Building a Serverless REST API with Node.js and MongoDB

March 8th 2020

@adnanrahic

Dev/Avocado at Sematext.com.
Co-Founder at Bookvar.co.
Author of "Serverless JavaScript by Example"

The Serverless movement has gained a bit of momentum in the past few months. Everybody seems to be talking about it. Some would even call it a revolution! But, let's not get too excited. Don't be like me. I get too hyped up about cool stuff like this, and start writing articles. If this is all new to you, here's a piece I wrote a while back, explaining the core concepts.

A crash course on Serverless with Node.js
*Regardless of your developer background, it's inevitable you've heard the term Serverless in the past year. The word…*hackernoon.com

In that spirit, the time I've invested into exploring what's reasonable to build using Serverless Architecture maybe exceeds what is viewed as healthy. My conclusion is that pretty much everything is eligible to be built in a Serverless manner. The only question you need to ask yourself is whether you really need it. Lambdas are stateless, meaning the whole concept of writing server-side code needs to be learned all over again.

Sounds fun? Yeah, it is for me too. I recently published a hands-on course on using Serverless Architecture in real life. I poured all my findings and sensible reasons for using Serverless into this course. I kept asking myself the question *"Why do I need Serverless?"* throughout the creation process. You can find my thoughts below.

Serverless JavaScript by Example [Video] - Video | Now just $5
*Become dexterous with live demonstrations on serverless web development*www.packtpub.com

Why use Serverless for REST APIs?

Why not? Is it because we can, or do we see a clear advantage over traditional servers? Both sides of the coin have valid arguments. Serverless is conceived as always up. Because you don't have to manage anything, you don't worry about uptime, it'll just work. It also scales automatically. That's nice. Really nice. Scaling servers is not fun.

But what about persistent storage? We can't spin up a MongoDB database on a server like we're used to. However, if you've been following the **"separation of concerns"** lifestyle that has been on the rise in the past year, you may already be used to separating your database from your back end. Even more so if you are used to writing microservices. You just give your app a connection URL and there's the database, ready to go.

You up for a challenge?

This article will show you how to hook up a MongoDB database as a service to a Serverless REST API. Maybe a bit cheeky, since the preferred way of using AWS Serverless Architecture is with their NoSQL DBaaS called DynamoDB. But I like combining weird stuff. And, to be honest, MongoDB Atlas is amazing. It's MongoDB's own DBaaS. You can get a dedicated MongoDB cluster for free.

What's awesome with this set up is that I'll show you how to write code the way you are already used to. Everything you know from working with Node.js, Express and Mongoose will be re-used in this tutorial.

What's new, is the mindset behind using the Lambda compute service. An AWS Lambda function is basically a Docker container. Once the Lambda is invoked, the container spins up and runs the code. This is when we want to initialize the database connection, the first time the function is invoked, when the Docker container is first initialized. Every subsequent request to the Lambda function should use the existing database connection. Simple enough? Let's get crackin'!

Getting up and running

I'll assume you already have a basic understanding of the Serverless framework. I would also hope you have an AWS account set up. If you don't, please take a look at the article I linked at the top.

1. Creating a service

First of all let's create a fresh service to hold all our code.

```
$ sls create -t aws-nodejs -p rest-api && cd rest-api
```

This command will scaffold out all the necessary files and code to create our

Lambda functions and API Gateway events. It will do this in the path we gave it with the -p flag. Meaning it will create a directory named rest-api. We want to change into that directory and work from there.

2. Installing modules

There are a couple of modules we need. First of all we need the Serverless Offline plugin to be able to run our code locally before deploying to AWS. Then we need to grab mongoose, my ORM of choice, and dotenv, because I like **not** pushing keys to GitHub. Pushing keys to GitHub sucks. Don't do that. Every time you push a key to GitHub a baby penguin dies. I mean, not really, but still, it's that bad.

Make sure you're in the rest-api directory. First install Serverless Offline, then mongoose and dotenv.

```
$ npm init -y
$ npm i --save-dev serverless-offline
$ npm i --save mongoose dotenv
```

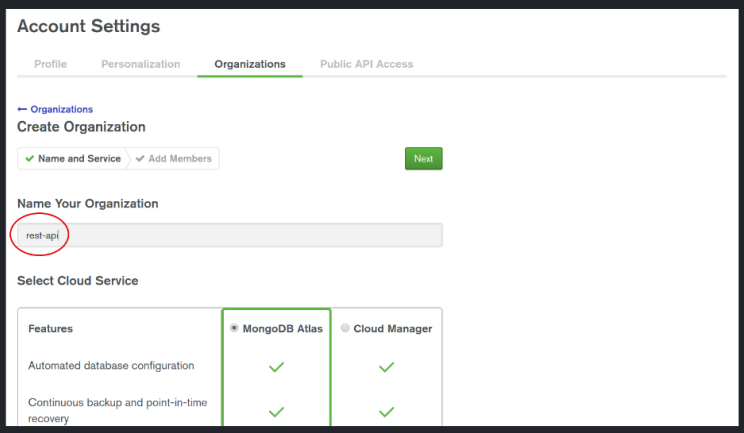
That's it, let's take a break from the terminal and jump over to Atlas to create a database.

3. Creating a database on MongoDB Atlas

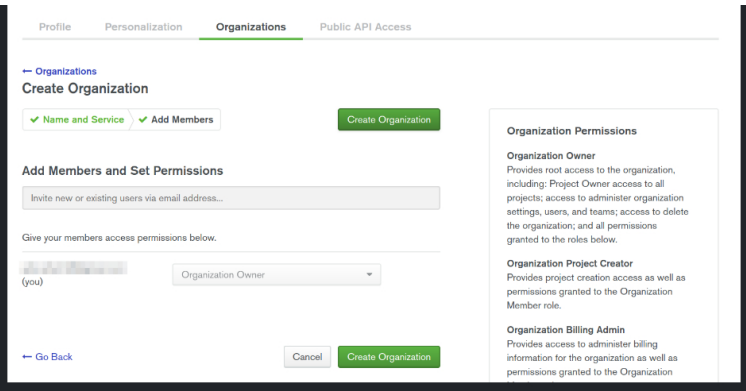
Ready for some more configuration? Yeah, nobody likes this part. But bare with me. Jump over to MongoDB Atlas and sign up.

Fully Managed MongoDB, hosted on AWS, Azure, and GCP
*MongoDB Atlas is a cloud-hosted MongoDB service engineered and run by the same team that builds the database. It…*www.mongodb.com

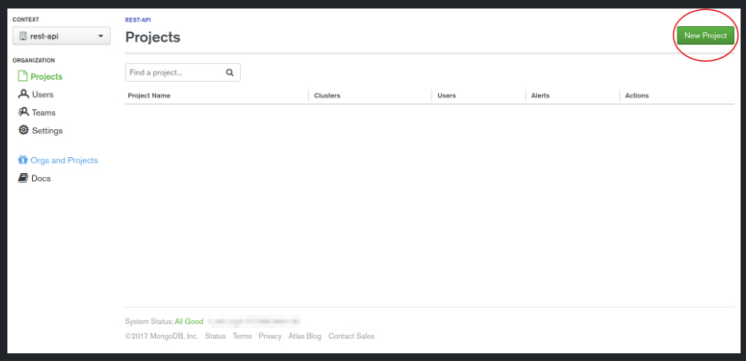
It's free and no credit card is required. It'll be the sandbox we need for playing around. Once you have your account set up, open up your account page and add a new organization.



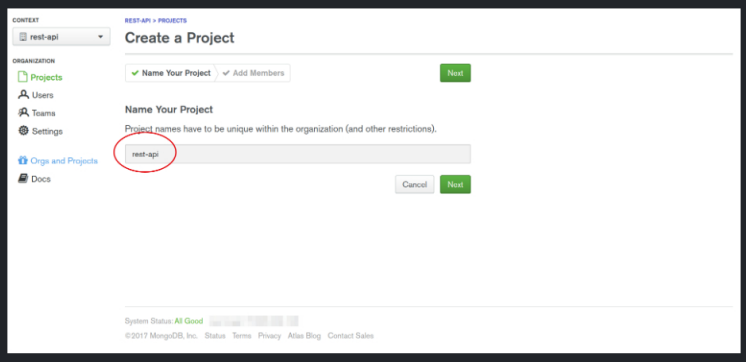
Add a name you think fits, I'll stick with `rest-api`. Press next and go ahead and create the organization.



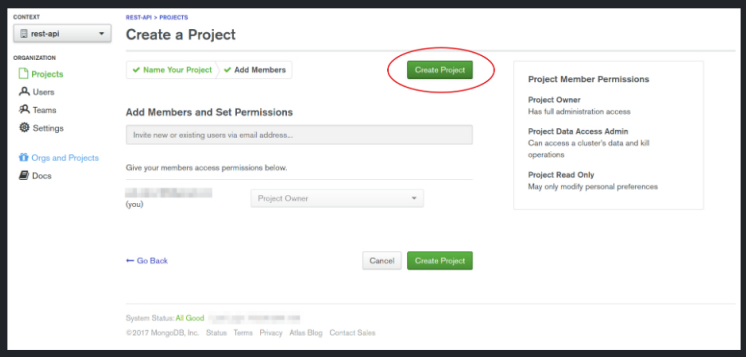
Nice. That'll take you to the organization page. Press on the new project button.



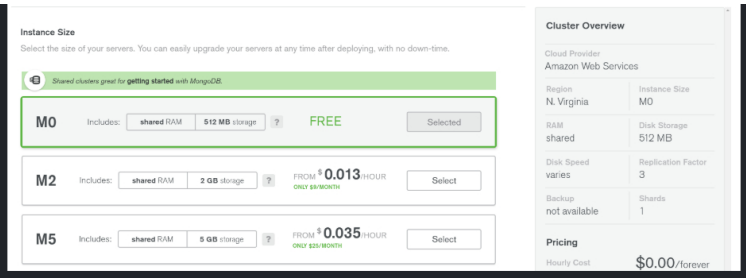
This will open up a page to name your project. Just type in `rest-api` once again and hit next.

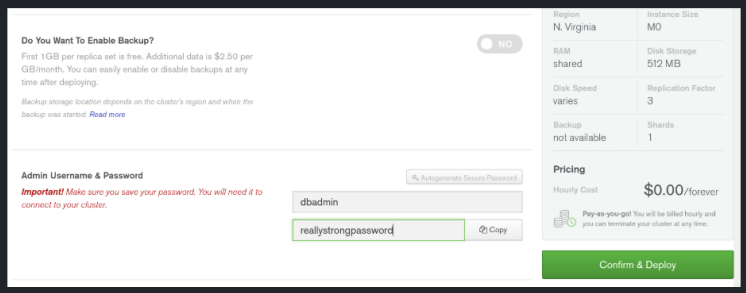


MongoDB cares about permissions and security so Atlas will show you another manage permissions page. We can just skip that for now, and create the project.



Phew, there we have it. Finally, we can create the actual cluster! Press on the huge green **"Build a new cluster"** button. This will open up a huge cluster creation window. You can leave everything default, just make sure to pick the **M0** instance size, and disable backups.





After all that, just add an admin user for the cluster and give him a really strong password. As you can see the price for this cluster will be **$0.00/forever**. Quite nice. That's it, hit **"Confirm & Deploy"**.

Your cluster will take a few minutes to deploy. While that's underway, let's finally start writing some code.

Writing some code

That setup was a handful. Now we need to jump into writing the resource configuration in the **serverless.yml** file, and add the actual CRUD methods to the **handler.js**.

4. Configure all the YAML

The awesomeness of the Serverless framework lies in the great initial scaffolds. You can pretty much create a great configuration using only the commented out code in the **serverless.yml** file. But, because I'm a sucker for cleanliness, let's just delete it all and add the code below. After you copy it into your **serverless.yml** file I'll go ahead and explain it all.

```
service: rest-api

provider:
  name: aws
  runtime: nodejs6.10 # set node.js runtime
  memorySize: 128 # set the maximum memory of the Lambdas in Megabytes
  timeout: 10 # the timeout is 10 seconds (default is 6 seconds)
  stage: dev # setting the env stage to dev, this will be visible in the routes
  region: us-east-1

functions: # add 4 functions for CRUD
  create:
    handler: handler.create # point to exported create function in handler.js
    events:
      - http:
          path: notes # path will be domain.name.com/dev/notes
          method: post
          cors: true
  getOne:
```

```
    handler: handler.getOne
    events:
      - http:
          path: notes/{id} # path will be domain.name.com/dev/notes/1
          method: get
          cors: true
  getAll:
    handler: handler.getAll # path will be domain.name.com/dev/notes
    events:
     - http:
         path: notes
         method: get
         cors: true
  update:
    handler: handler.update # path will be domain.name.com/dev/notes/1
    events:
     - http:
         path: notes/{id}
         method: put
         cors: true
  delete:
    handler: handler.delete
    events:
     - http:
         path: notes/{id} # path will be domain.name.com/dev/notes/1
         method: delete
         cors: true

plugins:
- serverless-offline # adding the plugin to be able to run the offline emulation
```

This configuration is bare bones and just enough for our needs. We've set the maximum memory size of the Lambdas to 128MB which is more than enough for our needs. After testing them on my own for a couple of days, they never went over 50MB.

Let's get to the interesting stuff, the **functions** section. We added a total of 5 functions: **create**, **getOne**, **getAll**, **update**, and **delete**. They all point to identically named exported functions in the **handler.js** file. Their paths all follow the naming convention of a standard REST API. Amazing how this is all we need to set up the API Gateway resources to trigger our Lambda functions.

That's pretty much it, the last thing is to add a plugins section and **serverless-offline**. We installed this module above and we'll use it for testing the service out before deploying to AWS. I guess we're ready to play with the **handler.js** next. Let's go!

5. Fleshing out the functions

We're ready to have some real fun now. We'll first define the 5 functions we need and create the initial layout of the behavior we want. After that, we can create the database connection and add the database interaction logic with Mongoose.

First of all open up the **handler.js** file. You'll see the default hello function. Go ahead and delete it all and add the code below.

```
'use strict';
```

```
module.exports.create = (event, context, callback) => {
  context.callbackWaitsForEmptyEventLoop = false;

  connectToDatabase()
    .then(() => {
      Note.create(JSON.parse(event.body))
        .then(note => callback(null, {
          statusCode: 200,
          body: JSON.stringify(note)
        }))
        .catch(err => callback(null, {
          statusCode: err.statusCode || 500,
          headers: { 'Content-Type': 'text/plain' },
          body: 'Could not create the note.'
        }));
    });
};

module.exports.getOne = (event, context, callback) => {
  context.callbackWaitsForEmptyEventLoop = false;

  connectToDatabase()
    .then(() => {
      Note.findById(event.pathParameters.id)
        .then(note => callback(null, {
          statusCode: 200,
          body: JSON.stringify(note)
        }))
        .catch(err => callback(null, {
          statusCode: err.statusCode || 500,
          headers: { 'Content-Type': 'text/plain' },
          body: 'Could not fetch the note.'
        }));
    });
};

module.exports.getAll = (event, context, callback) => {
  context.callbackWaitsForEmptyEventLoop = false;

  connectToDatabase()
    .then(() => {
      Note.find()
        .then(notes => callback(null, {
          statusCode: 200,
          body: JSON.stringify(notes)
        }))
        .catch(err => callback(null, {
          statusCode: err.statusCode || 500,
          headers: { 'Content-Type': 'text/plain' },
          body: 'Could not fetch the notes.'
        }))
    });
};

module.exports.update = (event, context, callback) => {
  context.callbackWaitsForEmptyEventLoop = false;

  connectToDatabase()
    .then(() => {
      Note.findByIdAndUpdate(event.pathParameters.id, JSON.parse(event.body), { new: true })
        .then(note => callback(null, {
          statusCode: 200,
          body: JSON.stringify(note)
        }))
        .catch(err => callback(null, {
          statusCode: err.statusCode || 500,
          headers: { 'Content-Type': 'text/plain' },
          body: 'Could not fetch the notes.'
        }));
    });
};
```

```
module.exports.delete = (event, context, callback) => {
  context.callbackWaitsForEmptyEventLoop = false;

  connectToDatabase()
    .then(() => {
      Note.findByIdAndRemove(event.pathParameters.id)
        .then(note => callback(null, {
          statusCode: 200,
          body: JSON.stringify({ message: 'Removed note with id: ' + note._id, note: note })
        }))
        .catch(err => callback(null, {
          statusCode: err.statusCode || 500,
          headers: { 'Content-Type': 'text/plain' },
          body: 'Could not fetch the notes.'
        }));
    });
};
```

(Step 1 of adding logic to the handler.js)

Okay, it's fine to get a bit overwhelmed. But, there's no need to worry. These are just 5 simple functions. Every function has the same value of `context.callbackWaitsForEmptyEventLoop` set to `false`, and start with the `connectToDatabase()` function call. Once the `connectToDatabase()` function resolves it'll continue with executing the database interaction through Mongoose. We'll be using the `Note` model methods for the actual database interaction. But wait, we haven't defined or created any of this! You must be asking yourselves what's wrong with me. Well I did it on purpose, I first want you to see that this is not that complicated, nor any different from creating a REST API with Node.js and Express.

Note: `context.callbackWaitsForEmptyEventLoop`—*By default, the callback will wait until the Node.js runtime event loop is empty before freezing the process and returning the results to the caller. You can set this property to false to request AWS Lambda to freeze the process soon after the* `callback` *is called, even if there are events in the event loop. AWS Lambda will freeze the process, any state data and the events in the Node.js event loop (any remaining events in the event loop processed when the Lambda function is called next and if AWS Lambda chooses to use the frozen process).*
- AWS Documentation

The time has come to add the actual database connection. What's important to understand before we add the code is that the connection will be established once. When the Lambda is invoked for the first time, which is called a cold start, AWS will spin up a Docker container to run the code. This is when we connect to the database. All subsequent requests will use the existing database connection. Conceptually it's rather easy to understand, but a real handful when we need to wrap our heads around it in the code. Here it goes.

6. Adding the database connection

The process of connecting to MongoDB is twofold. We need to create a dynamic way of creating the connection but also make sure to re-use the same connection if it's available. We'll start slow.

Create a new file in the root directory of the service, right alongside the **handler.js**. Give it a pretty logical name of **db.js**, and add the code below.

```
const mongoose = require('mongoose');
mongoose.Promise = global.Promise;
let isConnected;

module.exports = connectToDatabase = () => {
  if (isConnected) {
    console.log('=> using existing database connection');
    return Promise.resolve();
  }

  console.log('=> using new database connection');
  return mongoose.connect(process.env.DB)
    .then(db => {
      isConnected = db.connections[0].readyState;
    });
};
```

***Note**: This syntax is valid for Mongoose 5.0.0-rc0 and above. It will not work with any version of Mongoose which is lower than 5.*

On line 1 we're requiring Mongoose, just as we're used to, and on line 2 adding the native promise library to be used by Mongoose. This is because we want the .thens to work properly in the **handler.js** when we call them with the Note model methods.

What about the `isConnected` variable then? We're creating a closure, and treating `isConnected` as the current database state in the running Docker container. Take a look at the `connectToDatabase` function which we export. On line 12 we're establishing a connection with a connection string we'll provide through an environment variable. This function returns a promise which we simply .then and get a db object back. This object represents the current connection and has one property of particular interest to us. The `.readyState` will tell us whether a connection exists or not. If yes, it'll equal `1` otherwise it's `0`.

We're basically caching the database connection, making sure it'll not get created if it already exists. In that case we just resolve the promise right away.

With the **db.js** file created, let's require it in the **handler.js**. Just add this snippet to the top of the handler.

```
// top of handler.js
const connectToDatabase = require('./db');
const Note = require('./models/Note');
```

7. Adding a Note model

Take another look at the handler.js. You can see we're calling the Note model in the functions to retrieve data, but there's no model defined. Well, now is as good a time as any.

Create a new folder in the service root directory and name it **models**. In it create another file and name it **Note.js**. This will be just a simple mongoose schema and model definition.

```
const mongoose = require('mongoose');
const NoteSchema = new mongoose.Schema({
  title: String,
  description: String
});
module.exports = mongoose.model('Note', NoteSchema);
```

We'll export the model itself so we can use it in the **handler.js**. That's it regarding database connectivity. We just need to add another require statement to the top of the handler and we're good to go.

```
// top of handler.js
const connectToDatabase = require('./db');
const Note = require('./models/Note');
```

Great, now what's left is to add an environment variable to hold our MongoDB database connection URL. That's a breeze with **dotenv**.

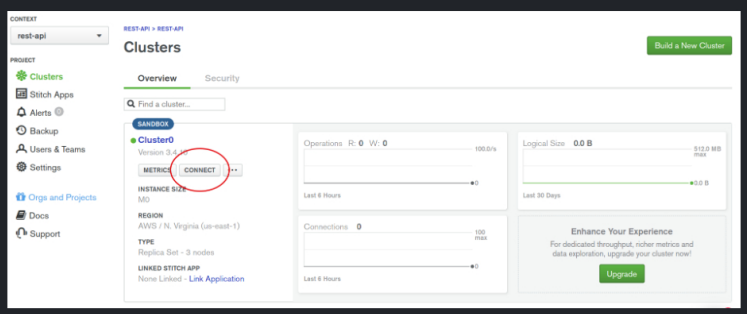
8. Using dotenv for environment variables

Leaving config files and keys in a totally separate file is incredibly easy with dotenv, and a real life saver. You just add the file to **.gitignore** and be sure you won't risk compromising any keys. Let me show you.

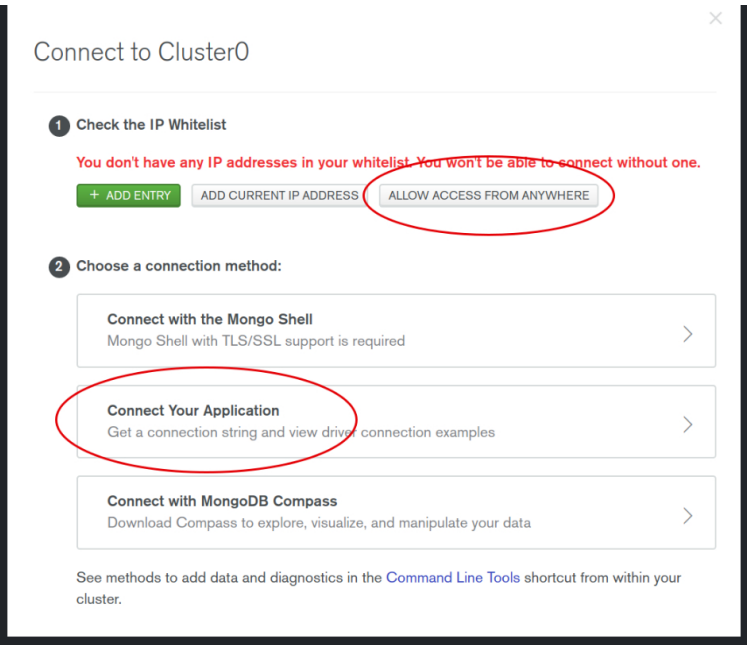
Add a new file, call it **variables.env**. Make sure to put it in the root directory of the service. The file itself will only have one line, and that's the name of the environment variable alongside the value. It should look somewhat like this.

```
DB=mongodb://<user>:<password>@mongodb.net:27017/db
```

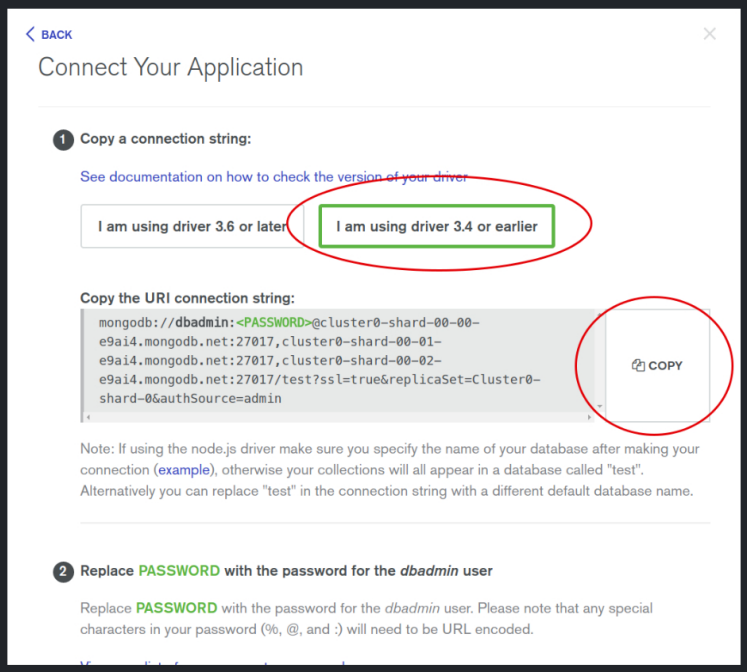
But, first we need to find the connection URL. For that, we need to go back to Atlas. On the main clusters page of the project you created earlier you'll see your cluster has been created. It has a connect button we want to press.



It'll open up a new popup where you need to add an IP address to the whitelist, so you can access the database at all. Then you grab the connection URL by pressing the**"Connect Your Application"** button.



After you've pressed **"Connect Your Application"** you'll be prompted to "**Copy a connection string"**. Press "**I am using driver 3.4 or earlier"** and you can FINALLY copy the URL. Whoa, that was a tiresome ride.



Once you've copied it, head back to the **variables.env** file and add the actual connection URL.

```
DB=mongodb://dbadmin:reallystrongpassword@cluster0-shard-00-00-e9ai4.mongodb.net:27017,cluster0-shard-00-01-e9ai4.mongodb.net:27017
```

Make sure not to add spaces between the **DB** and the connection URL. Change `<password>` for the password you set earlier. Mine was "reallystrongpassword". What will happen now? Well, the variables in this file will get loaded into the `process.env` object in Node.js, meaning you can access them in the standard way you're already used to.

***Note**: Don't forget to add the variables.env to the .gitignore!*

Lastly, before jumping into testing everything out, we need to require the dotenv module and point to the file where we're keeping the environment variables. Add this snippet to the top of your **handler.js** file.

```
require('dotenv').config({ path: './variables.env' });
```

That's it. Time to try it out.

How about some testing?

We're ready to test the API. First of all we need to run Serverless Offline. But, because of the Mongoose model definition we have in the **Note.js** there's a flag we need to add while running it.

```
$ sls offline start --skipCacheInvalidation
```

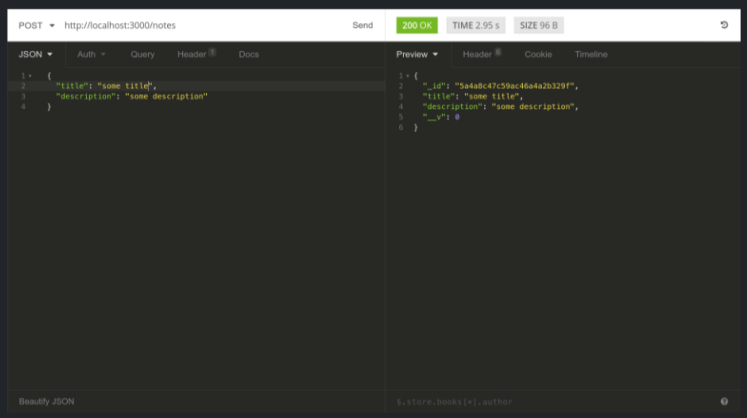
***Note**: Because Serverless Offline invalidates the Node require cache on every run by default, we add this flag to disable it. In Node.js when you require() a module, it stores a cached version of the module, so that all subsequent calls to require() do not have to reload the module from the file system.*

Once you've run the command in the terminal, you should see something like this.

```
Serverless: Starting Offline: dev/us-east-1.

Serverless: Routes for create:
Serverless: POST /notes

Serverless: Routes for getOne:
Serverless: GET /notes/{id}

Serverless: Routes for getAll:
Serverless: GET /notes

Serverless: Routes for update:
Serverless: PUT /notes/{id}

Serverless: Routes for delete:
Serverless: DELETE /notes/{id}

Serverless: Offline listening on http://localhost:3000
```

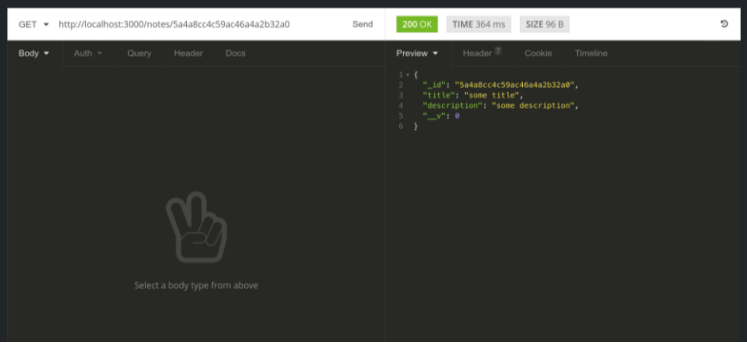
All our routes are up and running. Open up your REST client of choice, Postman, Insomnia or whatever you prefer, and let's get on with the testing.

Using Insomnia, I've created a POST request to http://localhost:3000/notes with a JSON body.

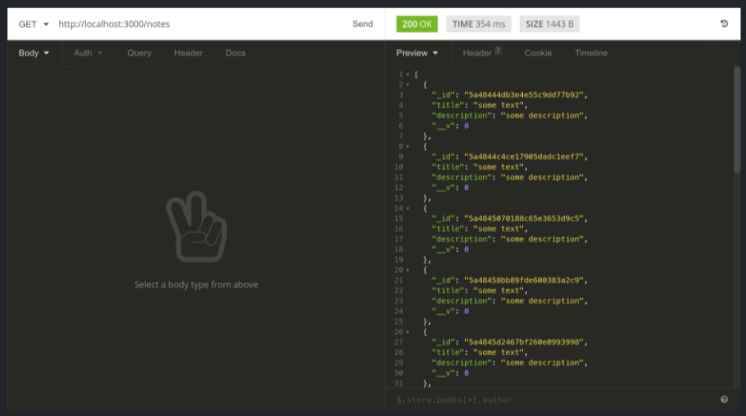


Checking the terminal you can see `=> using new database connection` get logged, meaning the initial database connection has been established. Send another POST request and you'll see `=> using existing database connection` get logged instead.

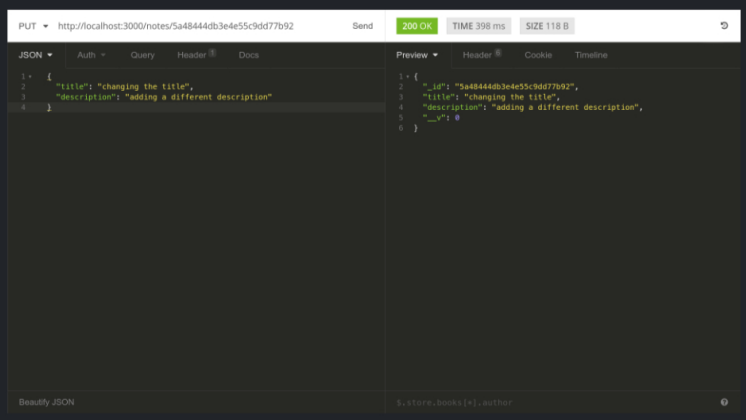
Awesome, adding a new note works. Let's retrieve the note we just added using the **getOne** method. Copy the `_id` from the reponse and paste it into the URL of the GET request.



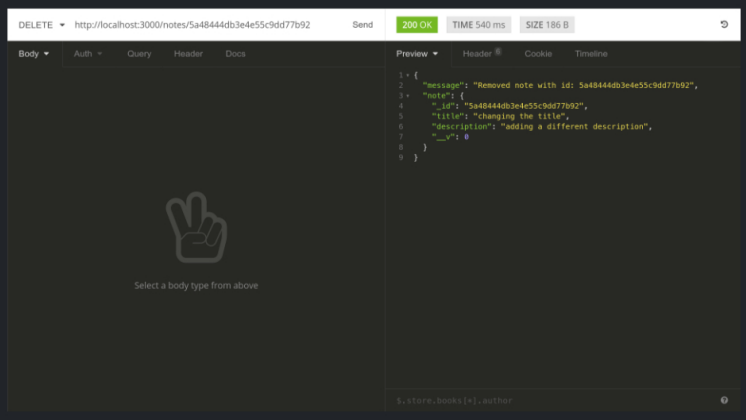
Retrieving a single note works fine as well. What about retrieving them all. Just delete the ID route path parameter and hit **“Send”** once again.



Only two more lest to try out, the edit and delete methods. Pick one of the _ids from the retrieved notes and add it as a path parameter once again. Now change the method to PUT and add a JSON body. Enter a different title and description and hit **“Send”**.



The editing works fine, just as we wanted. Only the delete left. Change to the DELETE method, remove the request body and hit **“Send”** one last time.



The note was successfully deleted. That’s more than enough regarding the testing. We’re ready to deploy the service to AWS.

Being responsible with deployment and monitoring

Phew, that's a lot of stuff you need to wrap your head around. We're at the home stretch. Only thing left is to deploy the service and make sure it's behaving the way we want by using a monitoring tool called Dashbird.

9. Deployment

The Serverless framework makes deployments quick and painless. All you need to do is to run one command.

```
$ sls deploy
```

It will automagically provision resources on AWS, package up and push all code to S3 from where it will be sent to the Lambdas. The terminal should show output similar to this.

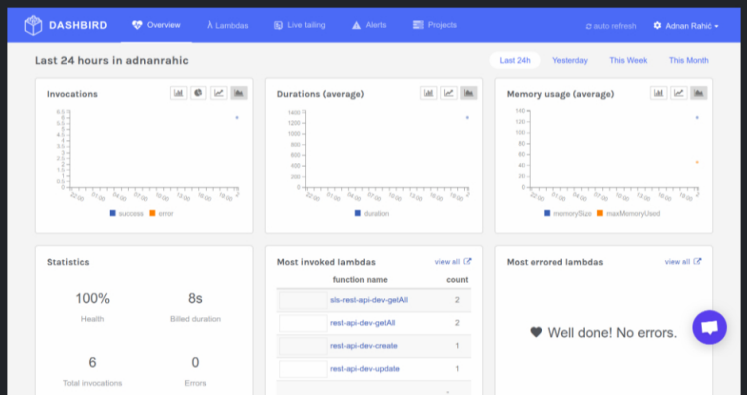
```
.....................................................................................
Serverless: Stack update finished...
Service Information
service: rest-api
stage: dev
region: us-east-1
stack: rest-api-dev
api keys:
  None
endpoints:
  POST - https://uh3b3d502m.execute-api.us-east-1.amazonaws.com/dev/notes
  GET - https://uh3b3d502m.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  GET - https://uh3b3d502m.execute-api.us-east-1.amazonaws.com/dev/notes
  PUT - https://uh3b3d502m.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  DELETE - https://uh3b3d502m.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
functions:
  create: rest-api-dev-create
  getOne: rest-api-dev-getOne
  getAll: rest-api-dev-getAll
  update: rest-api-dev-update
  delete: rest-api-dev-delete
Serverless: Publish service to Serverless Platform...
```

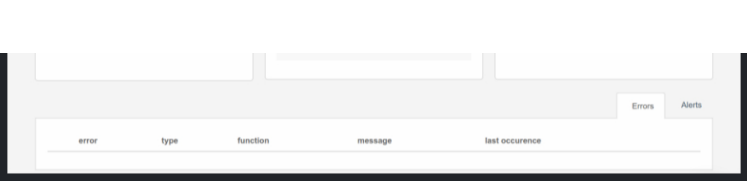
***Note**: You can repeat the testing process from above with the endpoints provided.*

That's all there is to the deployment process. Easy right? This is why I love the Serverless framework so much.

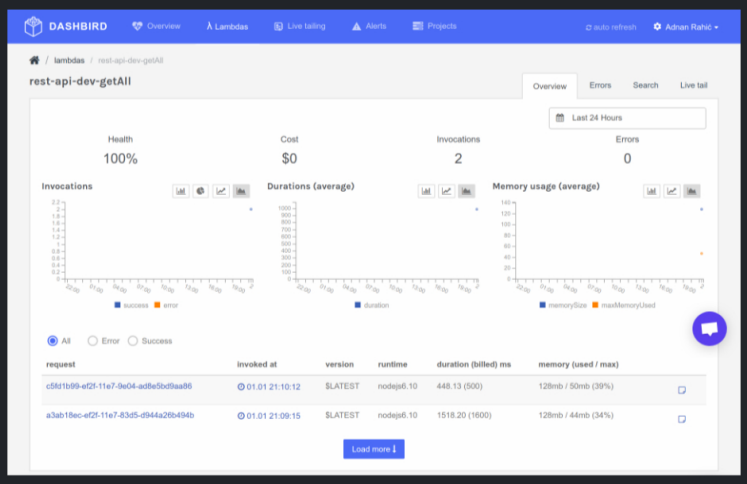
10. Monitoring

Let's wrap this is up with another cool tool. I monitor my Lambdas with Dashbird, and I'm loving it. It has a free tier and doesn't require a credit card to sign up! My point for showing you this is for you too see the console logs from the Lambda function invocations. They'll show you when the Lambda is using a new or existing database connection. Here's what the main dashboard looks like, where I see all my Lambdas and their stats.

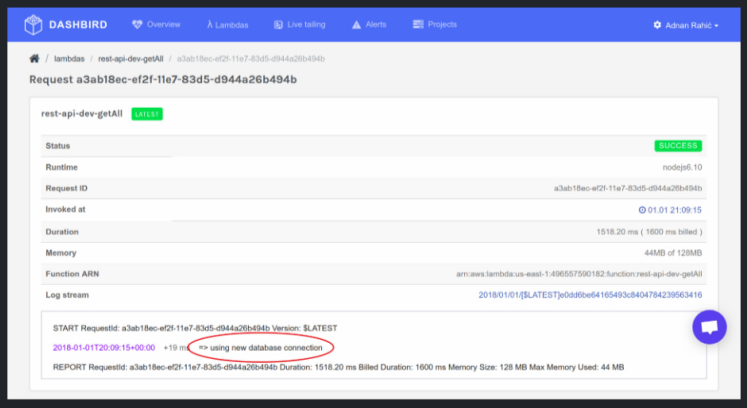




After pressing on the **rest-api-dev-getAll** Lambda function I'll be taken to a screen with all the stats and logs for this particular function.

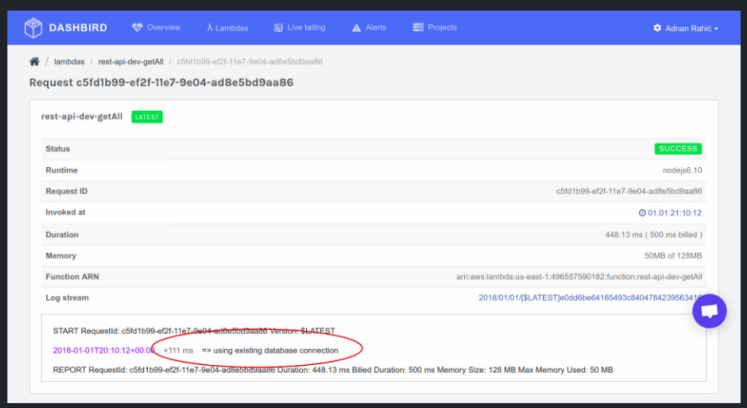


At the bottom you see two invocation of the getAll function. After pressing on the older of the two, it takes me to another page showing info about that particular invocation.



As you can see the console was logged with => using new database connection and the actual request took roughly 1.5 seconds.

Moving back and pressing on the other invocation we can see a similar but still, luckily for us, a different image.



Once the same Lambda function has been invoked again, it will re-use the existing connection. It can be clearly seen in the logs here.

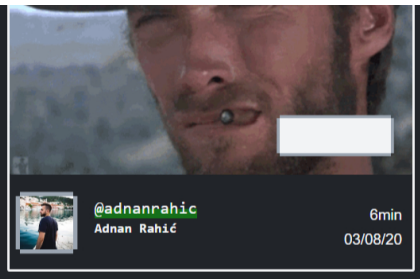
End of the line

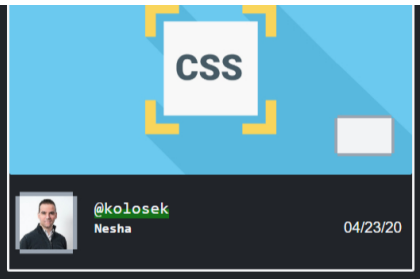
What an emotional roller coaster. You've been taken on a trip to creating a Serverless REST API with MongoDB. I've tried my best to transfer the experience I've gathered until today to show you the preferred way of creating a proper API. Many of the techniques I've shown are what I use myself on a daily basis. Use these skills wisely and enjoy digging deeper into the possibilities of Serverless Architecture and all that comes with it.

If you want to take a look at all the code we wrote above, here's the repository. Or if you want to read my latest articles, head over here.

Latest stories written by Adnan Rahić - Medium
*Read the latest stories written by Adnan Rahić on Medium. Software engineer @bookvar_co. Coding educator @ACADEMY387…*medium.com

*Hope you guys and girls enjoyed reading this as much as I enjoyed writing it. Do you think this tutorial will be of help to someone? Do not hesitate to share. If you liked it, smash the **clap** below so other people will see this here on Medium.*





Tags

THE NOON NOTIFICATION

Subscribe to get your daily round-up of top tech stories!