

## Tutorial - Create a Database Cluster in the Cloud With MongoDB Atlas

For years now, MongoDB has been the go-to NoSQL database for both individuals and enterprises building large-scale applications. It's open source, easily scalable, and provides high availability. It also supports very complex queries and fine-grained concurrency control.

However, necessary tasks such as installing the database, tuning it to maintain optimal performance over long periods of time, and securing it tend to require a lot of skilled and dedicated effort.

Fortunately, there's an easier alternative: MongoDB Atlas, a fully managed, cloud version of the database.

With MongoDB Atlas, you can create a MongoDB cluster on any major cloud provider of your choice and start using that cluster in a matter of minutes. Using Atlas's browser-based user interface, you can also intuitively configure the cluster and monitor its performance.

In this tutorial, I'll show you how to create a MongoDB Atlas free tier cluster and use it in a Python application.

## Prerequisites

Before you proceed, make sure you have the following installed and configured on your computer:

- Python 3.4 or higher
- pip 18.0 or higher

# 1. Creating a Cluster

To be able to use MongoDB's cloud services, you'll need a MongoDB Atlas account. To create one, go to its [home page](#) and press the **Get started free** button.



After you complete the short sign-up form, you'll be redirected to the cluster creation wizard. In its first section, you'll have to pick the cloud provider and region you prefer.

To minimize network latency, you'd ideally pick a region that's nearest to your computer. For now, however, because we are creating a free tier cluster, make sure the region you select is one that has a free tier available. Additionally, if you are using a Google Cloud VM or an Amazon EC2 instance as your development environment, do select the corresponding cloud provider first.

## Cloud Provider &amp; Region

GCP, Iowa



Create a **free tier cluster** by selecting a region with **FREE TIER AVAILABLE** and choosing the **M0** cluster tier below

★ recommended region ⓘ

## NORTH AMERICA / SOUTH AMERICA

🇺🇸 South Carolina (us-east1) ★

🇺🇸 N. Virginia (us-east4) ★

🇺🇸 Iowa (us-central1) ★

FREE TIER AVAILABLE

## EUROPE / MIDDLE EAST / AFRICA

🇧🇪 Belgium (europe-west1) ★

FREE TIER AVAILABLE

🇬🇧 London (europe-west2) ★

🇩🇪 Frankfurt (europe-west3) ★

## ASIA PACIFIC

🇹🇼 Taiwan (asia-east1) ★

🇯🇵 Tokyo (asia-northeast1) ★

🇸🇬 Singapore (asia-southeast1) ★

FREE TIER AVAILABLE

**\$0.44/hour**

**Pay-as-you-go!** You will be billed hourly and can terminate your cluster anytime. Excludes variable [data transfer](#), [backup](#), and taxes.

Cancel




In the **Cluster Tier** section, select the **M0** option to create your free tier cluster. It offers 512 MB of storage space, a recent version of MongoDB with WiredTiger as the storage engine, a replica set of three nodes, and a generous 10 GB of bandwidth per week.

## Cluster Tier

M0 (Shared RAM, 512 MB Storage)

Base hourly rate is for a MongoDB replica set with **3 data bearing servers**.

### Shared Clusters

 <b>M0</b>	Shared RAM	512 MB Storage	Shared vCPUs	
<b>M2</b>	Shared RAM	2 GB Storage	Shared vCPUs	from  ONLY
<b>M5</b>	Shared RAM	5 GB Storage	Shared vCPUs	from  ONLY

### Dedicated Development Clusters

<b>M10</b>	1.7 GB RAM	10 GB Storage	0.5 vCPUs	from
<b>M20</b>	3.75 GB RAM	20 GB Storage	1 vCPU	from

### Dedicated Production Clusters

**FREE**

**Pay-as-you-go!** You will be billed hourly and can terminate your cluster anytime. Excludes variable **data transfer**, **backup**, and taxes.

Cancel

Lastly, give a meaningful name to the cluster and press the **Create Cluster** button.

M200	240 GB RAM	1500 GB Storage	64 vCPUs	from
M300	360 GB RAM	2200 GB Storage	96 vCPUs	from

[PREVIOUS: CLOUD PROVIDER & REGION](#)[NEXT](#)

## Additional Settings

MongoDB 3.6

### Cluster Name

**One time only:** once your cluster is created, you won't be able to change its name.

Cluster names can only contain ASCII letters, numbers, and hyphens.

**FREE**

**Pay-as-you-go!** You will be billed hourly and can terminate your cluster anytime. Excludes variable **data transfer**, **backup**, and taxes.

[Cancel](#)

MongoDB Atlas will now take about five minutes to set up your cluster.

Advertisement

## 2. Configuring the Cluster

Before you start using the cluster, you'll have to provide a few security-related details, so switch to the **Security** tab.

First, in the **MongoDB Users** section, you must create a new user for yourself by pressing the **Add new user** button. In the dialog that pops up, type in your desired username and password, select the **Read and write to any database** privilege, and press the **Add User** button.

**Add New User**

**SCRAM Authentication**  
SCRAM is MongoDB's default authentication method.

alice  
e.g. new-user\_31

.....

Autogenerate Secure Password

**User Privileges**

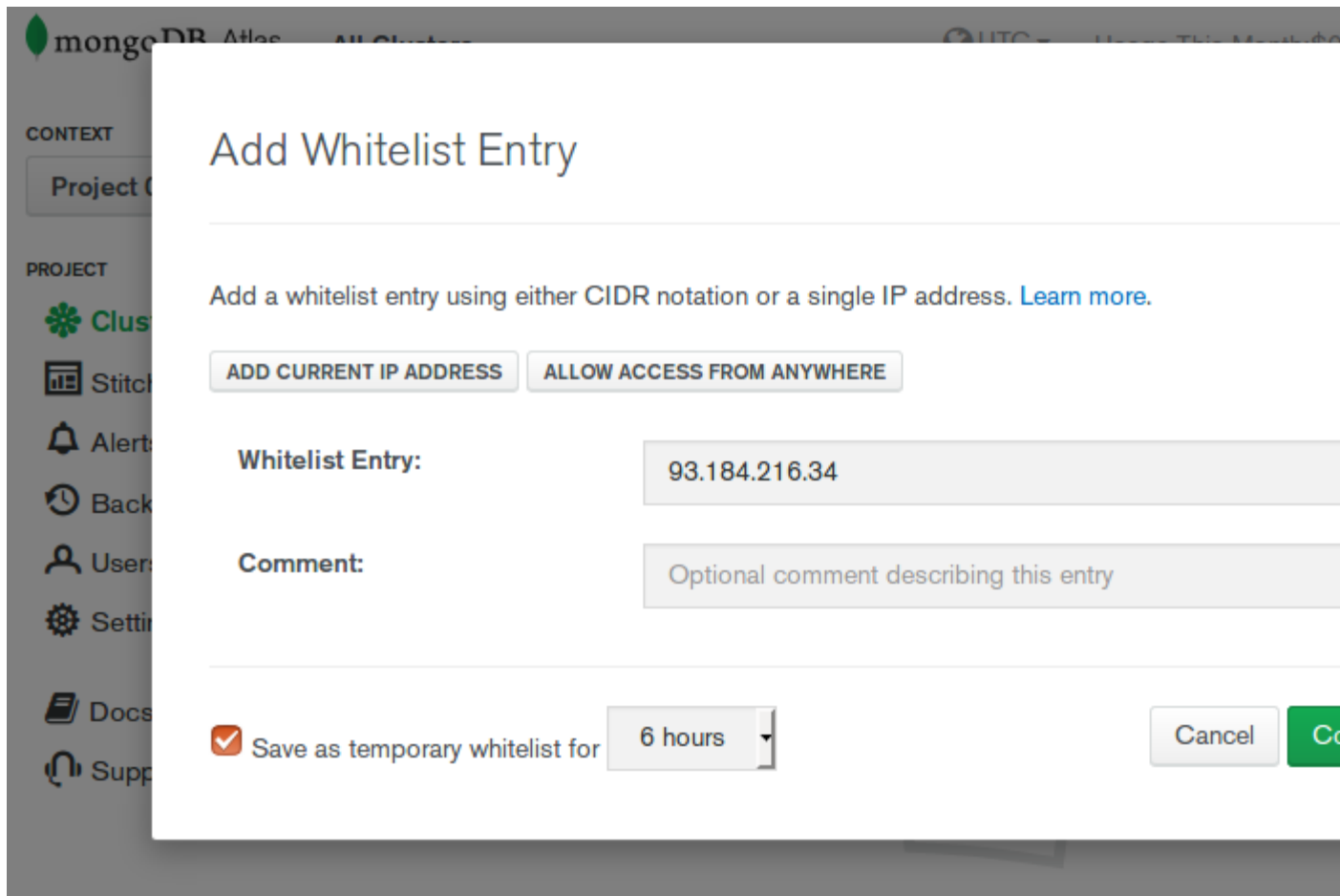
Atlas admin    **Read and write to any database**    Only read any data

Show Advanced Options

☐ Save as temporary user    Cancel

Next, in the **IP Whitelist** section, you must provide a list of IP addresses from which you'll be accessing the cluster. For now, providing the current IP address of your computer is sufficient.

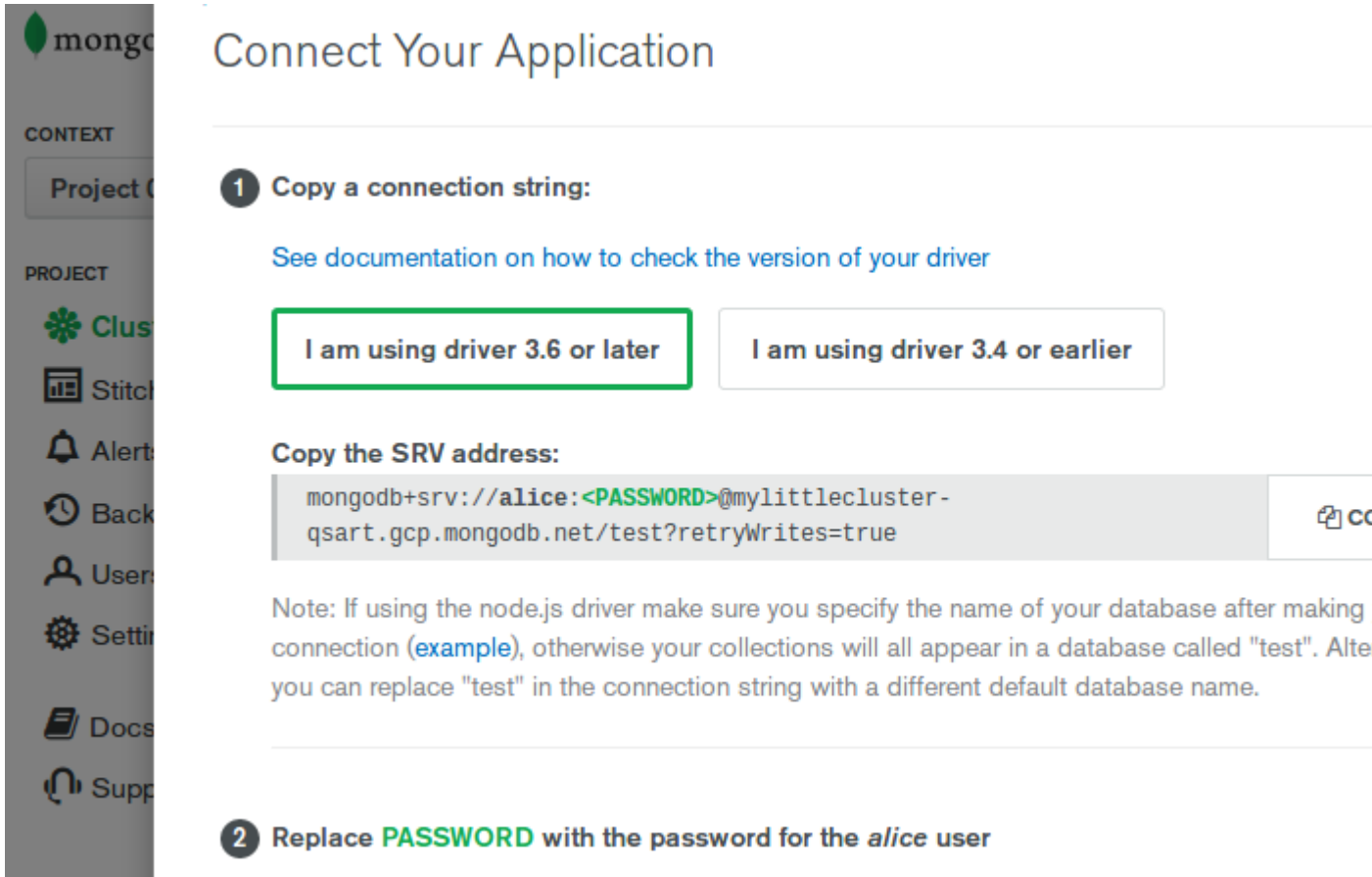
Press the **Add IP address** button to create a new IP address entry. In the dialog that pops up, press the **Add current IP address** button to autofill the **Whitelist Entry** field. Additionally, if you don't have a static IP address, it's a good idea to mark it is a temporary entry by checking the **Save as temporary whitelist** option. Finally, press **Confirm** to add the entry.



### 3. Getting the Connection String

You'll need a valid connection string to connect to your cluster from your application. To get it, go to the **Overview** tab and press the **Connect** button.

In the dialog that opens, select the **Connect Your Application** option and press the **I'm using driver 3.6 or later** button. You should now be able to see your connection string. It won't have your actual password, so you'll have to put it in manually. After you do so, make a note of the string so you can use it later.



**1 Copy a connection string:**

[See documentation on how to check the version of your driver](#)

**I am using driver 3.6 or later**    **I am using driver 3.4 or earlier**

**Copy the SRV address:**

```
mongodb+srv://alice:<PASSWORD>@mylittlecluster-qsart.gcp.mongodb.net/test?retryWrites=true
```

Note: If using the node.js driver make sure you specify the name of your database after making connection ([example](#)), otherwise your collections will all appear in a database called "test". Alternatively you can replace "test" in the connection string with a different default database name.

**2 Replace **PASSWORD** with the password for the *alice* user**

## 4. Installing the Python Driver

To be able to interact with your MongoDB Atlas cluster programmatically, you must have a [MongoDB driver](#) installed on your computer. For the Python programming language, [PyMongo](#) is the most popular driver available today. The recommended way to install it on your computer is to use the `pip` module as shown below:

```
1 python3 -m pip install pymongo --user
```

You may have noticed that your MongoDB Atlas connection string is a `mongodb+srv://` URI. To enable the driver to work with DNS SRV records, you must also install the `dnspython` module. Here's how:

```
1 python3 -m pip install dnspython --user
```



## 5. Connecting to the Cluster

You can now use your MongoDB cluster from any Python application. To follow along with me, create a new Python script and open it using any code editor.

Inside the script, to be able to interact with the cluster, you'll need an instance of the `MongoClient` class. As the only argument to its constructor, pass your connection string.

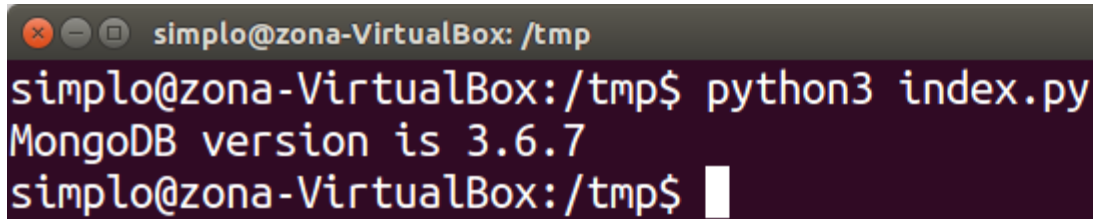
```
1  import pymongo
2
3  my_client = pymongo.MongoClient(
4      'mongodb+srv://alice:myPassword@mylittlecluster-qsart.gcp.mongodb.net/test?retry
5  )
```

The above constructor returns immediately and will not raise any errors. Therefore, to check if you've successfully established a connection, I suggest you try to perform an operation on the cluster. A call to the `server_info()` method, which gets you various details about your MongoDB instance, will suffice.

If there are no errors in your connection string, the call to the `server_info()` method will succeed. However, if the username or password you specified is incorrect, you'll encounter an `OperationFailure` error. The following code shows you how to catch it:

```
1  try:
2      print("MongoDB version is %s" %
3          my_client.server_info()['version'])
4  except pymongo.errors.OperationFailure as error:
5      print(error)
6      quit(1)
```

You can now go ahead and try running your script.

A terminal window with a dark background. The title bar shows a window icon, a close button, and the text 'simplo@zona-VirtualBox: /tmp'. The terminal content shows a prompt 'simplo@zona-VirtualBox:/tmp\$' followed by the command 'python3 index.py'. The output is 'MongoDB version is 3.6.7'. The prompt returns to 'simplo@zona-VirtualBox:/tmp\$' with a cursor.

```
simplo@zona-VirtualBox: /tmp
simplo@zona-VirtualBox:/tmp$ python3 index.py
MongoDB version is 3.6.7
simplo@zona-VirtualBox:/tmp$
```

## 6. Inserting Documents

The default connection string you get from MongoDB Atlas's web interface mentions a database named `test`. Let's continue to use the same database. Here's how you can get a reference to it:

```
1 my_database = my_client.test
```

A MongoDB database is composed of one or more collections, which are nothing but groups of [BSON documents \(short for binary JSON\)](#). Your free tier cluster on MongoDB Atlas can have a maximum of 500 collections.

For the sake of a realistic example, let's create a new collection named `foods`. With PyMongo, you don't have to explicitly call any method to do so. You can simply reference it as if it exists already.

```
1 my_collection = my_database.foods
```

It's worth mentioning that the above code doesn't create the `foods` collection immediately. It's created only after you add a document to it. So let's now create and add a new document containing nutritional data about a food item.

Using the `insert_one()` method is the simplest way to add a document to a collection. To specify the contents of the document, you pass a Python dictionary to the method. The following sample code shows you how:

```
01 my_collection.insert_one({
02     "_id": 1,
```

```
03     "name": "pizza",
04     "calories": 266,
05     "fats": {
06         "saturated": 4.5,
07         "trans": 0.2
08     },
09     "protein": 11
10 })
```

Adding documents one at a time can be inefficient. With the `insert_many()` method, you can add several documents to your collection at once. It expects an array of dictionaries as its input. The following code adds two more documents to the collection:

```
01
02 my_collection.insert_many([
03     {
04         "_id": 2,
05         "name": "hamburger",
06         "calories": 295, "protein": 17,
07         "fats": { "saturated": 5.0, "trans": 0.8 },
08     },
09     {
10         "_id": 3,
11         "name": "taco",
12         "calories": 226, "protein": 9,
13         "fats": { "saturated": 4.4, "trans": 0.5 },
14     }
15 ])
```

The `_id` field you see in the above documents is a field that's used as a primary key by MongoDB. As such, it is immutable and must be present in all MongoDB documents. If you forget to include it while creating your document, PyMongo will add it for you automatically and assign an auto-generated unique value to it.

## 7. Running Queries

When you've added a few documents to your collection, you can run queries on it by calling the `find()` method, which returns a `Cursor` object you can iterate over. If you don't pass any arguments to it, `find` returns all the documents in the collection.

The following code shows you how to print the names of all the food items present in our collection:

```
1  my_cursor = my_collection.find()
2
3  for item in my_cursor:
4      print(item["name"])
5
6  # Output is:
7  #   pizza
8  #   hamburger
9  #   taco
```

If you want the `find()` method to return only those documents that match specific criteria, you must pass a Python dictionary to it. For example, if you want to find the document whose `name` field is set to "pizza", you could use the following code:

```
1  my_cursor = my_collection.find({
2      "name": "pizza"
3  })
```

For more complex queries, you can use MongoDB's [intuitively named query operators](#) in the dictionaries you pass to the `find()` method. For instance, the following code shows you how to use the `$lt` operator to find documents whose `calories` field is set to a value that's less than 280:

```
01  my_cursor = my_collection.find({
02      "calories": { "$lt": 280 }
```

```

03     })
04
05     for item in my_cursor:
06         print("Name: %s, Calories: %d" %
07               (item["name"], item["calories"]))
08
09     # Output is:
10     #   Name: pizza, Calories: 266
11     #   Name: taco, Calories: 226

```

By using the dot notation, you can also use nested fields in your queries. The following code shows you how to find documents whose `trans` field, which is inside the `fats` field, is set to a value that's greater than or equal to 0.5:

```

01     my_cursor = my_collection.find({
02         "fats.trans": { "$gte": 0.5 }
03     })
04
05     for item in my_cursor:
06         print("Name: %s, Trans fats: %.2f" %
07               (item["name"], item["fats"]["trans"]))
08
09     # Output is:
10     #   Name: hamburger, Trans fats: 0.80
11     #   Name: taco, Trans fats: 0.50

```

## 8. Updating and Deleting Documents

Very similar to the `insert_one()` and `insert_many()` methods are the `update_one()` and `update_many()` methods, which you can use to change the contents of documents that are already inside your collection. Both the

update methods, in addition to new data, need a query to zero in on the documents that need to be changed.

You can use a variety of update operators in your update methods. The most commonly used one is `$set`, which lets you add new fields or update the values of existing fields. The following code shows you how to add two new fields named `fiber` and `sugar` to the document whose `name` field is set to "taco":

```
1 my_collection.update_one(  
2     { "name": "taco" }, # query  
3     {  
4         "$set": {          # new data  
5             "fiber": 3.95,  
6             "sugar": 0.9  
7         }  
8     }  
9 )
```

If the query you pass to the `update_one()` method returns more than one document, only the first document is updated. The `update_many()` method doesn't have this limitation.

Lastly, by using the `delete_one()` and `delete_many()` methods, you can delete documents in your collections. Both the methods need a query to determine which documents need to be deleted. Here's how you can delete all documents whose `calories` field is set to a value that's less than 300:

```
1 my_collection.delete_many({  
2     "calories": {  
3         "$lt": 300  
4     }  
5 })  
6  
7 # Deletes all the three documents
```

# Conclusion

If you are a web or mobile application developer who wants to use MongoDB for storing data, the MongoDB Atlas service is for you. It lets you focus on developing your application instead of worrying about details such as security, performance, and adherence to best practices. In this tutorial, you learned how to create a MongoDB cluster using the service, connect to it, and perform basic read and write operations on it.