Introduction to

Hands-On Workshop

Lab 3 - Stitch

Overview

In the first two labs of this workshop you created a MongoDB cluster and loaded your data. Now it's time to put that data to action. In this lab, you'll create microservices to expose the data via REST APIs, and then create a basic front-end application that leverages those APIs.

Specifically, you'll create the APIs and an associated UI to query and add new restaurants. All of this will be hosted on MongoDB Stitch!

Prerequisites

You've completed Labs 1 and 2 of this workshop, so you have a MongoDB cluster deployed in Atlas and you've loaded the restaurant data set.

Hands-On Exercises

Exercise 1 - Create a Microservice

Let's create a microservice that we'll expose to our application teams as a REST API. We'll accomplish this using a MongoDB Stitch Function and an HTTP Service. Our microservice will allow users 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** under Services on the menu on the left:

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 available clusters in 'BL_ATLAS-WORKSHOP' running MongoDB 3.4 or greater are shown. Refresh this page to view available clusters.

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 Model

Choose from two deployment models - 'Local' (Single Region) or 'Global' (distributed across all supported regions). Learn more about deployment models.

- ○ Local
- ◉ Global

Select a Primary Region

Stitch will process application requests in the region closest to your end users. We recommend choosing the region closest to your cluster's primary. Learn More.

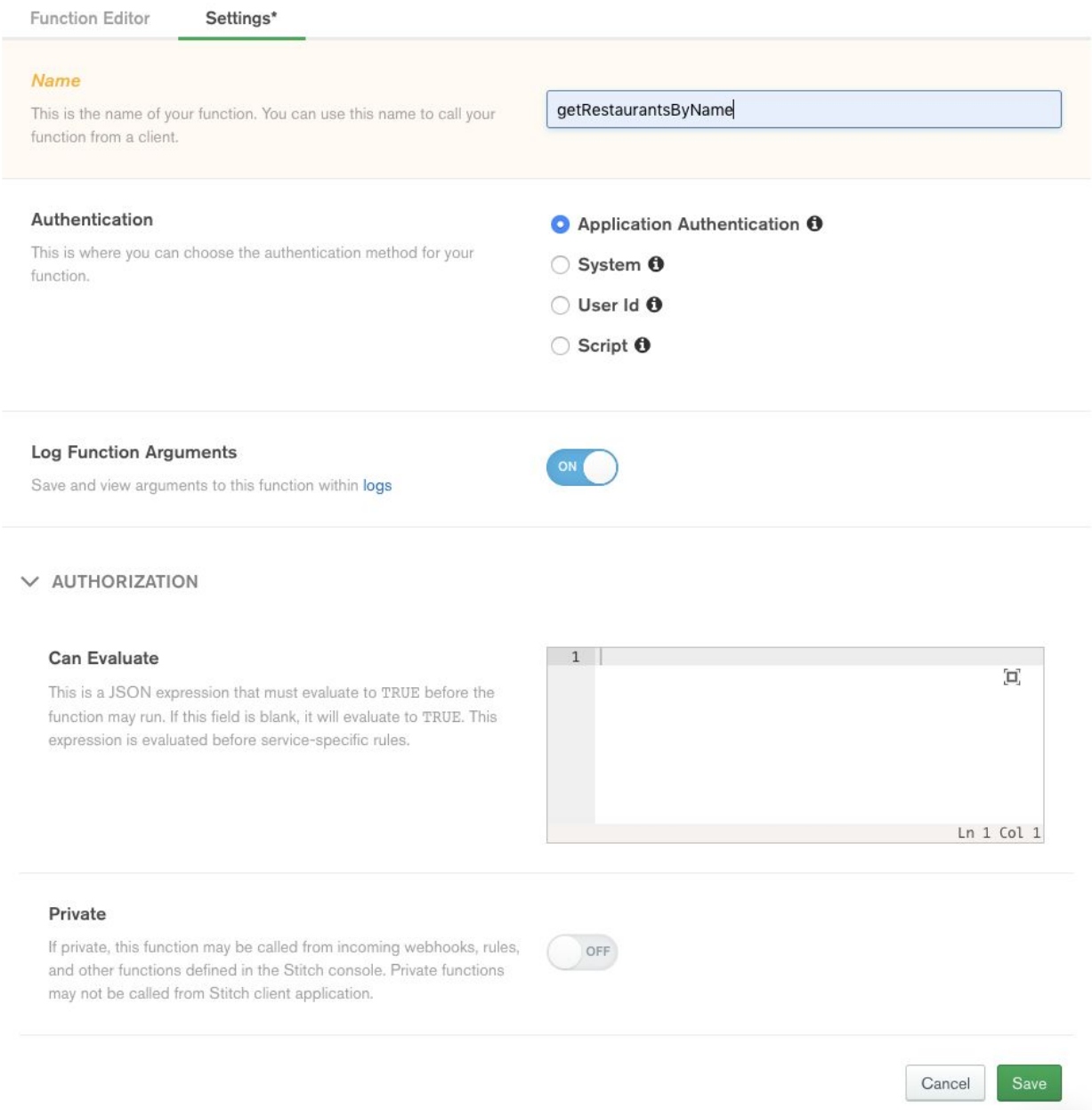
Virginia (us-east-1)

Cancel | Create

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

Create the Function

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



> *Note, leave the **Can Evaluate** field blank. You'll paste the function code below into a new editor that appears after you click save.*

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

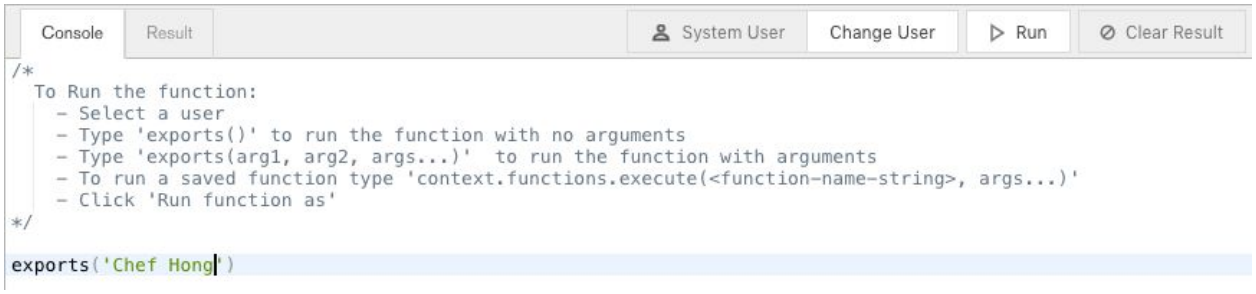
Replace the example code in the editor with the following:

```
exports = async function(arg){

    const db = "Workshop";
    const coll = "Restaurants";

    var collection = context.services
        .get("mongodb-atlas").db(db).collection(coll);
    var doc = await collection.findOne({name: arg});
    if (doc === null) {
        msg = `No restaurants named ${arg} were found. Check the spelling
of your database '${db}' and collection '${coll}'.`
        return msg;
    }
    return doc;
}
```

Note, you can ignore the "Missing semicolon." warnings shown in the editor.

Review the code as it's your first introduction to working with MongoDB from application code, in this case JavaScript. 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.

In the Console below the editor, change the argument from 'Hello world!' to **'Chef Hong'**:



Then click **Run** to test the function:

Console | Result | System User | Change User | Run | Clear Result

```
> 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": {
```

Click **Save** to save the function and then **REVIEW & DEPLOY CHANGES** at the top of the page:

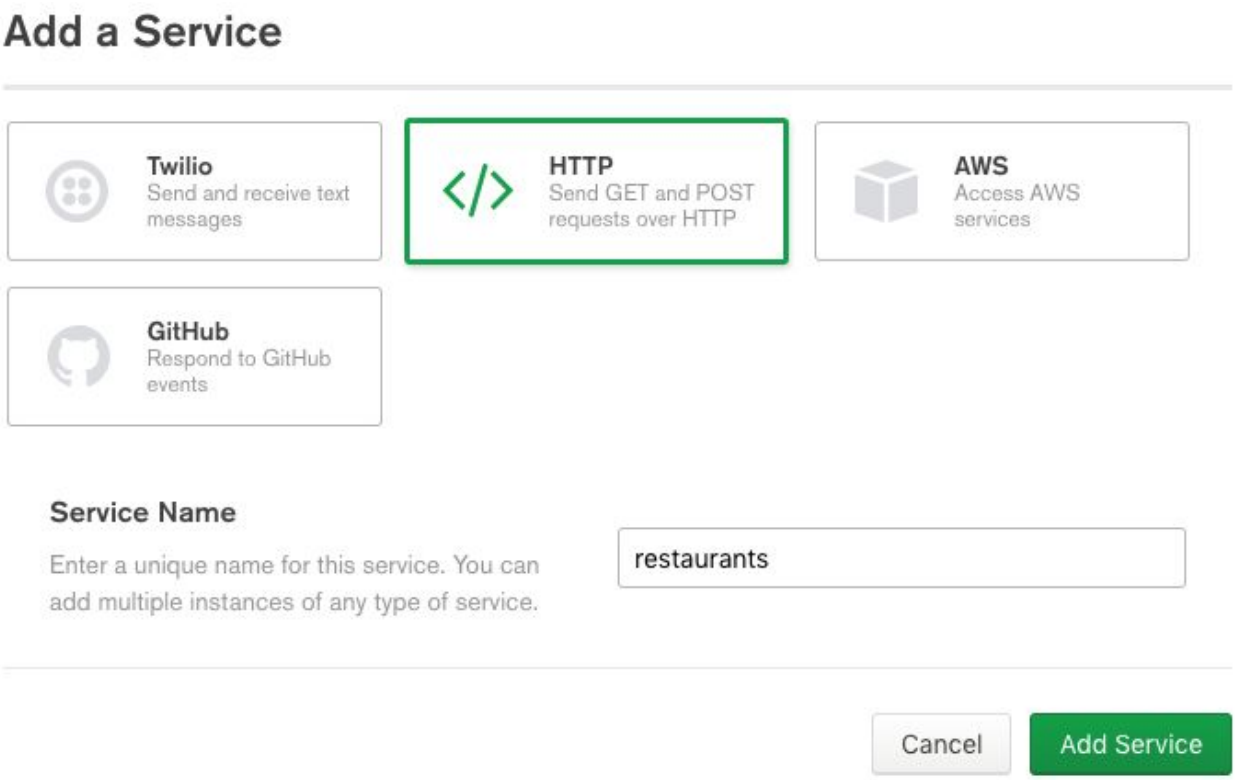
Changes have been made to your Stitch app since the last deploy | REVIEW & DISCARD | REVIEW & DEPLOY CHANGES

Functions > getRestaurantsByName

Expose the Function as a REST service

After the deployment banner turns green to indicate success, click the **3rd Party Services** menu on the left and then **Add a Service**. You'll notice Stitch supports service integrations with Twilio 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**:



Click **Add Service**. You'll then be directed to **Add** an **Incoming Webhook**. Do so 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 to **No Additional Authorization**):

Function Editor | **Settings***

Name

This is the name of your webhook.

getRestaurantsByName

Authentication

This is where you can choose the authentication method for your webhook.

- ○ Application Authentication
- ◉ System
- ○ User Id
- ○ Script

Log Function Arguments

Save and view arguments to this function within logs

OFF

WEBHOOK SETTINGS

Webhook URL

This is the callback URL for an incoming webhook from this third-party service to execute a Stitch function.

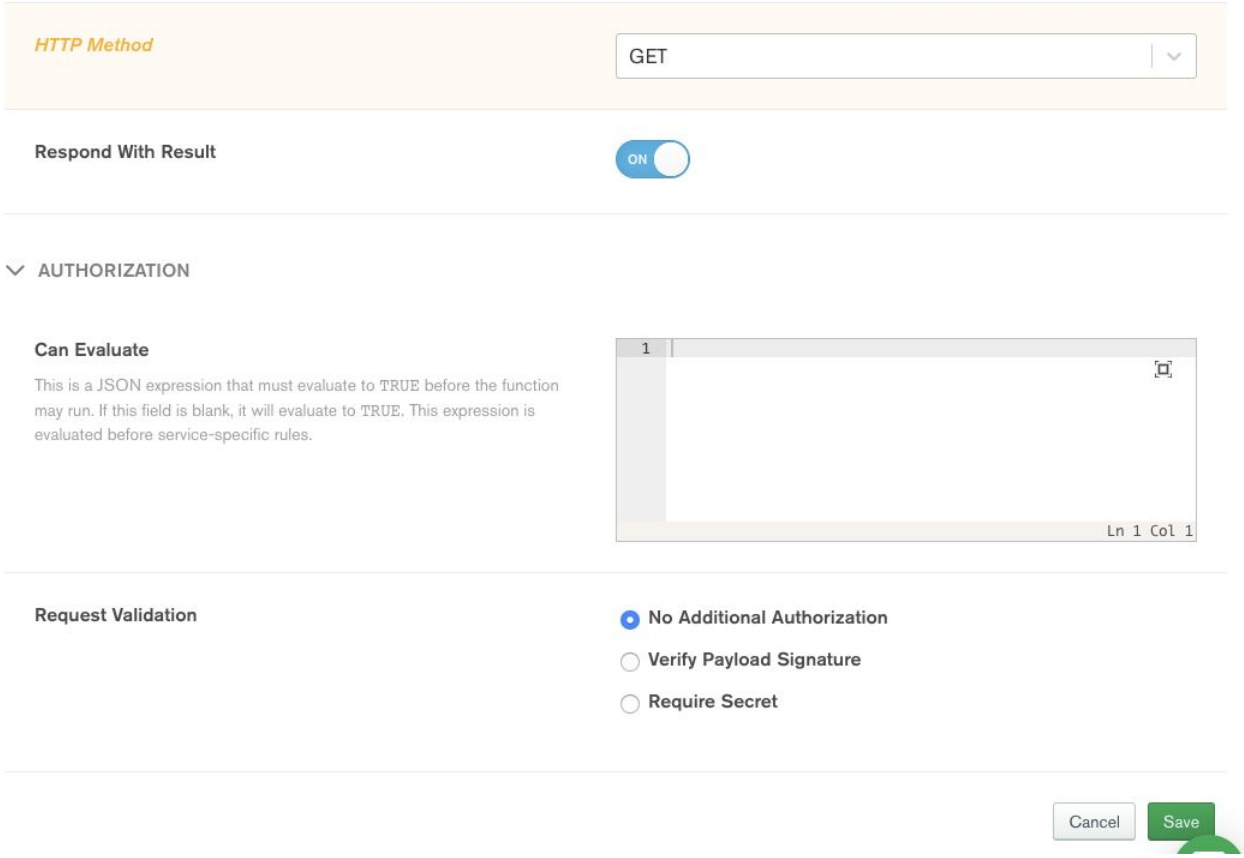
```
https://webhooks.mongodb-stitch.com/api/client/v2.0/app/workshop-vvuln/
```

COPY

You can make a test request to this webhook using this `curl` command.

```
curl \
-H "Content-Type: application/json" \
-d '{"foo":"bar"}' \
https://webhooks.mongodb-stitch.com/api/client/v2.0/app/workshop-vvuln/service/restaurants/incoming_webhook/webhook0
```

COPY



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, we 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.arg1 || '';
    return context.functions.execute("getRestaurantsByName", queryArg);

};
```

Then set the arg1 in the Console to **'Chef Hong'**. You can ignore arg2 and fromText because we're not using them.

Function Editor* Settings

```
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: {arg1: 'Chef Hong', arg2: "world!"}, body: BSON.Binary.fromText('{"msg": "world"}')})
```

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 save the service and then **REVIEW & DEPLOY CHANGES** at the top of the page.

Use the API

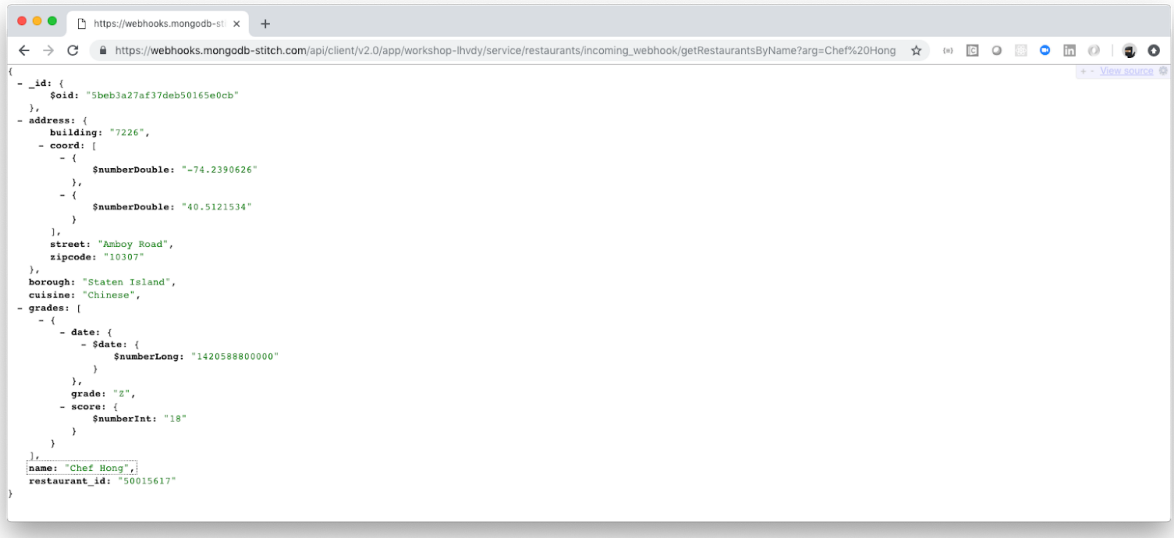
The beauty of a REST API is that it can be called from just about anywhere. For the purposes of this workshop, we're simply going to execute it in our browser. However, if you have tools like Postman installed, feel free to try that as well.

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

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:

?arg1=Chef%20Hong

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

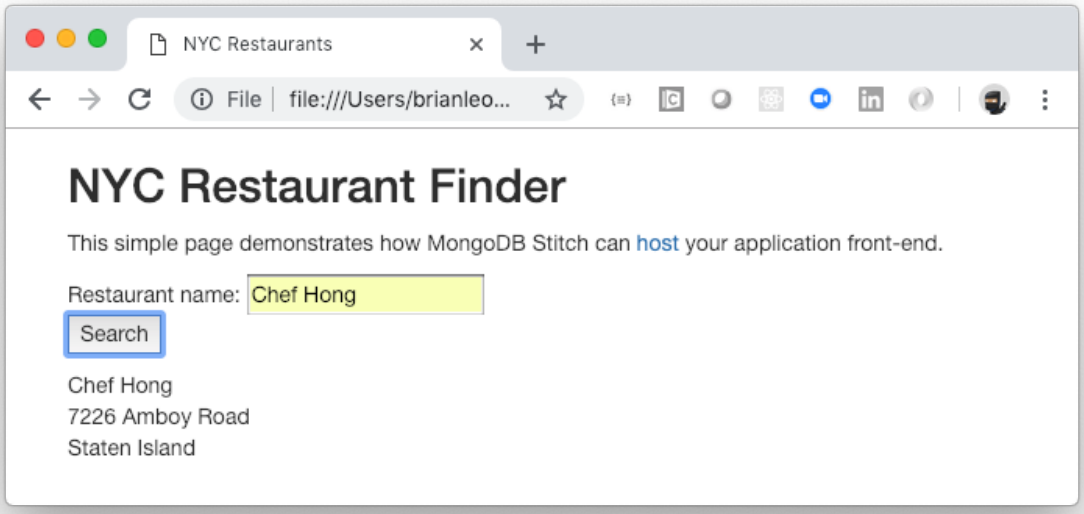


Exercise 2 - Host your Application

Now that we have a web service in place, let's create a web application (a simple form) that utilizes it. Stitch can host such an 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

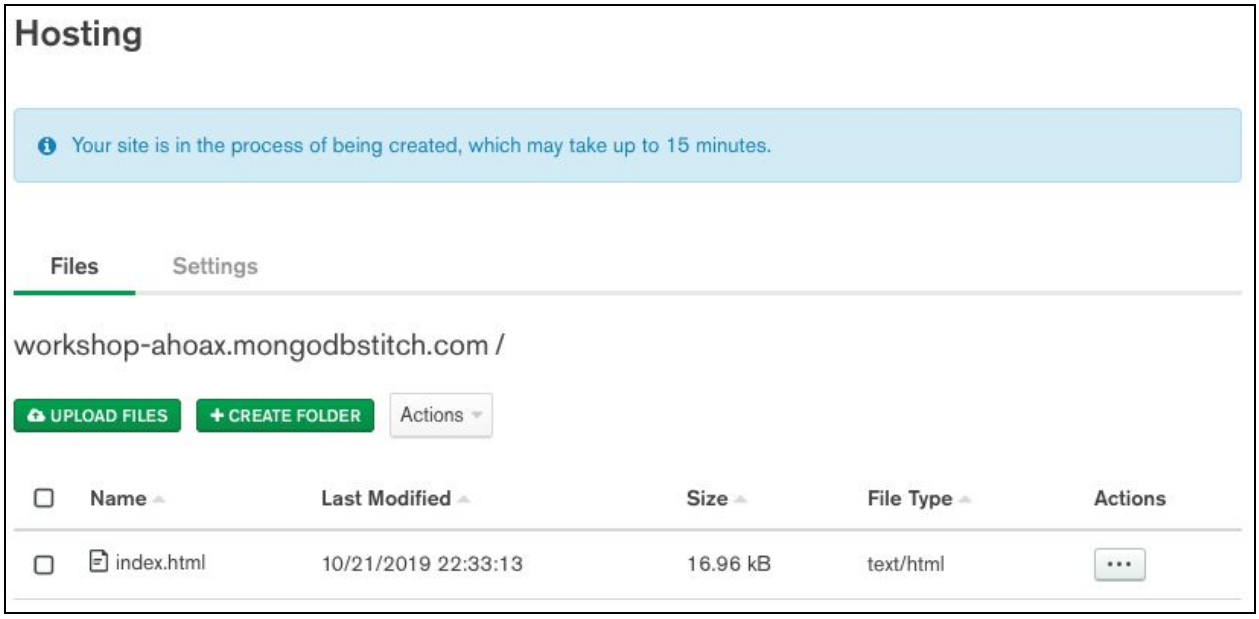
Download this index.html file and open it in your browser. It should work as is because it's currently pointing to a pre-existing REST API:



Open the index.html file in an editor and familiarize yourself with the contents. Then replace the value of the webhook_`url` variable with the Webhook URL from the Stitch Service you created earlier. Save and test the UI.

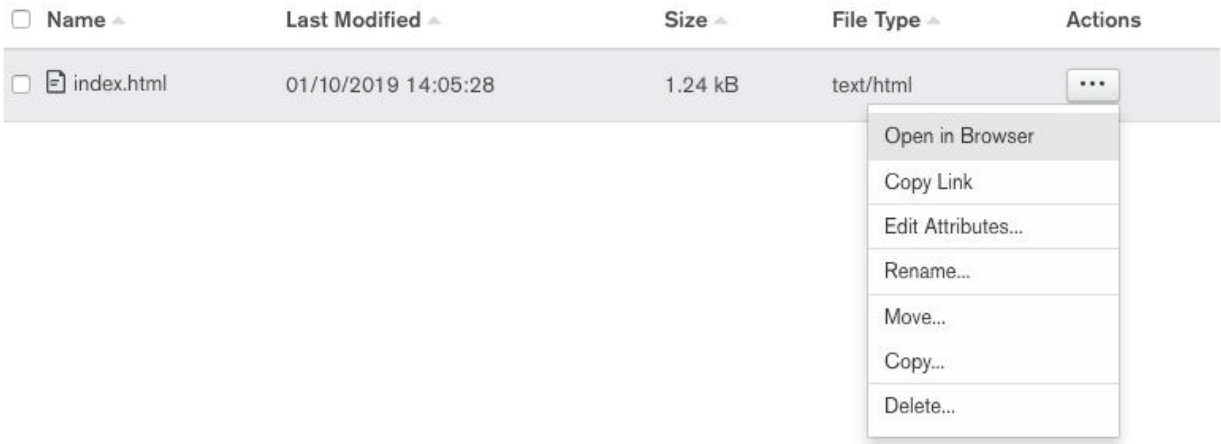
Host the UI on Stitch

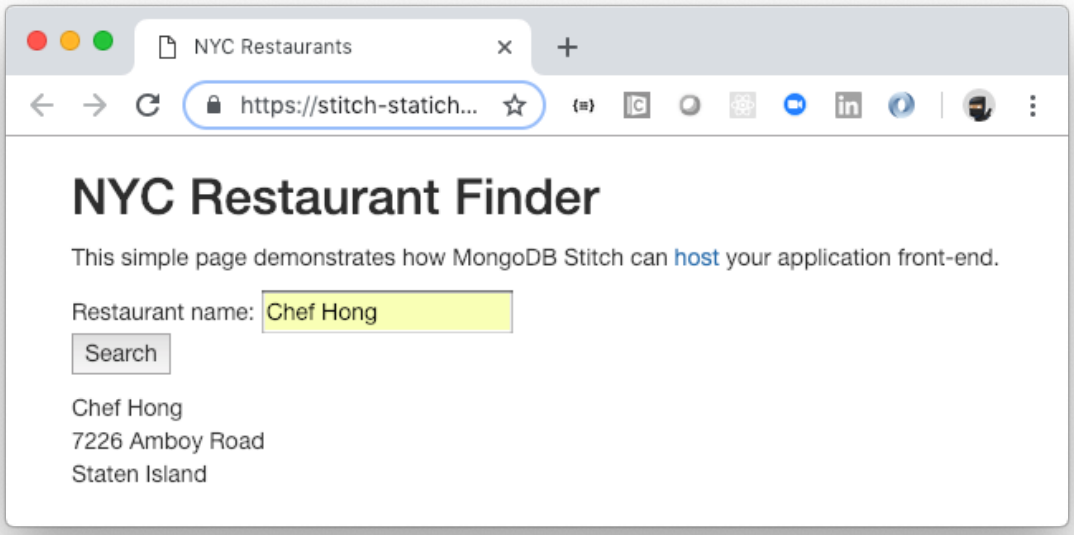
Click the **Hosting** menu on the left and then click **Enable Hosting**:



Upload your index.html file (which will overwrite the sample file) and click **REVIEW & DEPLOY CHANGES** and then wait while your site is being created:

When ready, select the action to open your file in a browser:





Exercise 3 - View the Stitch Logs

The Stitch Logs record all of the activity happening in your application and are a great resource for debugging. Click **Logs** from the left menu:

Exercise 4 - New Restaurant UI

Let's incorporate all of these new features into a user interface for adding new restaurants.

Create AddNewRestaurant HTTP Service

Let's create another HTTP service with a webhook that we can call from our UI.

Click **3rd Party Services** on the left and select the **restaurants** service we created earlier. Then add a new incoming webhook named **addNewRestaurant** configured with the settings as shown below (set the **Name** and **HTTP Method**, leaving the others unchanged):

Function Editor | **Settings***

Name

This is the name of your webhook.

addNewRestaurant

Authentication

This is where you can choose the authentication method for your webhook.

- ○ Application Authentication
- ◉ System
- ○ User Id
- ○ Script

Log Function Arguments

Save and view arguments to this function within logs

OFF

WEBHOOK SETTINGS

Webhook URL

This is the callback URL for an incoming webhook from this third-party service to execute a Stitch function.

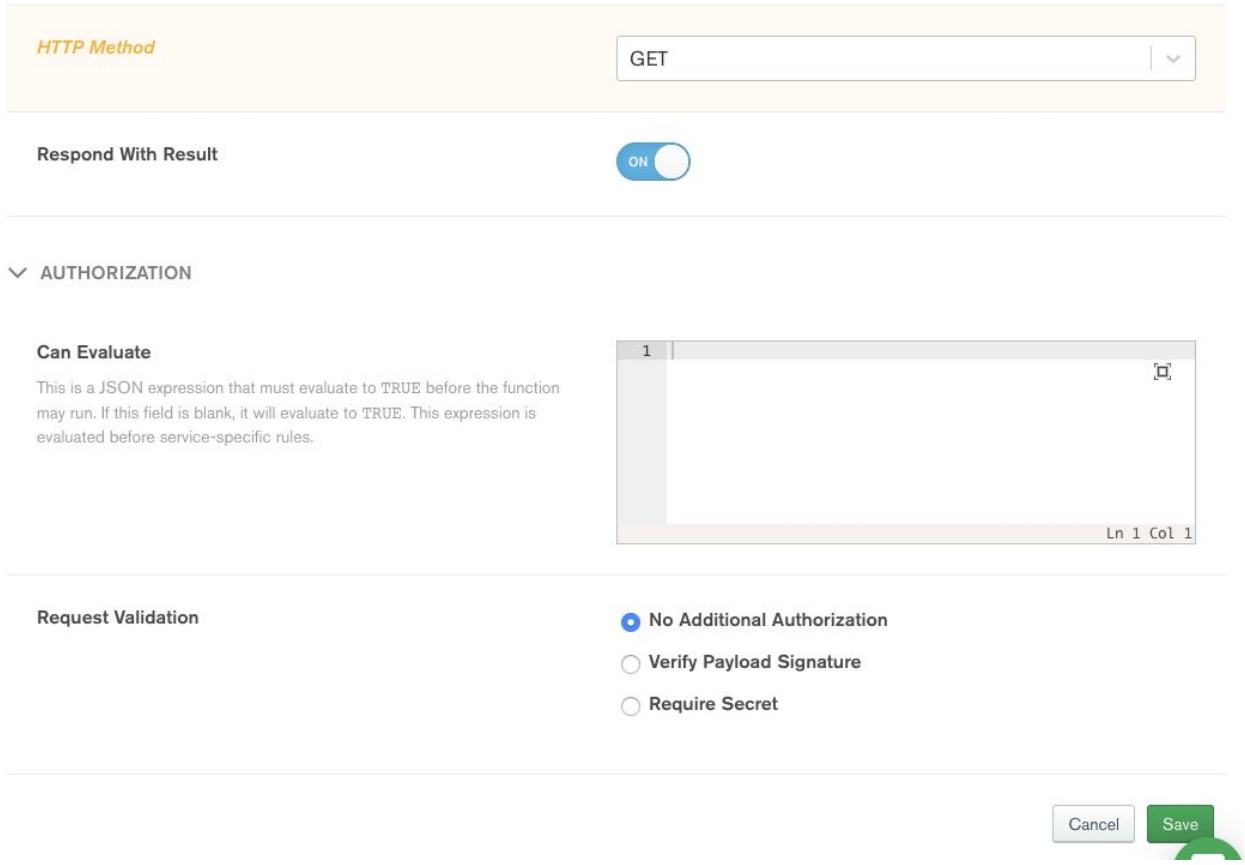
```
https://webhooks.mongodb-stitch.com/api/client/v2.0/app/workshop-vvuln/
```

COPY

You can make a test request to this webhook using this `curl` command.

```
curl \
-H "Content-Type: application/json" \
-d '{"foo":"bar"}' \
https://webhooks.mongodb-stitch.com/api/client/v2.0/app/workshop-vvuln/service/restaurants/incoming_webhook/webhook0
```

COPY



Click **Save**.

When this webhook is called, we will insert a new document for the restaurant into the collection. Replace the code in the Function Editor with the following:

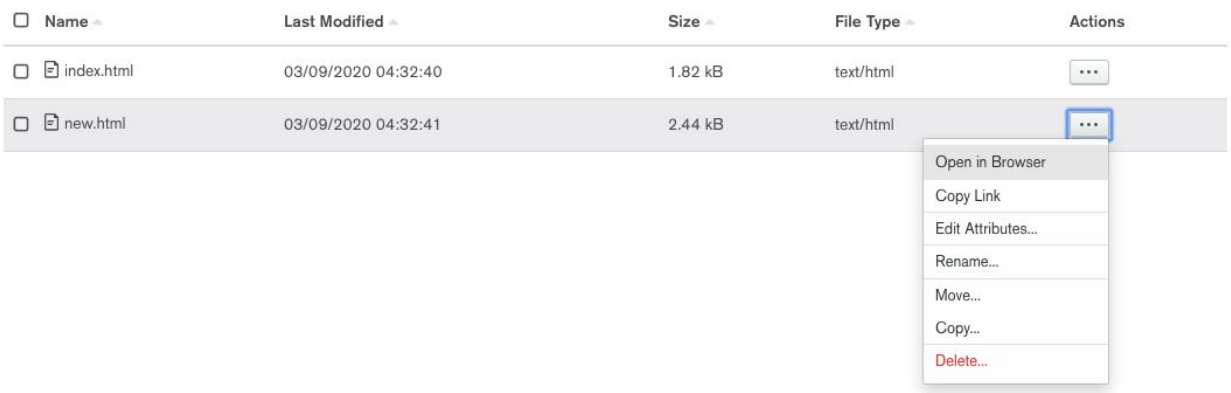
```
exports = async function(payload) {

  var restaurant = payload.query.arg1 || '';
  var newRestaurant = {name: restaurant, status: "No inspection info."};
  newRestaurant.populatedOn = Date();

  // Part 2: Uncomment the following line when adding a 2nd field to the
data entry form.
  // newRestaurant.capacity = payload.query.arg2 || '';

  var collection =
context.services.get("mongodb-atlas").db("Workshop").collection("Restaurant
s");

  var status = collection.insertOne(newRestaurant);
```

```
  console.log(status);
  return newRestaurant;
};
```

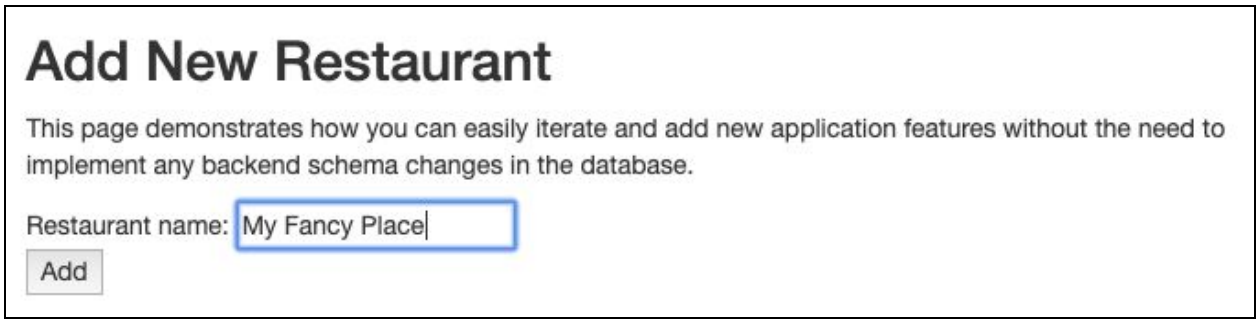
Click **Save**, and then **Review & Deploy** the changes.

Download and Host the Add New Restaurant UI

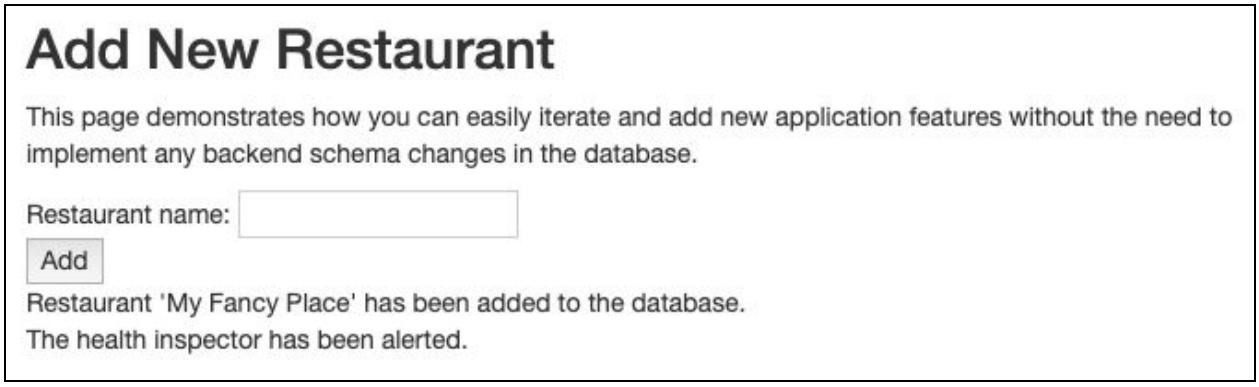
Download this [addNewRestaurant.html]() file and open it in a text editor. Replace the webhook URL placeholder with the actual URL from the AddNewRestaurant Stitch Service you created (the Webhook URL is identified in the service's Settings tab). Save and test the UI (i.e. in your local browser). Then host the UI in Stitch and test again. Recall that to deploy the new web form to Atlas you will select **Hosting** from the left menu, **upload** the addNewRestaurant.html file, and then **Review & Deploy** your changes. Then, **open** the new form in a browser using the appropriate Action:



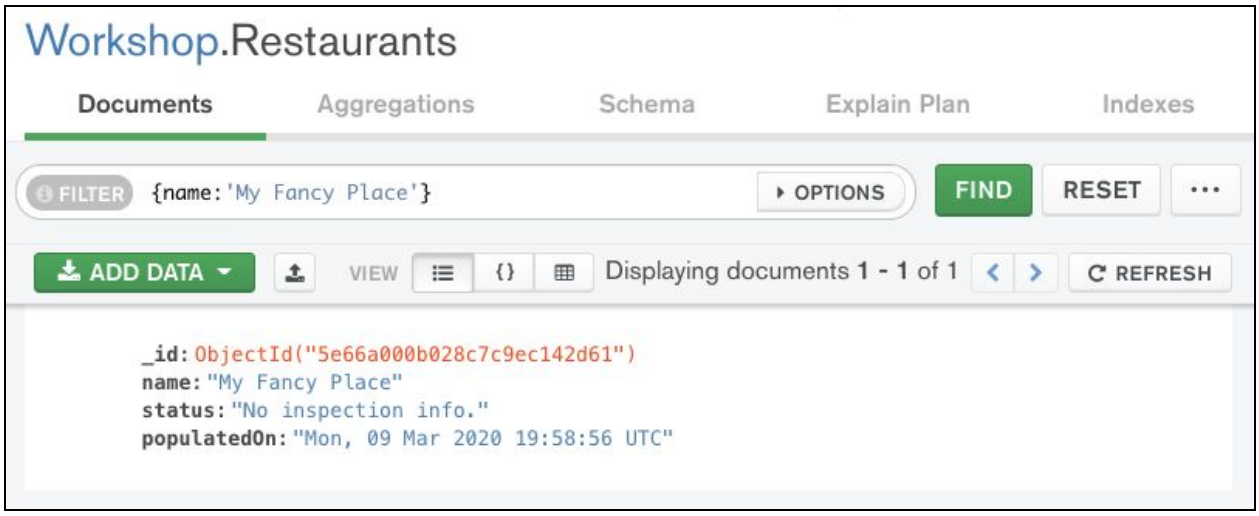
Use the form to add a new restaurant to the database:



If successful, you should see a confirmation message:



You can use either Compass or the Atlas Data Explorer to query the collection for the restaurant you just added.



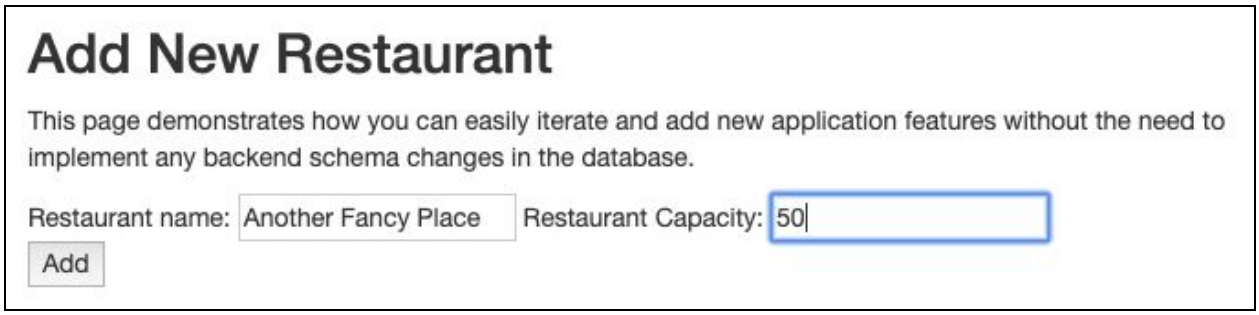
Exercise 5 - Fast Iteration: Adding a New Feature

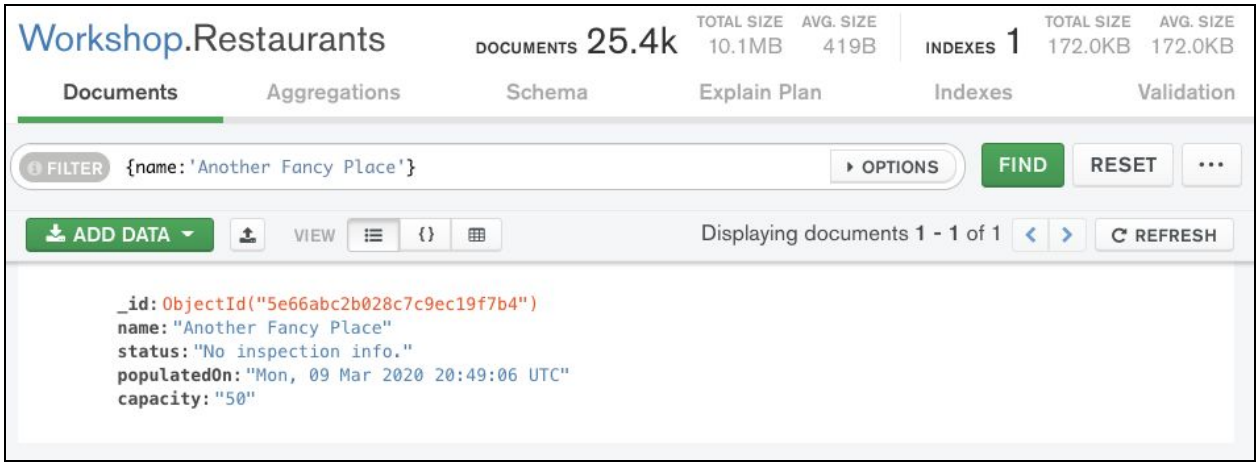
So far, so good! But, now you're being told there needs to be a way to capture restaurant capacity. With MongoDB, this is done entirely within the application code; no database-level modification is required. To implement this change, perform the following steps:

1. Using a text editor, modify the addNewRestaurant.html file to include a new data field for restaurant capacity. Do this by uncommenting the code in 3 separate places labeled as "PART 2". Save the file. You can reload your browser window using this local file copy, or you can upload the file to Stitch, replacing the previously uploaded version. Do not submit the form until we make the necessary change in the Stitch Function.

2. Within Stitch, acces the Function Editor for the addNewRestaurant Webhook. Uncomment the single line of code that converts the new capacity value to an appropriate field within the JSON document. **Save** the modified function, and then **Review & Deploy** the changes.

3. Now, in the modified HTML form, specify the name of a new restaurant as well as its capacity, and **Submit** the form. If the operation was successful, use Compass or Atlas to query the collection for the newly added restaurant, verifying that the capacity was indeed stored as expected.





If you encounter difficulties and need some help, the completed exercise is available here, in the file *addNewRestaurantV2.html*.

Congratulations!

You've completed this lab on building an application with MongoDB using JavaScript and Stitch! You've seen how Stitch can host serverless functions, as well as entire applications. And you've also seen how new requirements can be rolled out much more quickly and efficiently, without the need for any server-side schema alteration scripts.

Next, we will be looking at using Triggers to automatically react to data events in real time. **So leave a browser tab open with the "Add a New Restaurant" application page.**

If you're interested in exploring Stitch further, there's a Part 5 - Rich UI workshop, that builds on what we've completed thus far. In addition, you can also check out the Stitch Tutorials.