

Introduction to



Hands-On Workshop

Part 2 - Stitch

Overview

In Part 1 of this workshop you've set the foundation by creating a MongoDB cluster and loading some data. Now it's time to put that data to action. In part 2 of this workshop we'll create microservices to expose the data via REST APIs and create a basic front-end application that leverages those APIs.

Specifically, we'll create APIs to query and add new restaurants. When new restaurants are added we'll create a trigger to notify the health inspector. And we'll host all of this on MongoDB Stitch!

Video Learning Resources

Video Walk-Through: <https://vimeo.com/328353427>.

Additional video resources for learning: <https://hub.mongodb.com/femisphere-codeswitch>

Prerequisites

You've completed the [MongoDB Hands-On Workshop Part 1 - Atlas](#).

Hands-on Lab

Lab 7 - Create a Microservice

Next we'll create a microservice that we'll expose to our application teams as a REST API. We'll accomplish this via a [MongoDB Stitch Function](#) and [HTTP Service](#). Our microservice will allow us to query for restaurants by name.

Create the Stitch Application

Stitch is a serverless platform, where functions written in JavaScript automatically scale to meet current demand. Return to the Atlas UI and click **Stitch Apps** on the menu on the left and then click the **Create New Application** button.

Name the application **Workshop**. The other defaults are fine:

×

Create a new application

Application Name

Workshop

Link To Cluster

Only clusters with no pending changes running MongoDB 3.4 or greater are shown

Workshop

Note:

 Stitch is currently only located in select AWS regions. Linking it to Atlas clusters in other regions may result in lower performance.

Stitch Service Name ⓘ

mongodb-atlas

Select A Deployment Region

All Stitch deployments will be globally distributed. Stitch will ensure there is a readable copy of your data in all regions and will route requests to the correct region.

• Global ⓘ

Select Region (write)

Virginia (us-east-1)

Cancel

Create

Click **Create**, which will take you to the **Welcome to Stitch!** page.

Create the Function

Now we'll create the function that queries restaurants by name. Click **Functions** on the left and then **Create New Function**. Name the function **getRestaurantsByName**:

Function Editor **Settings***

Function Name
This is the name of your function. You use this name to call your function from a client. You can come back here to change it at any time.

getRestaurantsByNam

Private
If private, this function may be called from incoming webhooks, rules, and other functions defined in the Stitch console. Private functions may not be called from Stitch client application.

☐

Can Evaluate
This is a JSON expression that must evaluate to `TRUE` before the function may run. If this field is blank, it will evaluate to `TRUE`. This expression is evaluated before other service-specific rules.

1

Cancel

Save

Click **Save**, which will open the Function Editor.

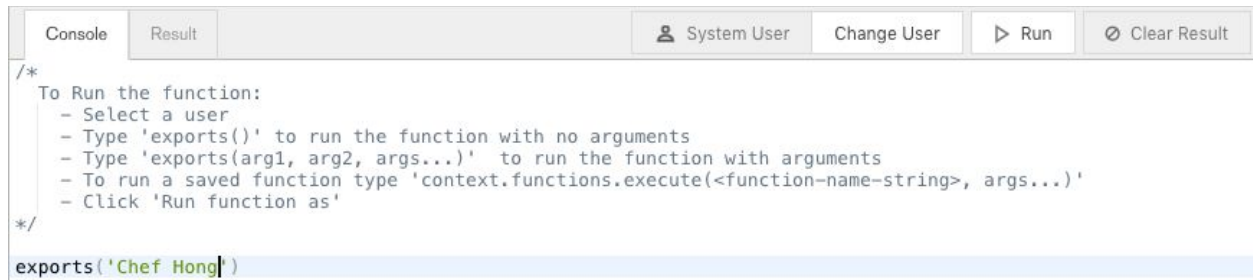
Replace the example code in the editor with the following:

```
exports = async function(arg){  
  
  var collection = context.services  
    .get("mongodb-atlas").db("Workshop").collection("restaurants");  
  
  console.log ("IN GETRESTAURANTSBYNAME FUNCTION");  
  
  //Return a single document to matching the arg/restaurant name.  
  var doc = await collection.findOne({name: arg});  
  if (typeof doc === "undefined") {  
    return `No restaurants named ${arg} were found.`;  
  }  
  
  console.log(`FOUND A MATCHING RESTAURANT: ${arg}.`);  
  
  return doc;  
}
```

You can ignore the “Missing semicolon.” warnings shown in the editor.

Let's review the code together. MongoDB has idiomatic [drivers](#) for most languages you would want to use. In this example we're using the [findOne](#) method to return a single document.

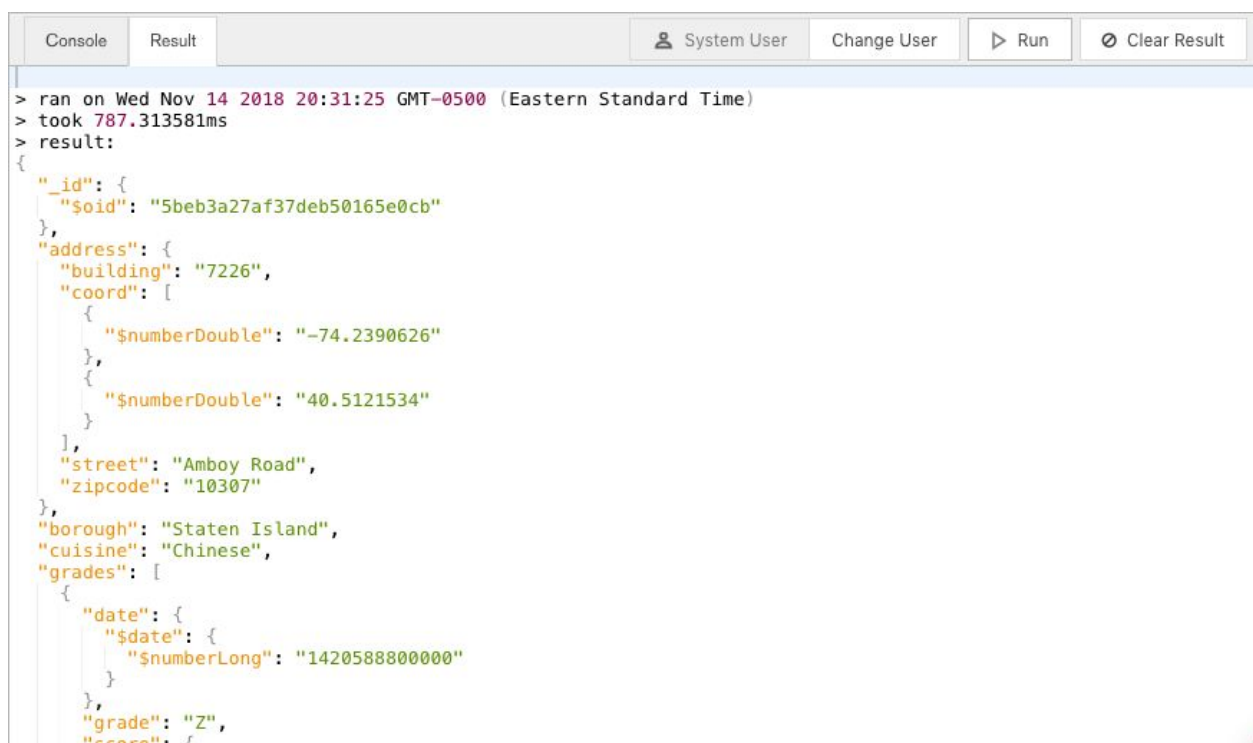
Click the **Console** tab below the editor to expand it. In the Console, change the argument from 'Hello world' to '**Chef Hong**':



```
/*
To Run the function:
- Select a user
- Type 'exports()' to run the function with no arguments
- Type 'exports(arg1, arg2, args...)' to run the function with arguments
- To run a saved function type 'context.functions.execute(<function-name-string>, args...)'
- Click 'Run function as'
*/

exports('Chef Hong')
```

Then click **Run** to test the function. You should get something similar to below with the result being the full document of the Chef Hong restaurant.



```
> ran on Wed Nov 14 2018 20:31:25 GMT-0500 (Eastern Standard Time)
> took 787.313581ms
> result:
{
  "_id": {
    "$oid": "5beb3a27af37deb50165e0cb"
  },
  "address": {
    "building": "7226",
    "coord": [
      {
        "$numberDouble": "-74.2390626"
      },
      {
        "$numberDouble": "40.5121534"
      }
    ],
    "street": "Amboy Road",
    "zipcode": "10307"
  },
  "borough": "Staten Island",
  "cuisine": "Chinese",
  "grades": [
    {
      "date": {
        "$date": {
          "$numberLong": "1420588800000"
        }
      },
      "grade": "Z",
      "score": 1
    }
  ]
}
```


Click **Save** to save the function.


Expose the Function as a REST service


Click the **Services** menu on the left and then **Add a Service**. You'll notice Stitch supports service integrations with [Twilio](#), [AWS](#) and [GitHub](#), making it very easy for you to leverage these providers' unique capabilities. More generically, Stitch also provides an [HTTP Service](#), which we will use to expose our function as a REST API.


Select the HTTP service and name it **restaurants**:

Add a Service

**Twilio**
Send and receive text messages

**HTTP**
Send GET and POST requests over HTTP

**AWS**
Access AWS services

**GitHub**
Respond to GitHub events

Service Name
Enter a unique name for this service. You can add multiple instances of any type of service.

restaurants

Cancel

Add Service

Click **Add Service**. You'll then be directed to add an incoming webhook. Click **Add Incoming Webhook** and configure the settings as shown below (the Webhook Name is **getRestaurantsByName** and be sure to enable **Respond with Result**, set the HTTP Method to **GET** and Request Validation **Do Not Validate**):

Function Editor

Settings*

Webhook Name

getRestaurantsByName

Respond With Result

☒

Run Webhook As

☒ System ⓘ

☐ User Id ⓘ

☐ Script ⓘ

HTTP Method

GET

POST

PUT

DELETE

PATCH

Request Validation

☐ Verify Payload Signature

☐ Require Secret As Query Param

☒ Do Not Validate

Cancel


Save

To keep things simple for this introduction, we're running the webhook as the System user and we're skipping validation. Click **Save**, which will take us to the function editor for the service.

In the service function we will capture the query argument and forward that along to our newly created function. Note, I could have skipped creating the function and just coded the service functionality here, but the function allows for better reuse, such as calling it [directly from a client application](#) via the SDK. Replace the code with the following:

```
exports = function(payload) {  
  
    var queryArg = payload.query.arg || '';  
    return context.functions.execute("getRestaurantsByName", queryArg);  
  
};
```


Then set the arg in the Console to **'Chef Hong'**:

 Expand Editor


```
1 exports = function(payload) {
2
3   var queryArg = payload.query.arg || '';
4   return context.functions.execute("getRestaurantsByName", queryArg);
5
6 };
```


Console


Result

 System User

Change User

 Run

 Clear Result




```
/*
To Run the function:
- Select a user
- Type 'exports()' to run the function with no arguments
- Type 'exports(arg1, arg2, args...)' to run the function with arguments
- To run a saved function type 'context.functions.execute(<function-name-string>, args...)'
- Click 'Run function as'
*/

exports({query: {arg: 'Chef Hong'}, body: BSON.Binary.fromText('{"msg": "world"}')})
```


and click **Run** to verify the result:


Console

Result

 System User

Change User

 Run

 Clear Result

```
{
  "zipcode": "10307",
},
"borough": "Staten Island",
"cuisine": "Chinese",
"grades": [
  {
    "date": {
      "$date": {
        "$numberLong": "1420588800000"
      }
    },
    "grade": "Z",
    "score": {
      "$numberInt": "18"
    }
  }
],
"name": "Chef Hong",
"restaurant_id": "50015617"
}
```

Click **Save** to the service.

Use the API

The beauty of a REST API is that it can be called from just about anywhere. For the purposes of this workshop, we're simply going to execute it in our browser. However, if you have tools like [Postman](#) installed, feel free to try that as well.

Switch back to the **Settings** tab of the getRestaurantsByName service and you'll notice a Webhook URL has been generated.

getRestaurantsByName

Save...

Function Editor

Settings

Webhook URL ⓘ

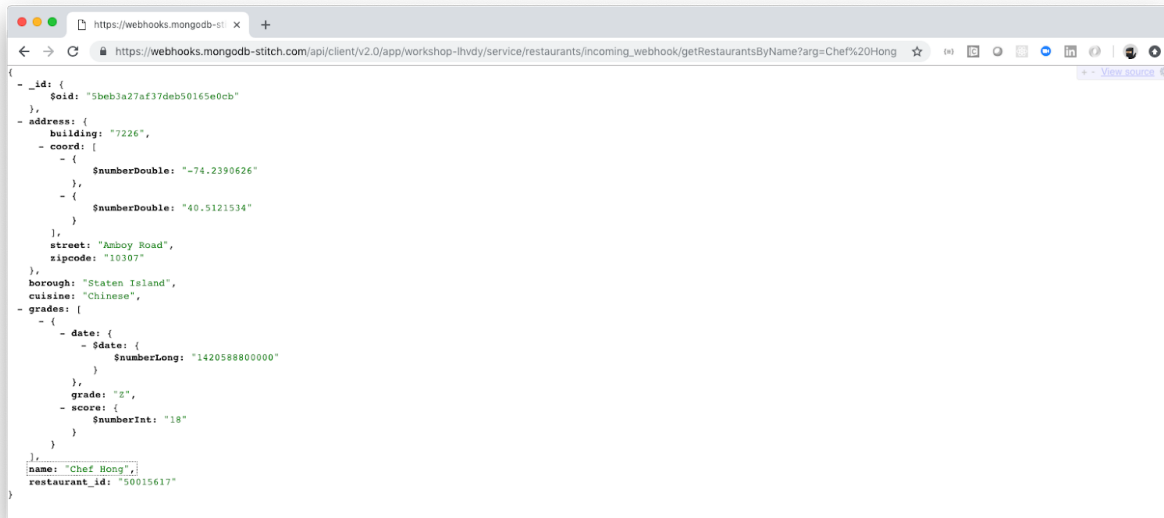
https://webhooks.mongodb-stitch.com

COPY

Click the **COPY** button and paste the URL into your browser. There's actually a restaurant in the dataset with no name, so you'll get a result. However, append the following to the end of your URL:

?arg=Chef%20Hong

and submit again (your output will look different if you don't have a [JSON viewer](#) installed):



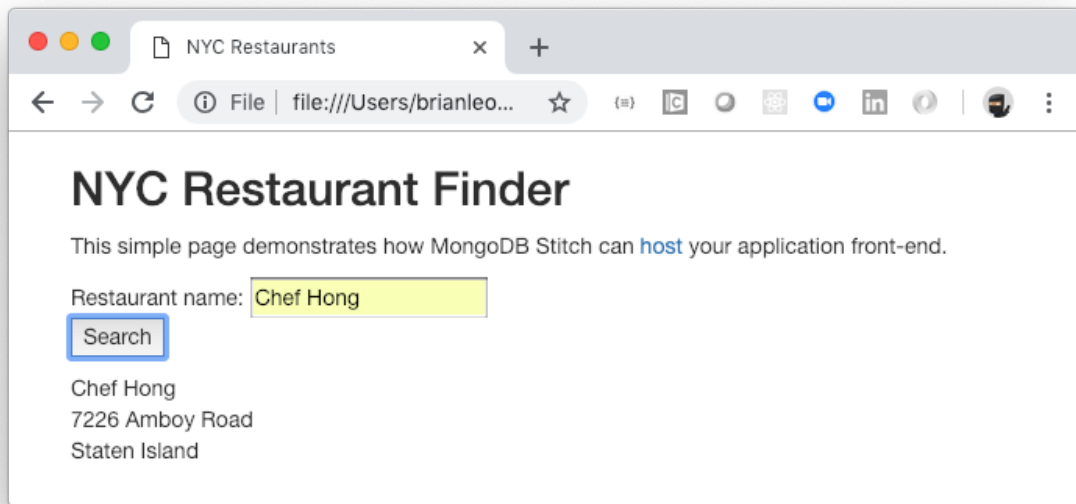
```
{
  "_id": {
    $oid: "5beb3a27af37deb50165e0cb"
  },
  "address": {
    "building": "7226",
    "coord": [
      {
        $numberDouble: "-74.2390626"
      },
      {
        $numberDouble: "40.5121534"
      }
    ],
    "street": "Amboy Road",
    "zipcode": "10307"
  },
  "borough": "Staten Island",
  "cuisine": "Chinese",
  "grades": [
    {
      "date": {
        $date: {
          $numberLong: "1420588800000"
        }
      },
      "grade": "2",
      "score": {
        $numberInt: "18"
      }
    }
  ],
  "name": "Chef Wong",
  "restaurant_id": "50015617"
}
```

Lab 8 - Host your Application

Yes, Stitch can also [host](#) your application, therefore supporting the entire application stack. Let's see this in action using a very simple front-end that will use the REST API we just created and allow us to search for restaurants in NYC.

Download and Test the UI

Download this [index.html](#) file and open it in your browser. It should work as is because it's currently pointing to a pre-existing REST API:



Open the index.html file in a code editor and familiarize yourself with the contents. Then replace the value of the webhook_ur1 variable around line 38 with the Webhook URL from the Stitch Service you created earlier. Save and test the UI.

Host the UI on Stitch

In the Stitch UI, click the **Hosting** in the left navigation bar and then click **Enable Hosting**:

Hosting (Beta)

Files Settings

workshop-lhvdy.mongodbstitch.com /


UPLOAD FILES

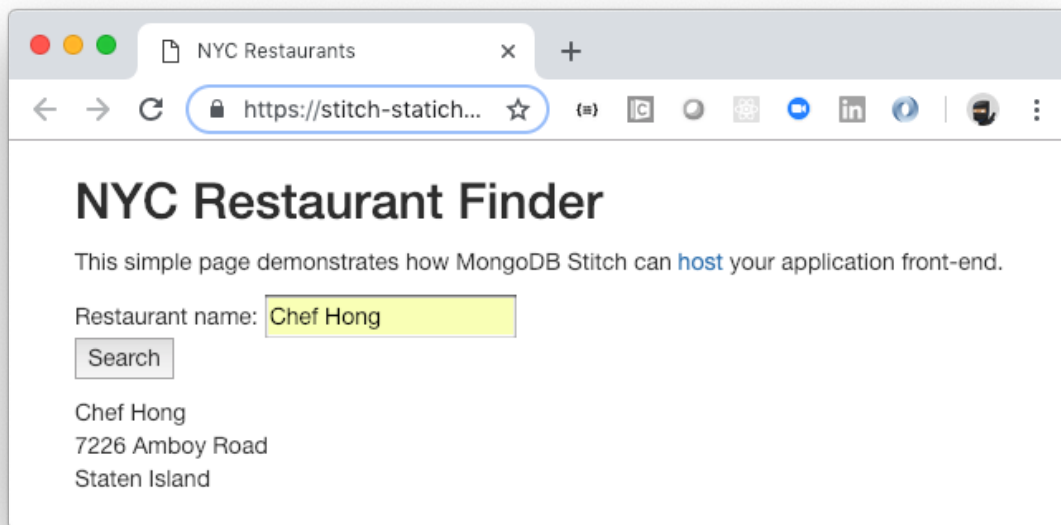
+ CREATE FOLDER

ACTIONS

<input type="checkbox"/>	Name	Last Modified	Size	File Type	Actions
<input type="checkbox"/>	index.html	01/10/2019 14:02:31	16.96 kB	text/html	...

Upload your index.html file using the **UPLOAD FILES** button. When prompted if you want to overwrite the existing index.html file, click **Upload**. Then select the action to open your file in a browser:

<input type="checkbox"/> Name ^	Last Modified ^	Size ^	File Type ^	Actions
<input type="checkbox"/>  index.html	01/10/2019 14:05:28	1.24 kB	text/html	<div><div>...</div><div><div>Open in Browser</div><div>Copy Link</div><div>Edit Attributes...</div><div>Rename...</div><div>Move...</div><div>Copy...</div><div>Delete...</div></div></div>



Notice the url in your browser. Your restaurant application is now live on the Internet! Test it and confirm that the app is still successfully using your restaurant microservice.

And that's a wrap!

Lap Optional 1 - View the Stitch Logs

Logs

Type (All) ▾	Status (All) ▾	From: dd/mm/yyyy	To: dd/mm/yyyy	Filter by UID... 🔍	Filter by Request ID... 🔍	Apply
Status	Time	Time Taken	Id	User	Name	Type
▶ OK	2018-11-15T10:19:06-05:00	98ms	5bed8e6a3e0e0b017...	--	getRestaurantsByName	Webhook
▶ OK	2018-11-15T10:19:00-05:00	24ms	5bed8e64e707c0eae...	--	getRestaurantsByName	Webhook

Lab Optional 2 - Create a Trigger

Stitch [Triggers](#) provide an easy way to enable event processing in your applications. For example, for our restaurants collection:

- We could send a text via [Twilio](#) to a restaurant owner when they receive a new review.
- We could email registered users, using AWS Simple Email Service ([SES](#)), when a new restaurant in their neighborhood opens up.
- For any restaurants added to our Restaurants collection, we could add a document to a *different* collection that health inspectors will use for their reviews.

Let's implement that last possibility.

In the Stitch UI, click the **Triggers** menu on the left and **Add a Database Trigger**. Supply the information as shown below:

▼ TRIGGER DETAILS

Trigger Name

This is the name your trigger will use in the Stitch Admin UI and application configuration.

Enabled



Event Ordering

By default, database trigger events are processed sequentially. Turning Event Ordering off will allow requests to be processed in parallel and more quickly if many matching events are created at the same time. Learn more about [ordered triggers](#).



▼ TRIGGER SOURCE DETAILS

Select Linked Cluster

 | ▼

Database Name

Collection Name

Operation Type

Stitch will only trigger on these operations.

☒ Insert ☐ Update ☐ Delete ☐ Replace

Full Document

Get the full document in your [change event](#).



Then link a New Function named **triggerHealthInspection** and replace the example code with the following:

```
exports = function(changeEvent) {  
  
    var fullDocument = changeEvent.fullDocument;  
    var collection = context.services.get("mongodb-atlas")  
        .db("Workshop").collection("NewRestaurants");
```

```
var status = collection.insertOne(fullDocument);
console.log(status);

};
```

▼ LINKED FUNCTION

Function

Select the function to be executed on a [change event](#). Selecting a new function will create a default function you can edit in the future.

+ New Function ▼

Function Name

Enter the name for the new function.

triggerHealthInspection

Function Code


The following code will be executed for each event. The default function provided has no side effects. You can edit this code later.

Expand Editor

```
1 exports = function(changeEvent) {
2
3   var fullDocument = changeEvent.fullDocument;
4   var collection = context.services.get("mongodb-atlas")
5     .db("Workshop").collection("NewRestaurants");
6   var status = collection.insertOne(fullDocument);
7   console.log(status);
8
9 };
10
```

And **Save** the trigger.

To test the trigger, let's use the data explorer in Atlas. Navigate back to your cluster and select the **Collections** tab. Hover over a document in the Restaurants collection and click the Clone

Document icon () that appears in the upper right.

>

```
_id: ObjectId("5beb3a1aaf37deb501658311")
> address: Object
  borough: "Bronx"
  cuisine: "Bakery"
> grades: Array
  name: "Morris Park Bake Shop"
  restaurant_id: "30075445"
```

Optionally change some of the fields:

Insert Document

```
1  _id : ObjectId("5bedb3771c9d4400002a425a ")
2  > address : Object
3    borough : "Staten Island "
4    cuisine : "American "
5  > grades : Array
6    name : "Cheeseburger Town "
7    restaurant_id : "12345678 "
```

ObjectId

Object

String

String

Array

String

String

Cancel

Insert

And click **Insert**.

Refresh the Collections page to see your NewRestaurants collection:

The screenshot displays the MongoDB Atlas interface. On the left sidebar, under '1 DATABASES 2 COLLECTIONS', there is a '+ Create Database' button and a 'NAMESPACES' section. The 'Workshop' namespace is expanded, showing the 'NewRestaurants' collection. The main panel is titled 'Workshop.NewRestaurants' and shows a table with columns: 'COLLECTION SIZE' (484B), 'TOTAL DOCUMENTS' (1), and 'INDEXES TOTAL SIZE' (16KB). A '+ INSERT' button is visible. Below the table, there are 'Find' and 'Indexes' tabs. The 'Find' tab is active, showing a filter input with the text '{"filter": "example"}' and 'Find' and 'Reset' buttons. Below the filter, it says 'QUERY RESULTS 1-1 OF 1'. The query result is a JSON document:

```
{  "_id": ObjectId("5bedb3771c9d4400002a425a"),  "address": Object,  "borough": "Staten Island",  "cuisine": "American",  "grades": Array,  "name": "Cheeseburger Town",  "restaurant_id": "12345678"}
```

We hope you have enjoyed this tour of MongoDB, Atlas, and Stitch! As you see with Atlas and MongoDB Stitch, it is ridiculously easy to work with data! We can't wait to see what you build!

Please share any feedback on whether this was helpful and how to make it better.

If you're interested in exploring more, check out the [Stitch Tutorials](#).