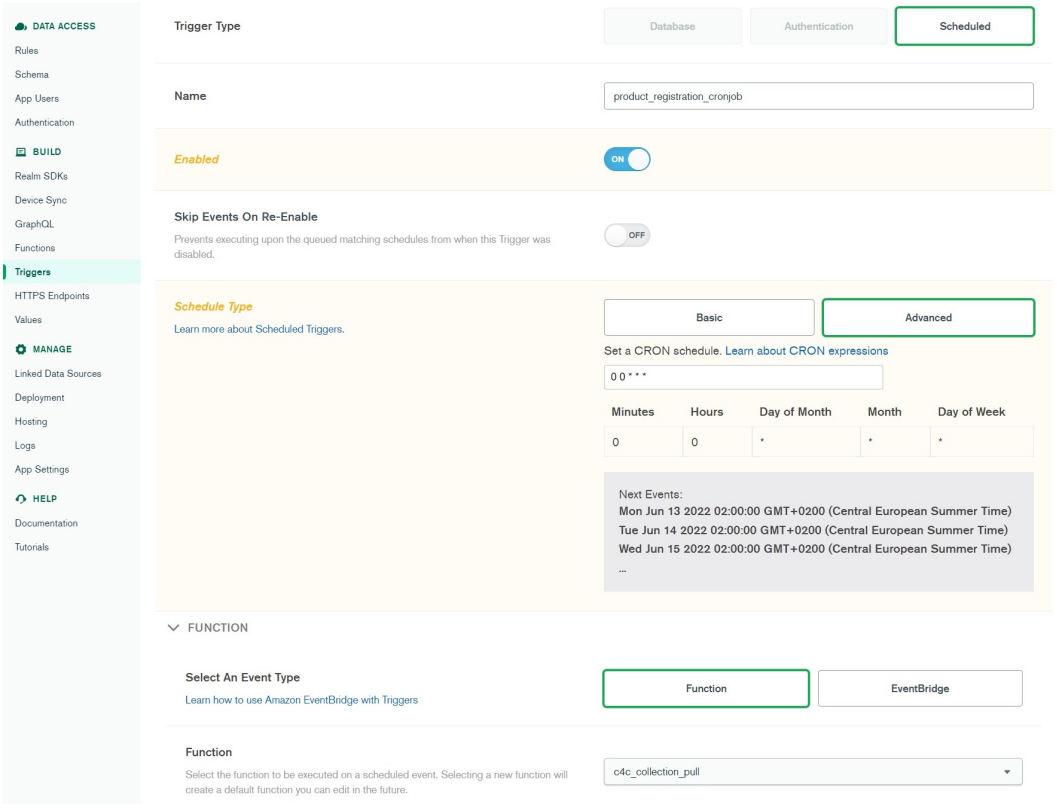
```
        }
      }, {
        '$group': {
          '_id': '$CCSRQ_ITM_PROD_UUID',
          'pnc': {
            '$first': '$CCSRQ_ITM_PROD_UUID'
          },
          'model': {
            '$first': '$Cs1ANs9BF65695852BA16'
          },
          'count': {
            '$sum': 1
          }
        }
      }
    ]
    return collection.aggregate(pipeline).toArray()
      .then(response => {
        response.forEach(object => {
          delete object['_id'];
        });
        return response
      })
      .catch(err => console.error(`Failed: ${err}`))
    // 3. Configure the response
    response.setStatusCode(201);
    response.setBody(JSON.stringify({ response }));
    console.log(JSON.stringify({ response }))
  } catch (error) {
    response.setStatusCode(400);
    response.setBody(error.message);
  }
}
```

Once We have our functions ready to be syncing data with the backend(s)

Using cronjobs or c4c events etc.

Cron jobs:

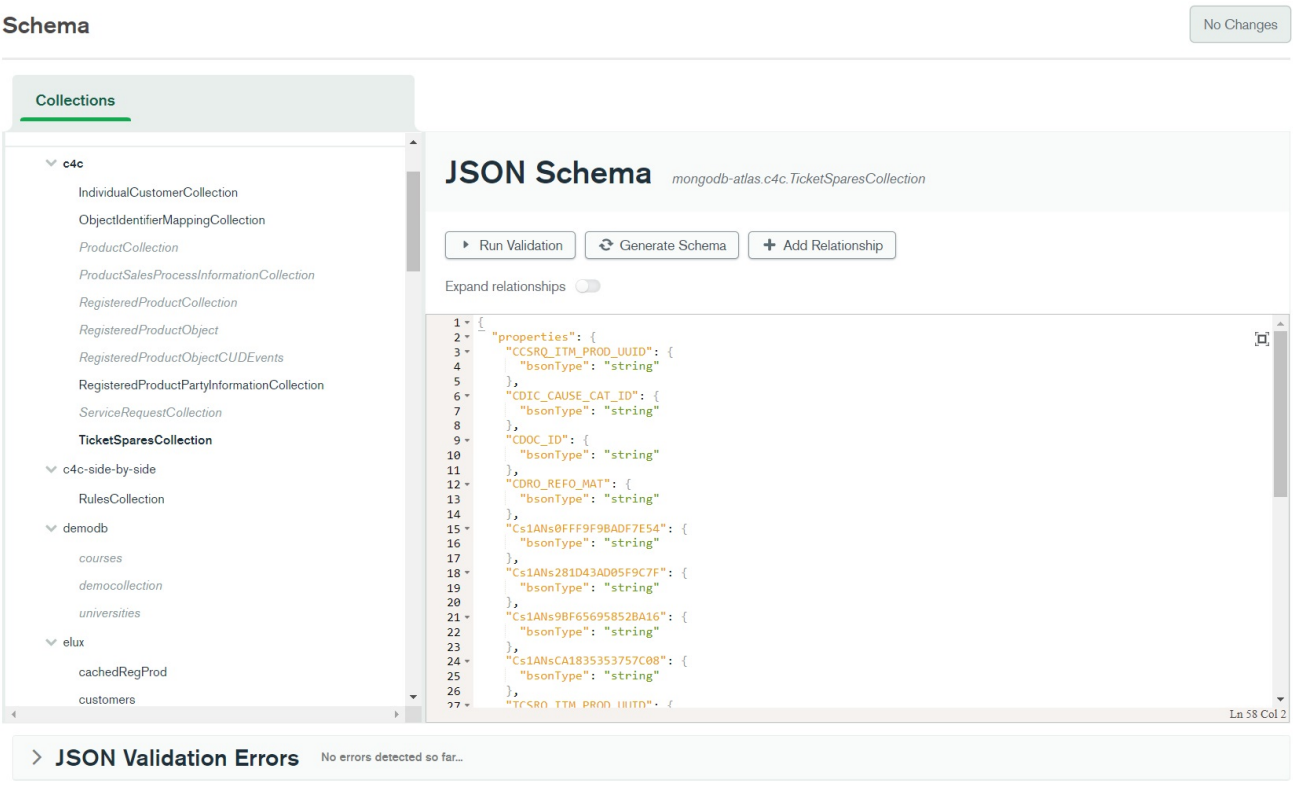
Here below is the configuration for a daily job using the c4c_collection_pull function



…

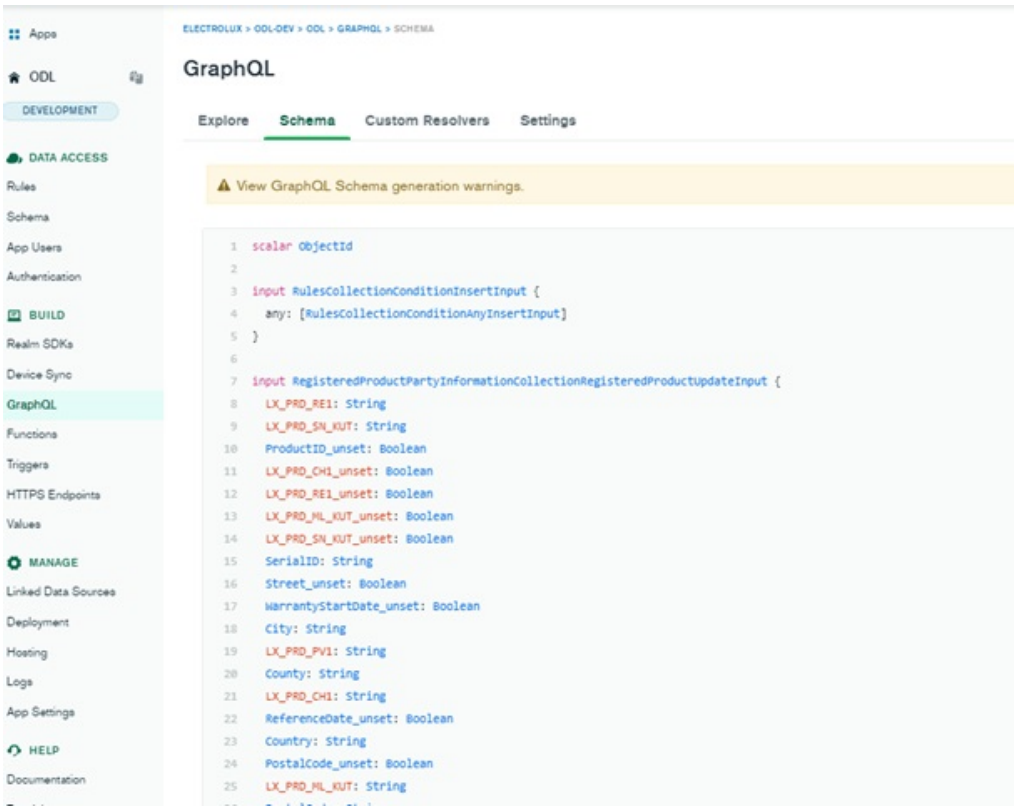
Schemas:

We can move to the next step which will be to enforce the schemas we are using, and on this screen below you can select a collection and generate the schema from a sample data you choose.



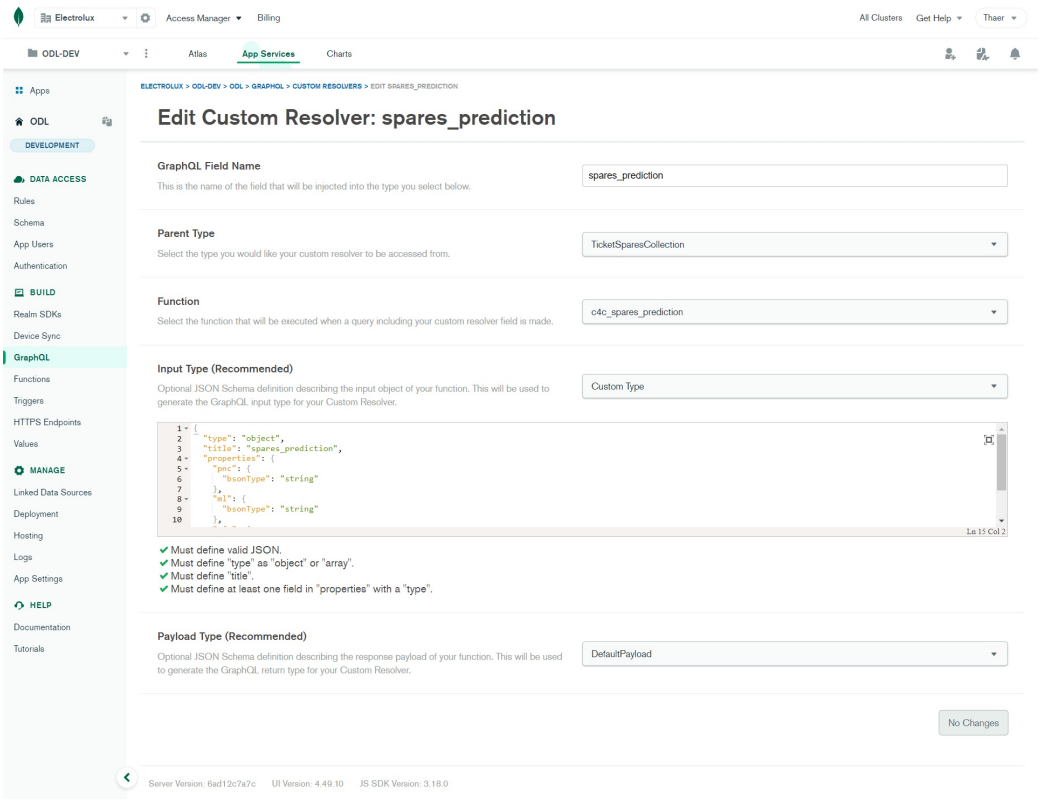
GraphQL:

And this will allow us to generate the graphQL schemas that can be used to expose our services in a different way, allowing the consumer channel to only select the data they need and also to aggregate requests and reduce network traffic and chattiness between services.



Adding to those capabilities we can create Custom resolvers and basically expose extra operations using functions, something similar to the HTTP endpoints.

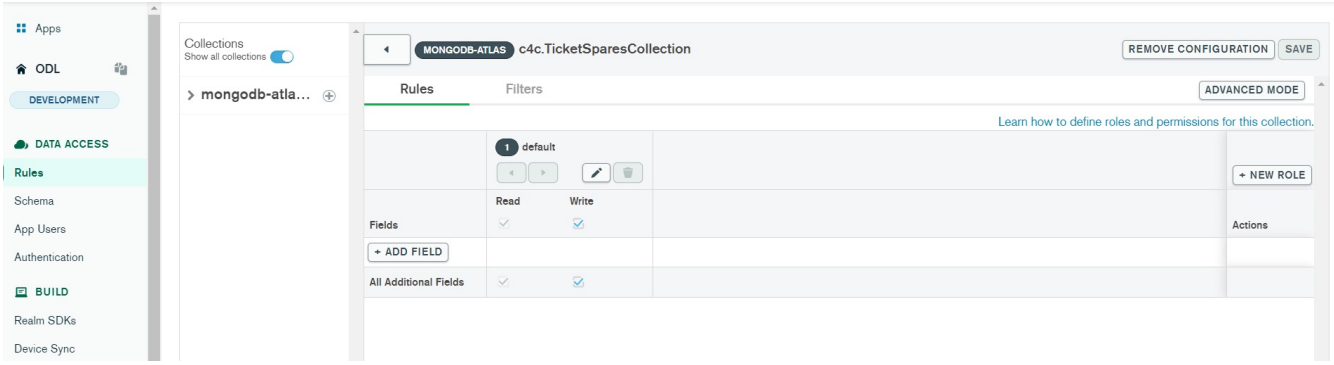
And the configuration for the custom resolver will look something like here below:



This is very good because not only we will expose endpoints to do crud operations across all the schemas we define but also to add extra endpoints for custom logic.

Access rules:

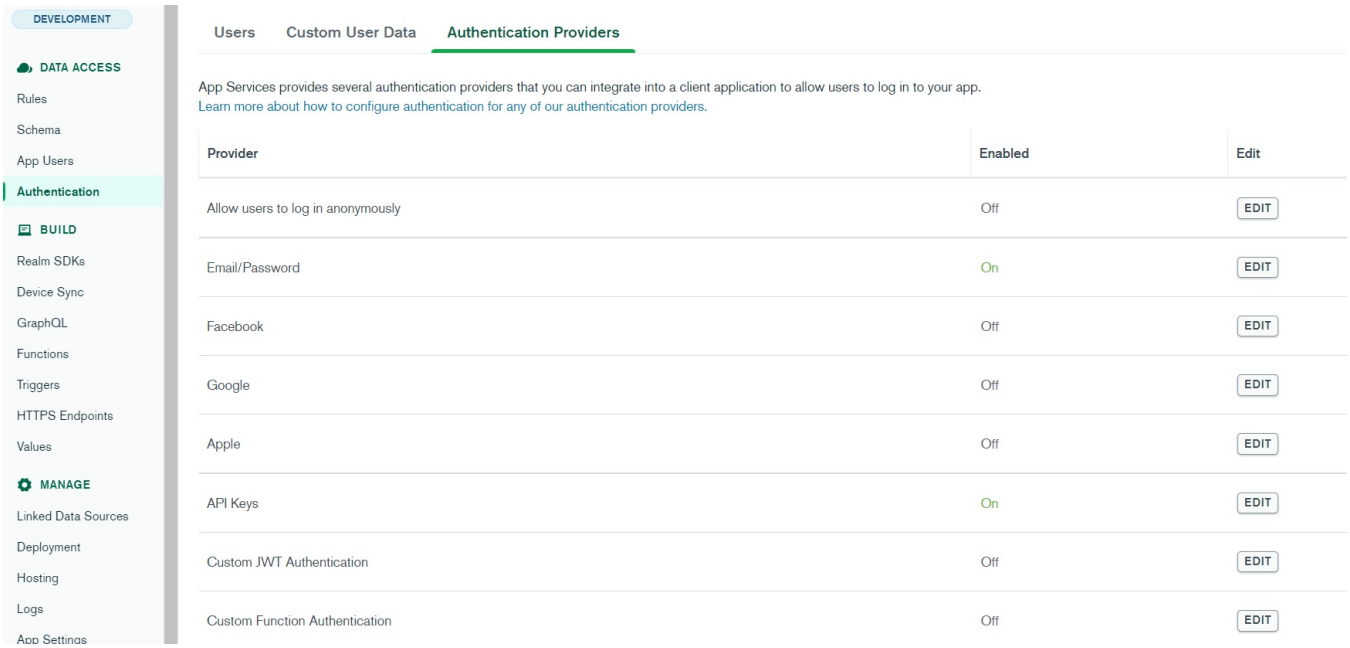
Now that we have a way to use all the services we would need to create Rules for access to the different collections and all those rules will be inherited to the API key we will be using.



Authorization:

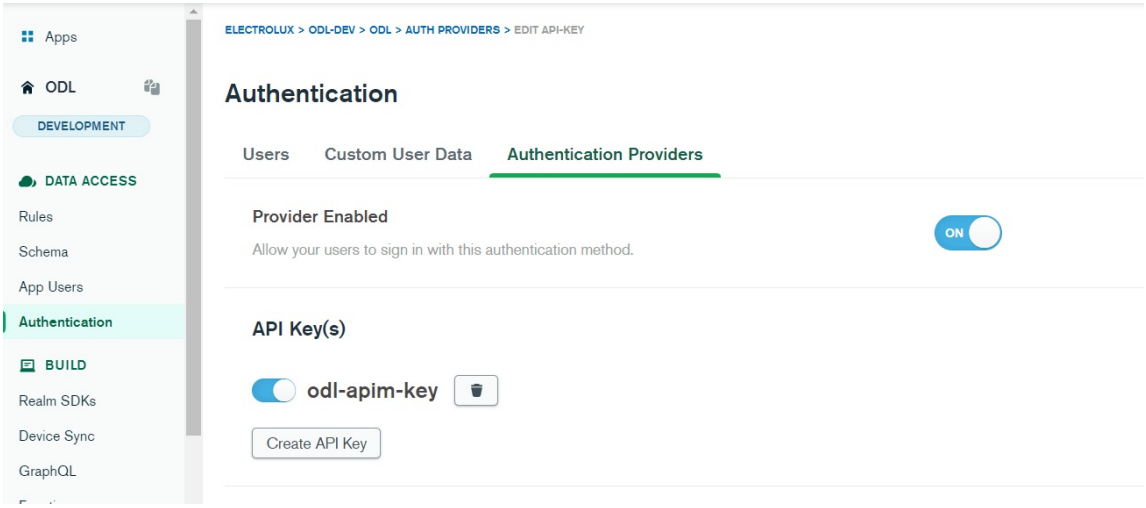
And then enable a way for Authorization.

Example here is to use API_KEY but there are a lot of other options as you see below:



Api_keys:

Once API_Key enabled, We can create a key like below:



Http endpoint example:

Once we have this in place we can use an HTTP endpoint as follows:

Request :

```
curl --location --request POST 'https://westeurope.azure.data.mongodb-api.com/app/odl-gmlbi/endpoint/spares_prediction' \
--header 'api-key: xxx' \
--header 'Content-Type: application/json' \
--data-raw '{
            "pnc": "933014286",
            "ml": "12",
           "rfc": "1.21.1.1"
}'
```

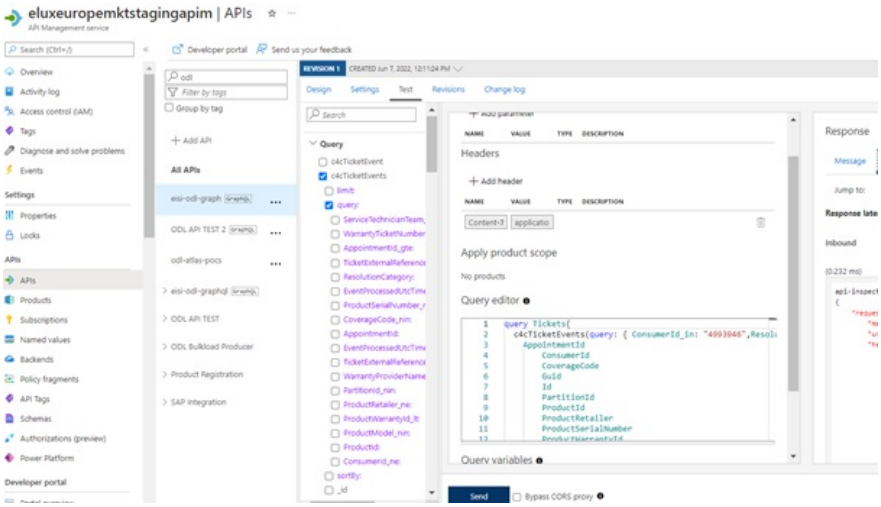
Response:

[{"_id":"1801133204","pnc":"1801133204","model":"1324738705","count":3},{"_id":"1801312936","pnc":"1801312936","model":"2262025071","count":2}]

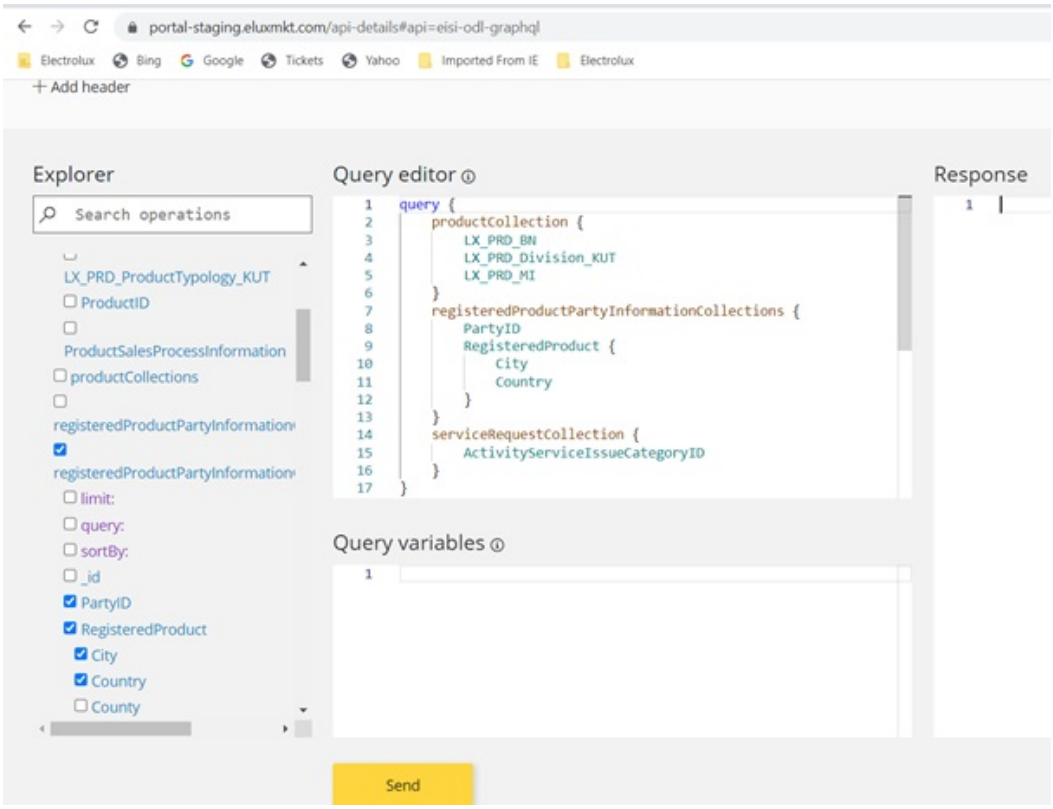
MongoDB graph on APIM:

And also we can setup the graphQL endpoint on API management and we can have something similar to this on the developer portal for all our teams to use.

This view can be used by the PE team to identify common patterns to be exposed as custom resolvers and other



And this will be used by the different channel teams to retrieve data and test their queries from the portal.



Example graphQL request to apim endpoint:

----

curl --location --request POST 'https://api-staging.eluxmkt.com/external/odl/graph'

--header 'Content-Type: application/json'

--header 'Access-Control-Request-Headers: *'

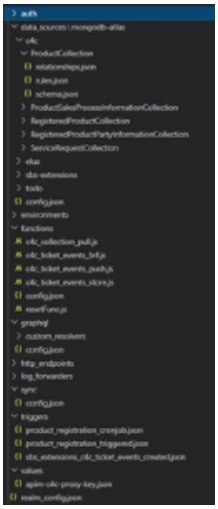
--header 'version: v1'

--header 'Ocp-Apim-Subscription-Key: xxx'

--data-raw '{"query":"query { cachedRegProd { address { city country county line1 postcode } consumerCountry consumerId hasExtendedWarranty id IncentivesOnlineRegistration_SDK product { maintenanceLevel productCode serialNumber } purchaseDate receipt { fileName type } registrationDate sourceApplication storeName } } ","variables":{}}'

Git

And all our solution will be having a similar structure:



And the repo can be pulled and updated using the mongo cli tool

https://www.mongodb.com/docs/atlas/app-services/manage-apps/deploy/manual/deploy-cli/

Or automatically using Github integration.