Storing Data in a MongoDB Database

# Introduction

There are two main categories of databases: ***relational databases*** and ***non-relational databases***. Relational databases such as ***MySQL***, ***SQL Server***, and ***Postgre***, store data in ***tables*** containing ***records*** of the same type, e.g., a ***tuits table*** would contain all the tuits, and a ***users table*** would contain all the users of an application. Records are represented as ***rows*** in the tables where each ***column*** stores data for attributes peculiar to the type of the table, e.g., the rows in the tuits table might have columns such as ***post***, ***postedBy***, ***postedOn***, ***likes***, ***liked***, ***comments***, ***retuits***, etc. Some of the columns might refer, or ***relate*** to other records in other tables such as the ***postedBy*** column in the tuits table might refer to, or relate to a particular row in the ***users*** table signifying that that particular user posted that particular tuit. Rows in one table relating to rows in another table is where ***relational*** databases get their name. The ***structured query language*** or ***SQL***, is the computer language commonly used to interact with relational databases. The ***query*** in ***SQL*** generally means to ***ask for***, or ***retrieve*** data that matches some criteria, often written as a ***boolean expression*** or ***predicate***.

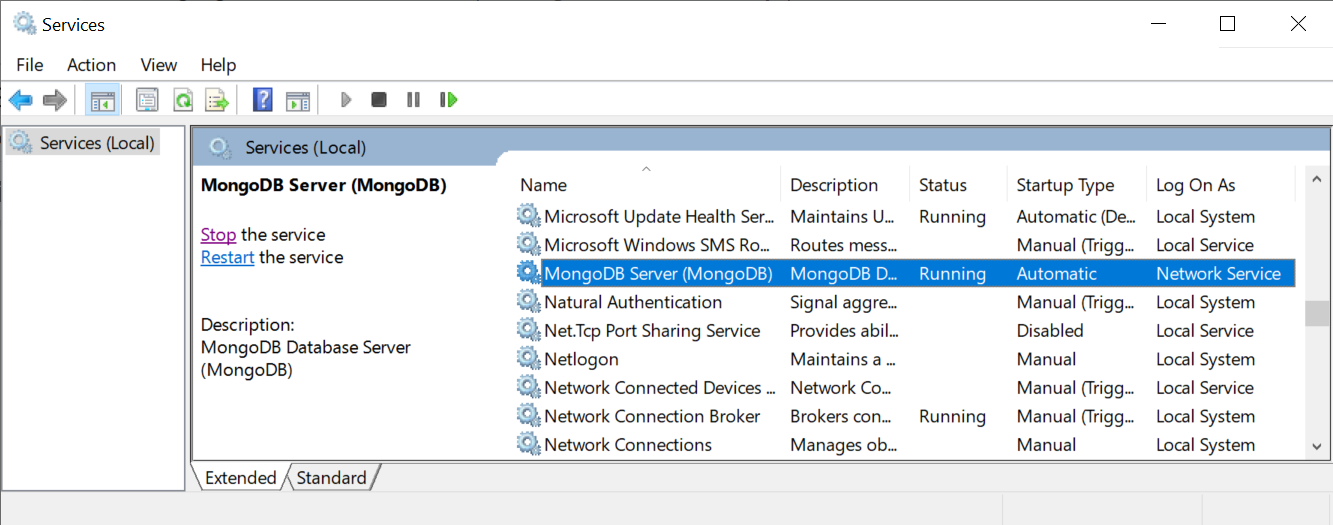
More recently there has been a growing interest in representing and storing data using alternative strategies which have collectively come to be referred to as ***non relational databases***, or ***NoSQL databases***. Non relational databases such as ***MongoDB***, ***Firebase***, and ***Coushbase***, store their data in ***collections*** containing ***documents*** which are roughly analogous to ***tables*** and ***records*** in their relational conterparts. The biggest difference though is that the columns, or ***fields*** in the rows in relational databases generally can only contain ***primitive data types***, e.g., simple strings, numbers, dates, and booleans, whereas the fields in non relational documents can be arbitrarily ***complex data types***, e.g., strings, numbers, booleans, dates as well as combinations of these in complex objects containing arrays of objects of arrays, etc. The other big difference is that relational databases require the structure, or ***schema*** of the data be explicitly described before storing any data, whereas non relational databases do not require predefined schemas. Instead, non relational databases delegate this responsibility to the applications using the database. The structure, or schema in relational databases is where ***structured query language*** gets its name.

In the previous chapter we learned how to create an HTTP server with Node.js and integrated it with a React.js application to store the application state on the server. In this chapter we expand this idea to store the data to ***MongoDB***, a non relational database. The first section demonstrates how to download, install and use a local instance of the MongoDB database. The next section covers how to use the ***Mongoose*** library to integrate and program a MongoDB database with a Node.js server application. The final section describes how to deploy the database to ***Mongo Atlas***, a remote MongoDB database hosted as a cloud service.

# Working with a local MongoDB instance

MongoDB is one of an increasingly popular family of non relational databases. Data is stored in collections of documents usually formatted as JSON objects which makes it very convenient to integrate with JavaScript based frameworks such as Node.js and React.js. This section describes how to install, configure and how to get started using MongoDB.

## 2.1 Installing and configuring MongoDB

To get started, download MongoDB for free from <https://www.mongodb.com/try/download/community>. Choose the latest version, your operating system and click ***Download***. Run the installer and, if given the choice, choose to run the database as a service so that you don't have to bother having to restart the database sever everytime you login or restart your computer. The MongoDB database will automatically start whenever you start your computer. On Windows, confirm the database is running by searching for MongoDB in the ***Services*** dialog. On macOS, confirm the database is running by clicking the MongoDB icon in the ***Systems Preferences*** dialog. The service dialog gives you controls to start and stop the database, but it should already be configured to start automatically when you restart your computer.

Alternatively you can unzip the server to a local file system and add the right commands to your operating PATH. On ***macOS***, unzip the file into ***/usr/local*** which should create a directory such as ***/usr/local/mongodb-macos-x86\_64-5.0.3*** (your version might differ). To be able to execute the database related commands, add the path to the ***.bash\_profile*** located in your home directory. The ***.bash\_profile*** should have a new row that roughly looks as shown below. Your actual version might differ.

| ***~/.bash\_profile*** | |
| --- | --- |
| export PATH="$PATH:/usr/local/mongodb-macos-x86\_64-5.0.3/bin" | |

If the ***.bash\_profile*** does not exist in the home directory, create it as a plain text file, but no extensions and a period in front of it. Configure it as shown above and then ***restart your computer***.

On ***Windows***, unzip the file into ***C:\Program Files***. To configure environment variables on ***Windows*** press the ***Windows + R*** key combination to open the ***Run*** prompt, type ***sysdm.cpl*** and press ***OK***. In the ***System Properties*** window that appears, press the ***Advanced*** tab and then the ***Environment Variables*** button. In the ***Environment Variables*** configuration window select the ***Path*** variable and press the ***Edit*** button. Copy and paste the path of the ***bin*** directory in the ***mongodb*** directory you unzipped the MongoDB download, e.g., ***"C:\Program Files\mongodb-macos- x86\_64-5.0.3\bin"***. The actual path might differ. Press ***OK*** and ***restart the computer***.

## 2.2 Using MongoDB Compass to interact with the MongoDB database

Your installation should have installed MongoDB Compass, a user interface client to the MongoDB database. You can start the Compass from your applications folder, or search for it in your operating system's search feature. On macOS bring up Spotlight by pressing the magnifying glass on the top right menu bar, or press the Command and Spacebar. Type MongoDB Compass in the search bar and select the application from the result list. On Windows press the Window key to bring up the search field, type MongoDB Compass, and select the application from the result list. When Compass comes up, confirm that the connection string ***mongodb://localhost:27017*** appears in the New Connection screen, and press ***Connect*** to connect to MongoDB.

## 2.3 Creating a MongoDB database

Once you are connected to a running MongoDB server, click on ***Databases*** on the left side bar and then click the ***Create database*** button on the ***Databases*** tab. In the ***Create Database*** dialog that appears, name your database ***tuiter*** and your first collection as ***tuits***. Click ***Create Database*** to create the ***tuiter*** database.

|  |  |
| --- | --- |

## 2.4 Inserting and retrieving data into a MongoDB database with Compass

In MongoDB, data is organized into ***collections***, not ***tables*** like in relational databases. Data contained in collections are referred to as ***documents***, not ***records*** like in other databases. To create, or ***insert*** documents into a collection in a MongoDB database, select the database on the left sidebar and then select the collection you want to insert documents into. For instance, select the ***tuiter*** database and then the ***tuits*** collection as shown here on the right. On the right side, selected ***Add Data*** and then ***Insert Document***. In the ***Insert to Collection tuiter.tuits*** dialog that appears, insert the document as shown below. Click ***Insert*** to insert the document. Confirm the document inserted as expected.

|  |  |
| --- | --- |

Insert three more tuits into the ***tuits*** collection using the following JSON objects.

* {tuit: "Tesla stock falls below 200", likes: 0, liked: false }
* {tuit: "Starlink likely to go public", likes: 42069, liked: true}
* {tuit: "Model Y is most American made car", likes: 69420, liked: true}

Note the objects stored in the database have a primary key ***\_id*** automatically added by MongoDB when they were inserted. MongoDB primary keys are of type ***ObjectId*** and are created automatically by the database so your ***\_id*** values will differ from the ones shown above.

## 2.5 Interacting with a MongoDB database with a Command Line

Compass is a great graphical user interface to the MongoDB database, but there is also value to knowing how to interact with the database through a command line interface. At the bottom of the Compass window there's a ***\_MONGOSH*** window you can expand to type commands to the database. Let's practice a few commands to retrieve data on the command line. First select the database we want to interact with.

| **> use tuiter**  'switched to db tuiter' |
| --- |

All the documents in a collection can be retrieved using the ***find()*** command on a collection as shown below.

| **> db.tuits.find();** |
| --- |

Documents in a collection can be retrieved by pattern matching their properties. The example below illustrates how to retrieve documents by pattern matching their primary key ***\_id***, that is, retrieving the document whose ***\_id*** field matches ***ObjectId('6370104926906053f1597ce6')***.

| **> db.tuits.find({\_id: ObjectId('6370104926906053f1597ce6')})**  { \_id: ObjectId("6370104926906053f1597ce6"),  tuit: 'Tesla stock falls below 200',  likes: 0,  liked: false } |
| --- |

We can also pattern match any of the other fields individually or combined with other fields. The following example retrieves a document from the ***tuits*** collection whose ***liked*** property is equal to ***false***.

| **> db.tuits.find({liked: false})**  { \_id: ObjectId("6370104926906053f1597ce6"),  tuit: 'Tesla stock falls below 200',  likes: 0,  liked: false } |
| --- |

Here's another example retrieving a document from the ***tuits*** collection whose ***liked*** property is equal to ***true***.

| **> db.tuits.find({liked: true})**  { \_id: ObjectId("63700d2626906053f1597ce5"),  tuit: '100s of Starships land on Mars carrying 1000s of colonists',  likes: 69420, liked: true }  { \_id: ObjectId("6370106326906053f1597ce7"),  tuit: 'Starlink likely to go public',  likes: 42069, liked: true }  { \_id: ObjectId("6370108326906053f1597ce8"),  tuit: 'Model Y is most American made car',  likes: 69420, liked: true } |
| --- |

## 2.6 Exercises

| 1. Install MongoDB as described in this section 2. Create the database as described in this section 3. Start the database as described in this section | 1. Insert the data as described in this section 2. Update the data as described in this section 3. Delete the data as described in this section |
| --- | --- |

# Programming with a MongoDB database

In the previous section we practiced interacting with the MongoDB database through the Compass graphincal insterface as well as manually on the command line with ***MONGOSH***. This is all and good to make occasional simple queries to confirm the data behaves as expected, but to create applications we're going to need to interact with the database programmatically with libraries such as [***Mongoose***](https://mongoosejs.com/). The following sections describe how to install, configure, and connect a Node.js application to a MongoDB database server using the Mongoose library. The final section discusses how to configure the application to integrate to a MongoDB database hosted in the Atlas cloud service. Do all your work in a new GitHub branch called ***a9*** in both your React.js and Node.js projects.

## 3.1 Installing and connecting to a MongoDB database

The [***Mongoose***](https://mongoosejs.com/) library provides a set of operations and abstractions that enhance a MongoDB database and leverages the familiarity of the MONGOSH command line client. To use the Mongoose library, install it from the root of the Node.js project as shown below.

| **$ npm install mongoose** |
| --- |

To connect to the database server programmatically, import the Mongoose library and then use the ***connect*** function as shown below. The URL in the ***connect*** function is called the ***connection string*** and is currently referring to a MongoDB server instance running in the ***localhost*** machine (your current laptop or desktop) listening at port ***27017*** and the ***tuiter*** database existing in that server. In the following section we'll revisit the connection string and configure it to connect to a database server running in a remote machine hosted by Mongo's Atlas cloud service.

| ***app.js*** |  |
| --- | --- |
| **import** *mongoose* **from "mongoose"**;  mongoose.*connect*(**'mongodb://localhost:27017/tuiter**); | *// load the mongoose library*  *// connect to the* ***tuiter*** *database* |

## 3.2 Creating a Mongoose schema and model

As mentioned earlier, non relational database do not require specifying the structure, or schema of the data stored in collections. That responsibility has been delegated to the applications using the non relational databases. Mongoose ***schemas*** describe the structure of the data being stored in the database and it's used to validate the data being stored of modified through the Mongoose library. The ***tuits-schema*** shown below describes the structure for the ***tuits*** collection we worked on earlier.

| ***tuits/tuits-schema.js*** |  |
| --- | --- |
| **import** mongoose **from 'mongoose'**;  **const *schema*** = mongoose.***Schema***({  **tuit**: ***String***,  **likes**: ***Number***,  **liked**: ***Boolean***,  }, {**collection**: **'tuits'**});  **export default *schema***; | *// load the mongoose library*  *// create the schema*  *// tuit property of type String*  *// likes property of type Number*  *// liked property of type Boolean*  *// collection name where tuits are stored in tuiter database*  *// export schema so it can be used elsewhere* |

In earlier sections we demonstrated using the command line client to interact manually with the mongo server using the ***find*** command. Mongoose ***models*** provide similar functions to interact with MongoDB programmatically instead of manually. The functions are similar to the ones found in the mongo shell client: ***find()***, ***create()***, ***updateOne()***, ***removeOne()***, etc. In ***/tuits/tuits-model.js*** below, create a Mongoose model from the tuit schema. The functions provided by Mongoose models are deliberately generic because they can interact with any collection configured in the schema. In the next section we'll create a ***data access object*** that implements higher level functions specific to the domain of tuits.

| ***tuits/tuits-model.js*** |  |
| --- | --- |
| **import** mongoose **from 'mongoose'**;  **import *tuitsSchema* from './tuits-schema.js'**  **const *tuitsModel*** = mongoose  .*model*(**'TuitModel'**, ***tuitsSchema***);  **export default *tuitsModel***; | *// load mongoose library*  *// load tuits schema*  *// create mongoose model from the schema*  *// export so it can be used elsewhere* |

## 3.3 Retrieving data from Mongo with Mongoose

The Mongoose model created in the previous section provides low level functions such as ***find***, ***create***, ***updateOne***, and ***deleteOne***, that are deliberately vague since they need to be able to operate on any collection. It is good practice to wrap these low level generic functions into higher level functions that are specific to the use cases of the specific projects. For instance instead of just using the generic ***find()*** function, we'd prefer something such as ***findTuits()*** or ***findTuitsPostedByMe()*** or ***findMostPopularTuits()***. The ***data access object*** (***DAO***) ***design pattern*** implements this encapsulation and abstraction principle by grouping data access by data type or collection. The following ***tuits-dao.js*** implements the four basic CRUD operations for the ***tuits*** collection written in terms of the low level Mongoose model operations.

| ***tuits/tuits-dao.js*** |  |
| --- | --- |
| **import *tuitsModel* from './tuits-model.js'**;  **export const** *findTuits* = () => ***tuitsModel***.find();  **export const** *createTuit* = (tuit) => ***tuitsModel***.create(tuit);  **export const** *deleteTuit* = (tid) => ***tuitsModel***.deleteOne({**\_id**: tid});  **export const** *updateTuit* = (tid, tuit) => ***tuitsModel***.updateOne({**\_id**: tid}, {**$set**: tuit}) | |

## 3.4 Creating an API to retrieve data in MongoDB from a React client application

DAOs implement an interface between an application and the low level access of a database, providing a high level API to the rest of the application hiding the details and ideosyncracies of using a particular database vendor. Likewise controllers implement an interface between the HTTP network world and the JavaScript object and function world by converting a stream of bits from a network connection request into a set of objects, maps, and function event handlers that participate in the client/server architecture of a multi tiered application. The Node.js server we've been implementing uses the controllers to talk to the user interface and the DAOs to talk to the database. The server sits between these two layers and therefore it is often referred to as the ***middle tier*** in a multi tiered application.

The current implementation of the ***tuits-controller*** retrieved, updated, and deleted tuits from a ***tuits*** array, but now that we have a database that can store data permanently, let's refactor the controller to store the tuits in the ***tuits*** collection stored in the mongo database. Remove all usage of the array from the ***tuits-controller*** and instead import the ***tuits-dao*** which will provide the functionality of interacting with the ***tuits*** collection. Since the Mongose model interacts with the MongoDB database asynchronously, we'll need to add ***async*** to all the functions in ***tuits-controller***. Refactor the ***findTuits*** function in the controller to retrieve the ***tuits*** using the ***tuits-dao*** as shown below.

| ***tuits-controller.js*** |  |
| --- | --- |
| **import** \* **as** tuitsDao **from '../../tuits/tuits-dao.js'**  **~~import~~** ~~posts~~ **~~from "./tuits.js"~~**~~;~~  **~~let~~** ~~tuits = posts;~~  **const** findTuits = **async** (req, res) => {  **const** tuits = **await** tuitsDao.*findTuits*()  res.json(tuits);  } | *// import the dao*  *// now it's asynchronous function*  *// retrieve tuits from database* |

To refactor ***createTuit*** we won't need to create the ***newTuit***'s primary key ***\_id*** since the database will do that for us when the document is inserted. We also won't be inserting the ***newTuit*** into the array since we're inserting into the ***tuits*** collection instead. Don't forget to add ***async*** to the signatures since we're calling the ***DAO***'s functions asynchronously.

| ***tuits-controller.js*** |  |
| --- | --- |
| **const** createTuit = **async** (req, res) => {  **const** newTuit = req.**body**;  ~~newTuit.~~**~~\_id~~** ~~= (~~**~~new~~ *~~Date~~***~~()).getTime()+~~**~~''~~**~~;~~  newTuit.**likes** = 0;  newTuit.**liked** = **false**;  ~~tuits.push(newTuit);~~  **const** insertedTuit = **await** tuitsDao  .*createTuit*(newTuit);  res.json(insertedTuit);  } | *// now it's an asynchronous function*  *// ID created by database instead*  *// not using array anymore*  *// actual tuit inserted in database*  *// with DAO's createTuit*  *// respond with actual inserted tuit* |

Refactoring ***deleteTuit*** consists of removing any references to the ***tuits*** array and instead using the ***DAO***'s ***deleteTuit*** to remove the tuit from the ***tuits*** collection in the database.

| ***tuits-controller.js*** |  |
| --- | --- |
| **const** deleteTuit = **async** (req, res) => {  **const** tuitdIdToDelete = req.**params**.tid;  **const** status = **await** tuitsDao  .*deleteTuit*(tuitdIdToDelete);  ~~tuits = tuits.filter(t =>~~  ~~t.~~**~~\_id~~** ~~!== tuitdIdToDelete);~~  res.json(status);  } | *// now it's an asynchronous function*  *// status reports success or failure*  *// to delete record from database*  *// no longer using array*  *// respond with status object* |

Finally, refactoring ***updateTuit*** also consists of removing references to the ***tuits*** array since we are updating tuits through the ***DAO*** ***updateTuit*** function.

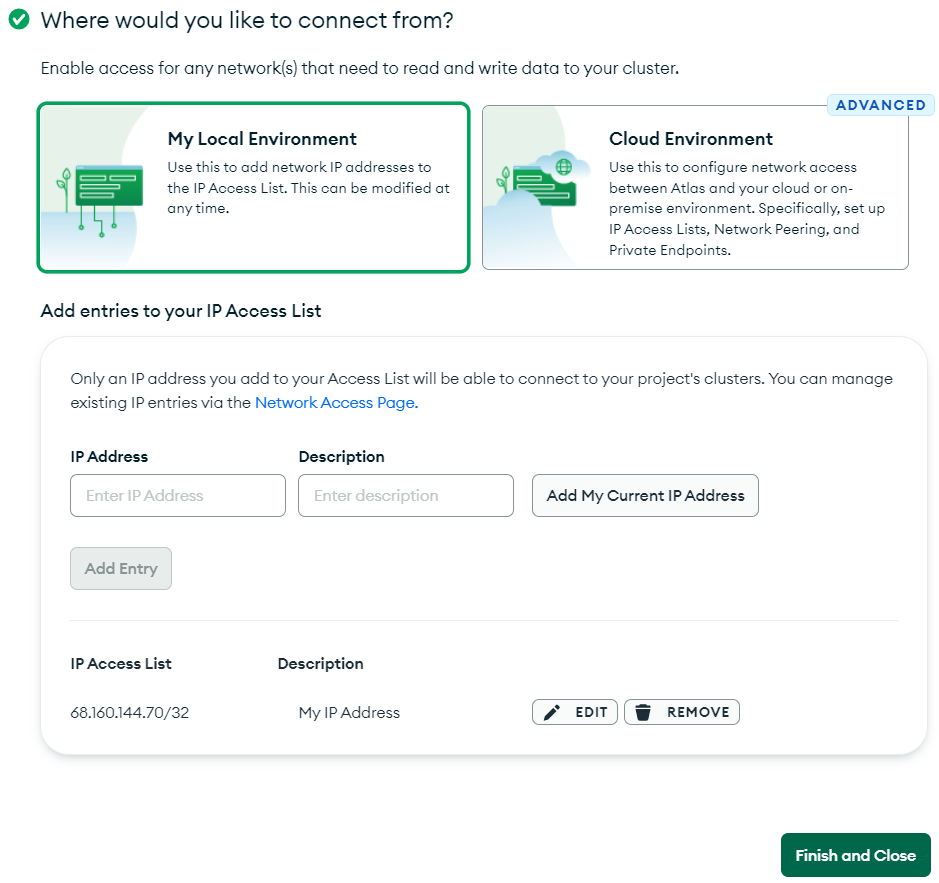
| ***tuits-controller.js*** |  |
| --- | --- |
| **const** updateTuit = **async** (req, res) => {  **const** tuitdIdToUpdate = req.**params**.tid;  **const** updates = req.**body**;  **~~const~~** ~~tuitIndex = tuits.findIndex(~~  ~~(t) => t.~~**~~\_id~~** ~~=== tuitdIdToUpdate)~~  ~~tuits[tuitIndex] =~~  ~~{...tuits[tuitIndex], ...updates};~~  **const** status = **await** tuitsDao  .*updateTuit*(tuitdIdToUpdate,  updates);  res.json(status);  } | *// now it's an asynchronous function*  *// no longer using array*  *// no longer using array*  *// status reports success or failure*  *// to update document in database*  *// respond with status object* |

The React client that interacts with the tuits controller should not need any refactoring since the RESTful API and its behavior is unchanged.

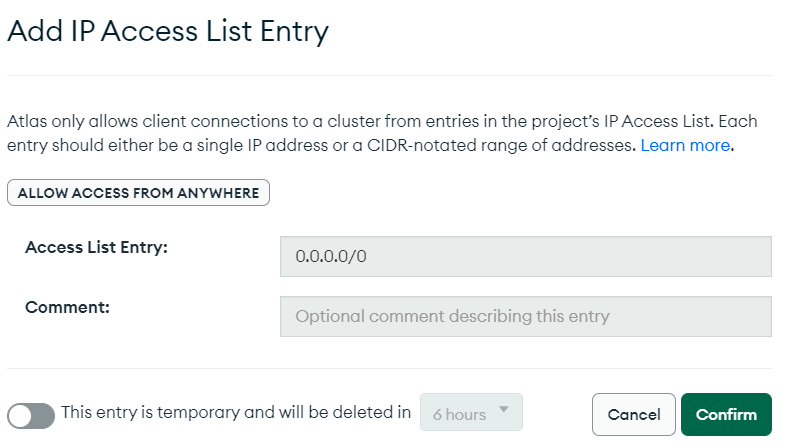
## 3.5 Exercises

1. Install and connect to MongoDB with Mongoose as described in this section
2. Create the Mongoose schema and model as described in this section
3. Create the tuits Data Access Object as described in this section
4. Refactor the tuits controller as described in this section
5. Confirm the React client still works with the API implemented by the tuits controller

# 4 Integrating with MongoDB hosted in the cloud

When you run your server on your development environment, it should be connecting to a MongoDB instance running on the same local development computer. When you deploy the server on a remote server such as Heroku or AWS, the server needs to connect to a database that is also hosted on a public site. MongoDB provides a hosted database service where a MongoDB instances run on public servers, and they provide a connection string to integrate our Node.js application. This section describes setting up and deploying the database online and then integrating with it from our Node.js server running on Heroku.

## 4.1 Setting up MongoDB Atlas

To get started, head over to <https://www.mongodb.com/> and click on ***Sign*** in at the top right corner. Login with your ***Google*** account or click on ***Sign Up*** to create an account with an email and password. If you get a validation email, confirm it and login. Answer any general questions if asked during the signup process. In the ***Deploy a cloud database*** screen choose a ***FREE*** ***Shared*** plan for now which should be enough for this course. In the ***Create a Shared Cluster*** screen, choose any of the cloud providers and region close to where you are, for instance ***AWS*** and ***North Virginia***, and then click ***Create Cluster***. In the ***Security Quickstart*** screen, choose ***Username and Password*** to create credentials to login to your database. In the ***Username*** and ***Password*** fields, type credentials you'll remember later since these are the credentials mongoose will use to login to the database from your Node.js server application when running on Heroku. If you forget these credentials you'll need to create new ones later. I went with ***giuseppi*** and ***supersecretpassword*** :) and clicked ***Create User***. Scroll a bit down and in the ***Where would you like to connect from*** section, select ***My Local Environment*** and then click ***Add My Current IP Address***. This will allow Node.js running on your development environment to connect to MongoDB running on Atlas. Click ***Finish and Close***. Then click on ***Go to Databases***. While in development, it can be convenient to allow connections from anywhere, so click on ***Network Access*** on the left and then ***ADD IP ADDRESS*** on the right. In the ***Add IP Access List Entry*** screen, click ***ALLOW ACCESS FROM ANYWHERE***. This will add 0.0.0.0/0 to the ***Access List Entry***, allowing any machine to connect. Click ***Confirm*** and verify the new entry appears in the ***Network Access*** screen. Click ***Database*** on the left to go back to the ***Database Deployments*** screen. To connect to the database, click ***Connect*** and in the ***Connect to Cluster 0*** screen, select ***Connect your application***. In the ***Select your driver and version*** section, confirm the ***Driver*** is set to ***Node.js*** and ***Version*** is set to ***4.1 or later***. In the ***Add your connection string into your application code*** section, copy the the URL. It should look similar to the following:

| mongodb+srv://giuseppi:<password>@cluster0.eerap.mongodb.net/tuiter?retryWrites=true&w=majority |
| --- |

## 4.2 Integrating Node.js with MongoDB Atlas

Now that we have a connection string, we can configure our Node.js server to use it to connect to the remote database server instead of the local one. In the ***app.js*** file, replace the connection string to use this new string. Better yet, declare an environment variable called ***DB\_CONNECTION\_STRING*** and set it's value to the URL connection string copied from Atlas. Make sure to replace ***<password>*** with the actual password created earlier in the ***Security Quickstart*** screen. Then in ***app.js*** use the code below to use the new environment variable. On macOS you can declare the ***DB\_CONNECTION\_STRING*** environment variable in ***~/.bash\_profile***. It's a bad practice to keep credentials in source code. You might need to restart your machine for the environment variable to take effect.

| ***app.js*** |  |
| --- | --- |
| **const** CONNECTION\_STRING = ***process***.**env**.**DB\_CONNECTION\_STRING**  || **'mongodb://localhost:27017/tuiter**  mongoose.*connect*(CONNECTION\_STRING); |  |

Start the server and the React application and confirm that the application is still working properly locally. When you deploy the Node.js application to ***Heroku***, make sure to create the ***DB\_CONNECTION\_STRING*** environment variable in the ***Heroku Settings*** tab pointing to the same Atlas MongoDB instance. Replace ***<password>*** with the actual password. Restart the React.js application in ***Netlify***, and restart the Node.js server in ***Heroku*** and confirm the two are able to interact properly.

## 4.3 Exercises

| 1. Create an account at MongoDB Atlas 2. Create a database in MongoDB Atlas 3. Refactor ***app.js*** to use the new database | 1. Deploy the Node.js server to Heroku, and configure the server to connect to the remote database. 2. Deploy the React application to Netlify, and confirm it still works with the Node.js server running in Heroku |
| --- | --- |

# 5 Deliverables

As a deliverable, make sure you complete all the exercises in this chapter. For both the React and Node repositories, all your work should be done in a branches called ***a9***. When done, add, commit and push both branches to their respective GitHub repositories. Deploy the new branches to Netlify and Heroku and confirm they integrate. Submit links to both your GitHub repositories as well as the Netlify and Heroku URLs where the branches deployed. Here's an example on the steps:

| ***Create a branch called a9*** |
| --- |
| git checkout -b a9  # do all your work |

Once you've completed all your work, add, commit and push your work to the remote repositories.

| ***Add, commit and push the new branch*** |
| --- |
| git add .  git commit -am "a9 Mongo fa22"  git push |

If you have ***Netlify*** configured to auto deploy, then confirm it auto deployed. If not, then deploy the branch manually.

In Canvas, submit the following

1. The new URLs where your ***a9*** branches deployed to on Netlify and Heroku
2. The link to your new branches in GitHub for the React and Node.js projects