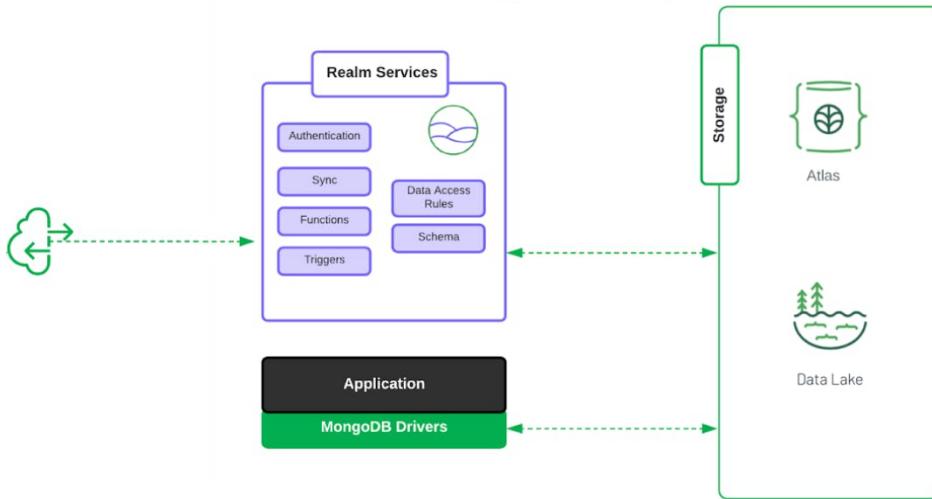


MongoDB Atlas Development



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

Route
This is the route of your endpoint.

Enabled

ENDPOINT SETTINGS

Operation Type
This is the callback URL for an HTTPS Endpoint to execute a function.

You can make a test request to the endpoint using this curl command.

```
curl -X POST -H "Content-Type: application/json" -d '{"foo": "bar"}' https://westeurope.azure.data.mongodb-api.com/app/cdl-gmhb/endpoint/spares_prediction
```

HTTP Method

Respond With Result

Return Type
Return documents from the Data API in your preferred method. By default, new apps will have a return type of JSON and existing apps will have a return type of EJSON. Learn more about Return Types.

JSON EJSON

FUNCTION

Authentication
Functions for new Endpoints will use Application Authentication by default. Authentication can be updated by navigating to the settings page for a particular Function (found on the Functions page) once the Function is saved.

Function
Select the function to be linked to the endpoint. Selecting a new function will create a default function you can edit in the future. Function settings, including authorization, can be edited by navigating to the linked function from the Functions page.

AUTHORIZATION

Request Validation
 No Additional Authorization
 Verify Payload Signature
 Reparse Signature

USER SETTINGS

Fetch Custom User Data
If your linked function is using Application Authentication, the user's custom data will be available in the configured function.

Create User Upon Authentication
If your linked function is using Application Authentication, the endpoint will create a new user with the passed-in authentication credentials if that user has not been created yet.

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.statusCode(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","RequestInitialReceiptdateimecontent":"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({
    method: "POST",
    url: url,
    headers: headers,
    body: body
  });
  return response;
}
```

```

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

The screenshot shows the Azure Cosmos DB portal interface. On the left, there's a sidebar with 'DEPLOYMENT' (selected), 'Database' (selected), 'Data Lake' (disabled), 'DATA SERVICES', 'Triggers', 'Data API', 'Data Federation', 'SECURITY', 'Database Access', 'Network Access', and 'Advanced'. Under 'Database', there's a 'PREVIEW' tab. The main area shows 'Cluster0' with 'OVERVIEW', 'REAL TIME', 'METRICS', 'COLLECTIONS' (selected), 'SEARCH', 'PROFILER', 'PERFORMANCE ADVISOR', 'BACKUP', 'ONLINE ARCHIVE', and 'CMD LINE TOOLS'. The 'VERSION' is 5.0.8, 'REGION' is AZURE Netherlands (westeurope), and 'CLUSTER TIER' is M10. The 'COLLECTIONS' tab shows 'c4c.TicketSparesCollection' with '11 DOCUMENTS IN THE COLLECTION' and 'INDEXES TOTAL SIZE: 20KB'. Below this, there are tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation' (selected), and 'Search Indexes'. The 'Aggregation' tab has two sections: 'Output after \$match stage (Sample of 5 documents)' and 'Output after \$group stage (Sample of 2 documents)'. The first section shows five documents matching the \$match criteria. The second section shows two groups from the \$group stage, each with a sample of documents.

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

Example for JS:

```

---
[{
  '$match': {
    'CDRO_REF0_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 = [
{
'$match': {
'CDRO_REF0_MAT': `${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.statusCode(201);
response.setBody(JSON.stringify({ response }));
console.log(JSON.stringify({ response }));

} catch (error) {
  response.statusCode(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

DATA ACCESS
Trigger Type

Database
Authentication
Scheduled

Name

Enabled

Skip Events On Re-Enable Prevents executing upon the queued matching schedules from when this Trigger was disabled.

Schedule Type Basic Advanced Advanced

Set a CRON schedule. [Learn about CRON expressions](#)

0 0 * * *

Minutes	Hours	Day of Month	Month	Day of Week
0	0	*	*	*

Next Events:
 Mon Jun 13 2022 02:00:00 GMT+0200 (Central European Summer Time)
 Tue Jun 14 2022 02:00:00 GMT+0200 (Central European Summer Time)
 Wed Jun 15 2022 02:00:00 GMT+0200 (Central European Summer Time)
 ...

FUNCTION

Select An Event Type Function EventBridge Function

Learn how to use Amazon EventBridge with Triggers

Function c4c_collection_pull

Select the function to be executed on a scheduled event. Selecting a new function will create a default function you can edit in the future.

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.

Schema

No Changes

Collections

- c4c
 - IndividualCustomerCollection
 - ObjectIdentifierMappingCollection
 - ProductCollection
 - ProductSalesProcessInformationCollection
 - RegisteredProductCollection
 - RegisteredProductObject
 - RegisteredProductObjectCUDEvents
 - RegisteredProductPartyInformationCollection
 - ServiceRequestCollection
 - TicketSparesCollection
- c4c-side-by-side
 - RulesCollection
- demodb
 - courses
 - democollection
 - universities
- elux
 - cachedRegProd
 - customers

JSON Schema

mongodb-atlas.c4c.TicketSparesCollection

Run Validation Generate Schema Add Relationship

Expand relationships

```
1 {  
2   "properties": {  
3     "CCSRQ_ITM_PROD_UID": {  
4       "bsonType": "string"  
5     },  
6     "CDIC_CAUSE_CAT_ID": {  
7       "bsonType": "string"  
8     },  
9     "CDOC_ID": {  
10      "bsonType": "string"  
11    },  
12     "CDR0_REF0_MATT": {  
13       "bsonType": "string"  
14     },  
15     "Cs1ANs0FFF9f98ADF7E54": {  
16       "bsonType": "string"  
17     },  
18     "Cs1ANs281D43AD05F9C7F": {  
19       "bsonType": "string"  
20     },  
21     "Cs1ANs9BF65695852BA16": {  
22       "bsonType": "string"  
23     },  
24     "Cs1ANsCA1835353757C08": {  
25       "bsonType": "string"  
26     },  
27     "TicketSparesCollection": {}  
}
```

Ln 58 Col 2

> JSON Validation Errors No errors detected so far...

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.

The screenshot shows the AWS AppSync console interface. The left sidebar has sections for App, ODL, Development, Data Access (Rules, Schema), Build (Realm SDKs, Device Sync, GraphQL, Functions, Triggers, HTTPS Endpoints, Values), Manage (Linked Data Sources, Deployment, Hosting, Logs, App Settings), and Help (Documentation). The 'GraphQL' section under 'Development' is selected. The main area is titled 'GraphQL' and has tabs for Explore, Schema (which is selected), Custom Resolvers, and Settings. A yellow banner at the top says 'View GraphQL Schema generation warnings.' Below it is a code editor showing the following GraphQL schema:

```
1 scalar ObjectId
2
3 input RulesCollectionConditionInsertInput {
4   any: [RulesCollectionConditionAnyInsertInput]
5 }
6
7 input RegisteredProductPartyInformationCollectionRegisteredProductupdateInput {
8   LX_PROD_REL1: String
9   LX_PROD_SN_UNSET: String
10  ProductID_unset: Boolean
11  LX_PROD_CH1_unset: Boolean
12  LX_PROD_REL1_unset: Boolean
13  LX_PROD_ML_KUT_unset: Boolean
14  LX_PROD_SN_UNSET_unset: Boolean
15  SerialID: String
16  Street_unset: Boolean
17  WarrantyStartDate_unset: Boolean
18  City: String
19  LX_PROD_PVI: string
20  County: String
21  LX_PROD_CH1: String
22  ReferenceDate_unset: Boolean
23  Country: String
24  PostalCode_unset: Boolean
25  LX_PROD_ML_KUT: String
```

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:

Edit Custom Resolver: spares_prediction

GraphQL Field Name
This is the name of the field that will be injected into the type you select below.

Parent Type
Select the type you would like your custom resolver to be accessed from.

Function
Select the function that will be executed when a query including your custom resolver field is made.

Input Type (Recommended)
Optional JSON Schema definition describing the input object of your function. This will be used to generate the GraphQL input type for your Custom Resolver.

```

1< [
2   "type": "object",
3   "title": "spares_prediction",
4   "properties": {
5     "type": {
6       "bsonType": "string"
7     },
8     "ml": {
9       "bsonType": "string"
10    }
}

```

Line 15 Col 2

Payload Type (Recommended)
Optional JSON Schema definition describing the response payload of your function. This will be used to generate the GraphQL return type for your Custom Resolver.

No Changes

Server Version: 6.6d12c7a7c UI Version: 4.49.10 JS SDK Version: 3.18.0

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.

MONGODB-ATLAS c4c.TicketSparesCollection

Rules **Filters**

	default	
Fields	Read	Write
+ ADD FIELD		
All Additional Fields		

Learn how to define roles and permissions for this collection.

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:

The screenshot shows the 'Authentication Providers' section of the App Services interface. On the left, a sidebar lists categories like Development, Data Access, Build, and Manage. The 'Authentication' section is selected. The main area displays a table of authentication providers:

Provider	Enabled	Edit
Allow users to log in anonymously	Off	<button>EDIT</button>
Email/Password	On	<button>EDIT</button>
Facebook	Off	<button>EDIT</button>
Google	Off	<button>EDIT</button>
Apple	Off	<button>EDIT</button>
API Keys	On	<button>EDIT</button>
Custom JWT Authentication	Off	<button>EDIT</button>
Custom Function Authentication	Off	<button>EDIT</button>

Api_keys:

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

The screenshot shows the 'Edit API-Key' page. The sidebar includes 'Apps', 'ODL', 'Development', 'Data Access', 'Build', and 'Authentication'. The 'Authentication' section is selected. The main area shows a table for API keys:

Provider Enabled
Allow your users to sign in with this authentication method. <input checked="" type="checkbox"/> ON

Below this, there's a section for 'API Key(s)' with a toggle switch labeled 'odl-apim-key' and a trash icon, along with a 'Create API Key' button.

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

The screenshot shows the API Management developer portal interface. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, APIs, Products, Subscriptions, Named values, Backends, Policy fragments, API Tags, Schemas, Authorizations (preview), and Power Platform. The main area is titled 'eluxeuropemktstagingapim | APIs' and shows a list of APIs under 'All APIs'. One API, 'eisi-odl-graph (schema)', is selected. The right side has tabs for Design, Settings, Test, Revisions, and Change log. Under the 'Test' tab, there's a 'Query editor' section with a code snippet:

```
query {
  queryTickets(
    c4cTicketEvents(query: { ConsumerId_In: "4993946" },Results)
  ) {
    ConsumerId
    ConsumerCode
    Guid
    Id
    PartitionId
    ProductId
    ProductRetailer
    ProductSerialNumber
    ProductWarehouseId
  }
}
```

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

The screenshot shows the developer portal with a browser address bar pointing to 'portal-staging.eluxmkt.com/api-details#api=eisi-odl-graphql'. The interface includes an 'Explorer' sidebar with a search bar and a tree view of operations like 'LX_PRD_ProductTypology_KUT' and 'registeredProductPartyInformation'. The main area has sections for 'Query editor', 'Query variables', and 'Response'. The 'Query editor' contains the same GraphQL query as the previous screenshot:

```
query {
  queryTickets(
    c4cTicketEvents(query: { ConsumerId_In: "4993946" },Results)
  ) {
    ConsumerId
    ConsumerCode
    Guid
    Id
    PartitionId
    ProductId
    ProductRetailer
    ProductSerialNumber
    ProductWarehouseId
  }
}
```

The 'Response' section shows a single line of JSON output:

```
1
```

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.