

使用 ServerLess, Nodejs, MongoDB Atlas cloud 构建 REST API

MongoDB Atlas 是一个在云端的数据库，免去了数据库的搭建、维护，通过其提供的 Web UI 能够让你在 5 分钟之内快速搭建一个 Clusters。Node.js 是一个 JavaScript 的运行时，在 JavaScript 中函数做为一等公民，享有着很高的待遇，通常使用 Node.js 我们可以快速的搭建一个服务，而 ServerLess 是一种“无服务器架构”，从技术角度来讲是 FaaS 和 BaaS 的结合，FaaS（Function as a Service）是一些运行函数的平台。

那么通过这些可以做些什么呢？本篇文章中我们将使用 ServerLess、MongoDB Atlas cloud 与 Node.js 的结合来快速构建一个 REST API，无论你是前端工程师还是后端工程师，只要你掌握一些 JavaScript 基础语法就可以完成。

作者简介：五月君，Nodejs Developer，慕课网认证作者，热爱技术、喜欢分享的 90 后青年，欢迎关注公众号「[Nodejs技术栈](#)」和 Github 开源项目 <https://www.nodejs.red>

MongoDB Atlas cloud

为什么选择 MongoDB Atlas cloud？这是一个运行在云端的数据库，无需安装、配置，也无需在我们的机器上安装 Mongo 服务，只需要一个 URL 即可访问数据库，还提供了非常酷的 UI 界面，易于使用。最重要的一点对于我们初学者来说它提供了免费使用，最大限制为 512 MB，这对于小型项目是足够的。

集群创建

1. 注册：<https://www.mongodb.com/cloud/atlas/register?v=2>
2. 注册成功进入个人中心出现以下页面，选择 Build a Cluster 按钮 创建 Cluster



Create a cluster

Choose your cloud provider, region, and specs.

Build a Cluster

Once your cluster is up and running, live migrate an existing MongoDB database into Atlas with our [Live Migration Service](#).

3. 以下提供了多种选择方案，对于初学者选择左侧免费版创建

Choose a path. Adjust anytime.

Available as a fully managed service across 60+ regions on AWS, Azure, and Google Cloud

Starter Clusters

For teams learning MongoDB or developing small applications.

- ✓ Highly available auto-healing cluster
- ✓ End-to-end encryption
- ✓ Role-based action control
- ✗ No downtime scaling
- ✗ Network isolation
- ✗ Realtime performance metrics

Starting at
FREE

Create a cluster

Single-Region Clusters

For teams building applications that need advanced development and production-ready environments.

- ✓ Includes all features from Starter Clusters
- ✓ No downtime scaling
- ✓ Network isolation
- ✓ Realtime performance metrics

Starting at
\$0.08/hr*
*estimated cost \$56.94/month

Create a cluster

Multi-Region Clusters

For teams developing world-class applications that require multi-region resiliency or ultra-low latency.

- ✓ Includes all features from Starter and Single-Region Clusters
- ✓ Replicate data across multiple regions

Global Clusters



- ✓ Globally distributed read and write operations
- ✓ Control data residency at the document level

Starting at
\$0.13/hr*
*estimated cost \$98.55/month

Create a cluster

4. 可以看到免费的集群给我们提供了 512MB 存储、共享的 RAM，这对我们初学者是绰绰有余的，还有一些其它选项可以自主选择，使用默认值也可。

Create a Starter Cluster

Welcome to MongoDB Atlas! We've recommended some of our most popular options, but feel free to customize your cluster to your needs. For more information, check our [documentation](#).

Cloud Provider & Region

AWS, N. Virginia (us-east-1) >

Cluster Tier

M0 Sandbox (Shared RAM, 512 MB Storage) >
Encrypted

Additional Settings

MongoDB 4.0, No Backup >

Cluster Name

Cluster0 >

5. 点击下面的 Create Cluster 按钮，开始集群创建，大概需要等待几分钟

FREE

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Back

Create Cluster

6. 创建成功如下所示

CONTEXT

test

ATLAS

Clusters

Data Lake BETA

SECURITY

Database Access

Network Access

Advanced

PROJECT

Access Management

Activity Feed

Alerts 0

Integrations

Settings

QUZHENFEI > TEST

Clusters

Find a cluster...

SANDBOX

Cluster0

Version 4.0.13

CONNECT METRICS COLLECTIONS ...

CLUSTER TIER

M0 Sandbox (General)

REGION

AWS / N. Virginia (us-east-1)

TYPE

Replica Set - 3 nodes

LINKED STITCH APP

None Linked

Operations R: 0 W: 0

100.0/s

0

Last 6 Hours

Connections 0

500 max

0

Last 6 Hours

链接到集群

1. 链接到集群，第一步将您的 IP 地址加入白名单，第二步创建一个 MongoDB 用户，完成这两步操作之后，选择 “Choose a connection method” 进入下一步

Connect to Cluster0

Setup connection security > Choose a connection method > Connect

You need to secure your MongoDB Atlas cluster before you can use it. Set which users and IP addresses can access your cluster now. [Read more](#)

You can't connect yet. Set up your firewall access and user security permission below.

1 Whitelist your connection IP address

IP Address Description (Optional)

Enter IP Address or CIDR Notation 2 An optional comment describing this entry

Cancel Add IP Address 3

2 Create a MongoDB User

This first user will have [atlasAdmin](#) permissions for this project. Keep your credentials handy, you'll need them for the next step.

Username Password

ex. dbUser ex. dbUserPassword SHOW

Create MongoDB User 5

2. 选择第二个选项 “connect your application”

Connect to Cluster0

✓ Setup connection security > Choose a connection method > Connect

Choose a connection method [View documentation](#)

See methods to add data and diagnostics in the [Command Line Tools](#) shortcut from within your cluster.

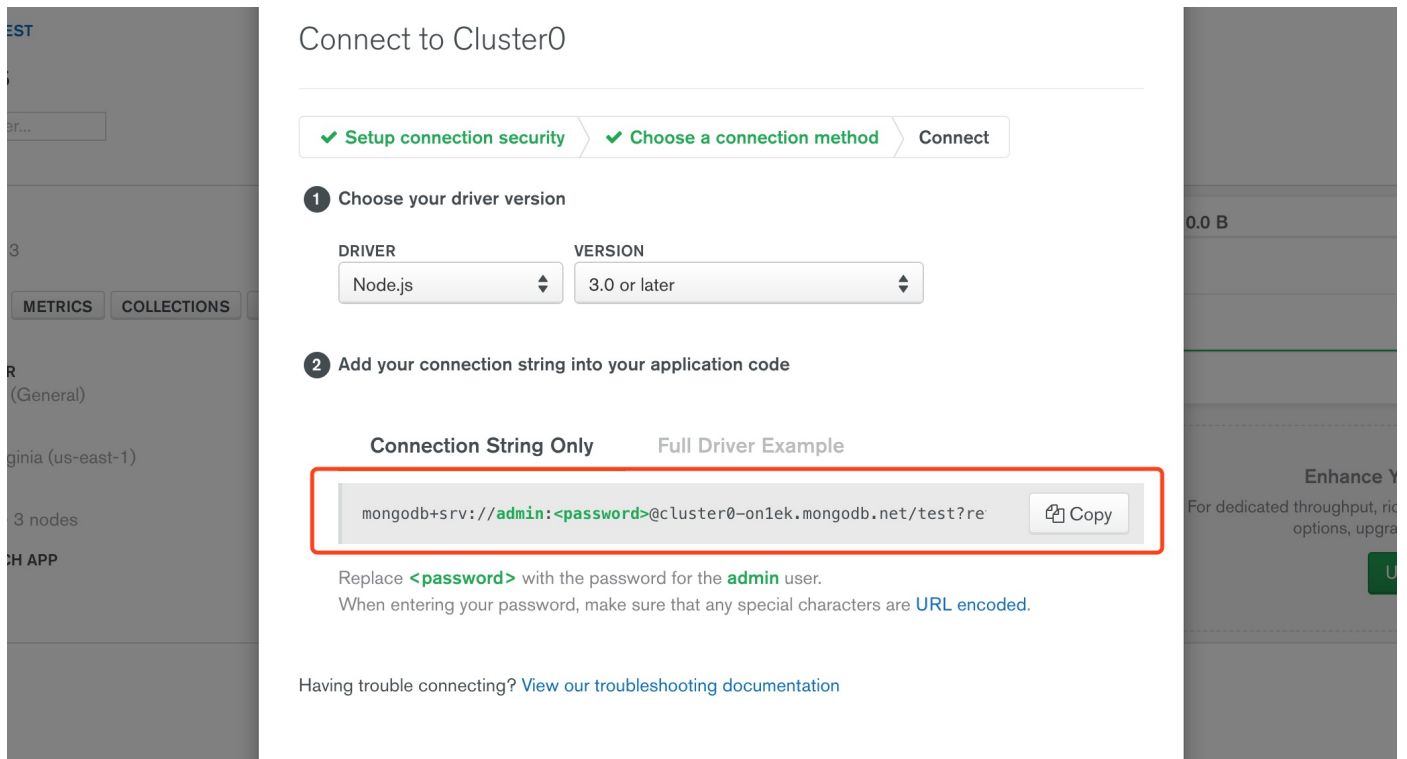
Connect with the Mongo Shell
Mongo Shell with TLS/SSL support is required

Connect Your Application
Get a connection string and view driver connection examples

Connect with MongoDB Compass
Download Compass to explore, visualize, and manipulate your data

Go Back Close

3. 驱动版本使用默认值 Node.js 3.0 or later，复制这个链接字符串，接下来的项目中会使用到



开启一个 Serverless Function 链接到 DB

1. 项目创建、插件安装

创建项目，安装 mongodb、serverless-offline 插件，这些都是基础内容，如果你是 ServerLess 初学者可以先去学习下这篇文章 [使用 Node.js 快速开启 ServerLess Functions：入门实践指南](#)

```
$ serverless create --template hello-world --path mongodb-serverless-conn-test
$ npm init
$ npm i mongodb -S
$ npm i serverless-offline --save-dev
```

2. 项目根目录下创建 db.js 文件

数据库链接字符串就是上面 MongoDB Atlas cloud 链接集群中所讲的，注意替换你的用户名和密码，以下代码中 initialize 函数接收两个参数 dbName、dbCollectionName 用来初始化一个 connection。

```

// db.js
const MongoClient = require("mongodb").MongoClient;
const dbConnectionUrl = 'mongodb+srv://<user>:<password>@cluster0-on1ek.mongodb.net/test'

async function initialize(
  dbName,
  dbCollectionName,
) {
  try {
    const dbInstance = await MongoClient.connect(dbConnectionUrl);
    const dbObject = dbInstance.db(dbName);
    const dbCollection = dbObject.collection(dbCollectionName);

    console.log("[MongoDB connection] SUCCESS");
    return dbCollection;
  } catch (err) {
    console.log(`[MongoDB connection] ERROR: ${err}`);

    throw err;
  }
}

module.exports = {
  initialize,
}

```

3. 修改 handler.js

我们想要测试下 MongoDB 的连接，以下是一个 ServerLess Function 我们在该函数中初始化了一个 Connection 然后调用了 find() 方法查找集合数据

```
// handler.js
'use strict';
const db = require('./db');

module.exports.find= async (event, context) => {
  const response = {
    statusCode: 200,
  };

  try {
    const dbCollection = await db.initialize('study', 'books2');
    const body = await dbCollection.find().toArray();

    response.body = JSON.stringify({
      code: 0,
      message: 'SUCCESS',
      data: body,
    });

    return response;
  } catch (err) {
    response.body = JSON.stringify({
      code: err.code || 1000,
      message: err.message || '未知错误'
    });

    return response;
  }
};
```

4. 测试

```
# 启动本地调试
$ serverless offline

# 接口测试
$ curl http://localhost:3000/find
Serverless: GET /find (λ: find)
[MongoDB connection] SUCCESS
[]
```

似乎一切都是 Ok 的，证明我们的集群创建、链接都是成功的，但是有时候你可能会遇到以下错误

```
Error: querySrv ENODATA _mongodb._tcp.cluster0-on1ek.mongodb.net
```

以上正是我在链接 MongoDB Atlas 过程中遇到的问题，这里再多提下，希望能对你有帮助，因为这花费了我很长时间，尝试使用 Google、Stackoverflow ... 来搜索，但并没有找到好的解决方案，通过报错

大致确认可能是网络 和 DNS 的问题，修改 DNS 之后还是没有结果，后来我切换了网络，这个问题解决了。。。如果你有答案欢迎和我讨论，另外也建议检查链接字符串和 MongoDB Alats 白名单是否设置的正确。

两个问题

以上例子虽然已经简单的完成了一个方法，但是它其实是糟糕的，从而引发以下两个问题：

- 1. 业务逻辑与 FaaS、BaaS 严重的耦合不利于单元测试、平台迁移：**上面这个例子是不好的，业务逻辑完全的写在了 handler.js 文件的 find 函数中，一方面 find 函数的 event、context 对象是由 FaaS 平台提供的，另一方面 db 属于后端服务，这就造成了业务逻辑与 FaaS、BaaS 严重的耦合。
- 2. 不利于上下文重用：**传动程序启动之后常驻内存，不存在冷启动问题，而 ServerLess 是基于事件驱动的，第一次请求来了之后会下载代码、启动容器、启动运行环境、执行代码，这个过程称为冷启动，但是以 AWS Lambda 为例，函数调用之后执行上下文会被冻结一段时间，在我们上面的例子中每次函数执行都会初始化数据库链接，这是一个很消时的操作，我们可以将这段逻辑放在函数之外，利用上下文重用，在开发层面可以做进一步优化。

Serverless REST API 开发最佳实践

带着上面提出的几点问题，本节将对这个业务逻辑进行重构，开发一个 REST API 最佳实践。

REST API 规划

以下是我们将要完成的 REST API 规划，包含四个 CRUD 操作

CRUD	API Routes	Description
POST	/books	增加一本书
GET	/books	获取所有书籍列表
PUT	/books/:id	根据 id 更新指定编号书籍
DELETE	/books/:id	根据 id 删除指定编号书籍

目录规划

一个好的项目离不开一个好的目录规划，当然你也可以按照自己思路来做


```
mongodb-serverless-node-rest-api
├── package.json
├── .env
├── serverless.yml
├── app
│   ├── handler.js
│   ├── controller
│   │   └── books.js
│   └── model
│       ├── db.js
│       ├── books.js (可选)
│       └── utils
│           └── message.js
└── test
    ├── controller
    └── books.test.js
```

项目创建、插件安装

这一次我没有直接使用 MongoDB 驱动，而用的 mongoose 来代替 MongoDB 操作。

```
$ serverless create --template hello-world --path mongodb-serverless-node-rest-api
$ npm init
$ npm i dotenv mongoose -S
$ npm i serverless-offline --save-dev
```

创建 .env 配置文件

将配置独立出来放入 .env 配置文件，统一管理。

```
DB_URL=mongodb+srv://admin:admin123456@cluster0-on1ek.mongodb.net/test?retryWrites=true&w=majority
DB_NAME=study1
DB_BOOKS_COLLECTION=books
```

创建 Model

app/model/db.js

```
const mongoose = require('mongoose');
mongoose.connect(process.env.DB_URL, {
  dbName: process.env.DB_NAME,
});
```

app/model/books.js

Mongoose 的一切始于 Schema。每个 schema 都会映射到一个 MongoDB collection，定义这个 collection 里的文档构成。

```
const mongoose = require('mongoose');
const BooksSchema = new mongoose.Schema({
  name: String,
  id: { type: Number, index: true, unique: true },
  createdAt: { type: Date, default: Date.now },
});

module.exports = mongoose.models.Books || mongoose.model('Books', BooksSchema, process.env
```

使用 mongoose 创建 model，serverless-offline 运行之后调用多次，可能会出现以下问题

```
OverwriteModelError: Cannot overwrite `Books` model once compiled.
```

这个错误是因为你已经定义了一个 Schema，之后又重复定义该 Scheme 导致的，错误代码如下所示：

```
module.exports = mongoose.model('Books', BooksSchema, process.env.DB_BOOKS_COLLECTION);
```

解决这个问题，一种方案是要保证仅实例化一次，正确代码如下所示，另一种是在 serverless offline 之后加上 --skipCacheInvalidation 参数跳过 require 缓存无效操作，详情可参见 [serverless-offline/issues/258](#)。

```
module.exports = mongoose.models.Books || mongoose.model('Books', BooksSchema, process.env
```

编写业务逻辑 Books

将业务逻辑处理放在 Books 这个类里面，并且可以不依赖于外部的任何服务，this.BooksModel 这个在测试时可以模拟数据进行传入。做到 **业务逻辑与 FaaS、BaaS 的分离**。

app/controller/books.js

```

const message = require('../utils/message');

class Books {
  constructor(BooksModel) {
    this.BooksModel = BooksModel;
  }

  /**
   * 创建 Book 数据
   * @param {*} event
   */
  async create(event) {
    const params = JSON.parse(event.body);

    try {
      const result = await this.BooksModel.create({
        name: params.name,
        id: params.id,
      });

      return message.success(result);
    } catch (err) {
      console.error(err);

      return message.error(err.code, err.message);
    }
  }

  /**
   * 更新
   * @param {*} event
   */
  async update(event) {
    try {
      const result = await this.BooksModel.findOneAndUpdate({ id: event.pathParam }, {
        $set: JSON.parse(event.body),
      }, {
        $upsert: true,
        new: true
      });

      return message.success(result);
    } catch (err) {
      console.error(err);

      return message.error(err.code, err.message);
    }
  }

  /**
   * 查找所有 Books 数据

```

```

    * @param {*} event
    */
    async find() {
        try {
            const result = await this.BooksModel.find();

            return message.success(result);
        } catch (err) {
            console.error(err);

            return message.error(err.code, err.message);
        }
    }

    /**
     * 删除一条数据
     * @param {*} event
     */
    async deleteOne(event) {
        try {
            const result = await this.BooksModel.deleteOne({
                id: event.pathParameters.id
            });

            if (result.deletedCount === 0) {
                return message.error(1010, '数据未找到! 可能已被删除! ');
            }

            return message.success(result);
        } catch (err) {
            console.error(err);

            return message.error(err.code, err.message);
        }
    }
}

module.exports = Books;

```

编写 handler

在 handler 里 event、context 这些参数是由 FaaS 平台提供，上面我们又把业务逻辑单独放置于 Controller 下的 Books.js 里，这样做好处是假如我们要从一个平台迁移到另一个平台，只需要修改 handler.js 里 Books 的调用方式即可，业务逻辑是不受影响的。

对于这种初始化链接的操作，尽量放在函数之外，避免每次函数来临都要去初始化这样一个耗时的操作，我们可以利用函数的执行上下文重用，在启动环境执行代码时去初始化我们的数据库链接，例如 handler.js 头部的 require('./model/db')。

app/handler.js

```
require('dotenv').config();
require('./model/db');

const BooksModel = require('./model/books');
const BooksController = require('./controller/books');
const booksController = new BooksController(BooksModel);

module.exports = {
  create: event => booksController.create(event),

  update: event => booksController.update(event),

  find: () => booksController.find(),

  deleteOne: event => booksController.deleteOne(event),
}
```

Serverless 配置文件

这个也是重点，plugins 插件的 serverless-offline 是为了本地调试用，functions 里面则定义了函数文件的路径和路由规则，注意如果是 /books/:id 这样的路由在 serverless.yml 里的路由规则为 books/{id}

```
service: mongodb-serverless-node-rest-api
```

```
provider:
```

```
  name: aws
```

```
  runtime: nodejs12.x
```

```
plugins:
```

```
  - serverless-offline
```

```
functions:
```

```
  create:
```

```
    handler: app/handler.create
```

```
    events:
```

```
      - http:
```

```
        path: books
```

```
        method: post
```

```
  update:
```

```
    handler: app/handler.update
```

```
    events:
```

```
      - http:
```

```
        path: books/{id}
```

```
        method: put
```

```
  find:
```

```
    handler: app/handler.find
```

```
    events:
```

```
      - http:
```

```
        path: books
```

```
        method: get
```

```
  deleteOne:
```

```
    handler: app/handler.deleteOne
```

```
    events:
```

```
      - http:
```

```
        path: books/{id}
```

```
        method: delete
```

部署

部署这是我们完成 REST API 的最后一步，就是这么简单一条命令 `serverless deploy` 就可将我们的服务部署在云端。

```
$ serverless deploy
Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service mongodb-serverless-node-rest-api.zip file to S3 (2.17 MB)...
Serverless: Stack update finished...
...
endpoints:
  POST - https://*****.execute-api.us-east-1.amazonaws.com/dev/books
  PUT - https://*****.execute-api.us-east-1.amazonaws.com/dev/books/{id}
  GET - https://*****.execute-api.us-east-1.amazonaws.com/dev/books
  GET - https://*****.execute-api.us-east-1.amazonaws.com/dev/books/{id}
  DELETE - https://*****.execute-api.us-east-1.amazonaws.com/dev/books/{id}
```

endpoints 列举了服务的接口调用地址，现在你可以在 POSTMAN 进行调试了。

完整代码参考源码地址

<https://github.com/Q-Angelo/project-training/tree/master/serverless/mongodb-serverless-node-rest-api>

总结

ServerLess 是一种全新的技术体系，降低了服务端研发成本，而 Node.js 使用起来很轻量级，对前端开发者也很友好，但是前端开发者对服务端运维还是相对陌生的，使用了 ServerLess 可以帮助开发者隔离服务器的运维、环境搭建等一系列操作，把更多时间聚焦在业务开发中。本文中在数据存储方面结合了 MongoDB Atlas Cloud 免去了数据库的搭建、维护工作，现在只要你掌握一些 JavaScript 基础语法通过本文的讲解就可轻松的完成一个 REST API，这是多么 Nice 的事情呀，快来实践下吧！