|  |  |
| --- | --- |
| -  Sindy Saintclair  Monday, January 24, 2022  Lesson 1 – Getting Started with NoSQL | |
| **Learning Objectives and Questions** | **Notes and Answers** |
| **OVERVIEW** | Welcome to the world of Database Solutions outside of the scope of the traditional relational database model, known collectively as NoSQL. During this course, you will learn the basics of a particular kind of NoSQL solution known as MongoDB, which has been labeled as “web-scale” in some circles. Databases have always served as a core component of software development. Understanding how to store and retrieve data is essential to any non-trivial application. Throughout this course, I will learn more about the history of databases, the different types of databases, and more. Understanding document databases and NoSQL solutions provide a valuable skill that will be useful in nearly any development environment.  By the end of this course, you will be able to design, implement and modify basic databases, as well as interact to retrieve, store, and modify meaningful data.  Databases are a complex topic with a lot of history, but this foundational course will prepare you with a basic working knowledge of these concepts. |
| **Different Types of NoSQL Databases** | In this course, you will be using the most common form of NoSQL database, a document store database. However, there are other forms of NoSQL databases. They are Key-Value databases, Graph-based databases, Wide-Column stores, and the Document store you’ll be using here.  ***Document Stores***  Document store databases are likely the most common NoSQL databases I will encounter. They contain documents usually in a JSON format but include a unique key maintained by the data store to identify each document. Using the unique key, one document can reference another document similar to a foreign key in a SQL table. Since there is no static structure of the JSON documents, the data store can remain very flexible. The most common Document Store databases are MongoDB and Couchbase. MongoDB is the database you will be using in this course.  ***Key-Value Stores***  A Key-Value store is the simplest and fastest NoSQL database since it only contains an identifier and value for each record, which is similar to the use of variables in programming languages. An identifier is used to determine where a piece of information can be located within the database. This is the key. To return the value, simply use the key which represents the storage of that value. Some of the more common Key-Value store databases are Redis, Riak, and Amazon DynamoDB.  ***Graph Databases***  Graph databases are very similar to document stores with an additional layer of relationship binding between the documents. Instead, these documents are called Verticies, Nodes , or Points while the relationships between these vertices are called Edges. Graph databases are designed so that vertices are based around nouns such as persons, places, and objects. Using edges to declare the relationships between vertices rather than a key/foreign-key relationship separates the connections from the data itself. This allows for a change of relationships or data to not affect each other. Some of the more popular Graph Databases are Neo4J.  ***Wide Column Stores***  Wide columns stores are perhaps the hardest to understand. The simplest way to think of them is like a SQL database table, but each record has its own column name and, the key is required. Another way of imagining it is like a Document store where a predefined key is associated with the document, and the document cannot have any sub-documents. The complexity of Wide Column stores tends to shy people away from using them, but large amounts of data can be stored this way conveniently without the need for a specific structured table like a SQL database has. They also tend to be incredibly fast because of the one-for-one relationship between the key and the data. The most popular Wide Column store database is Cassandra. |
| **Setup** | To continue with this lesson, you will need to set up and configure the required software on your computer. The instructions are below.  Setup Instructions  For this course, you will be using MongoDB, a popular NoSQL implementation. Your MongoDB database will be provided by Atlas, a cloud-hosted database service specializing in MongoDB.  Before you start, you will first need to sign up for a free MongoDB Atlas account at [www.mongodb.com](http://www.mongodb.com)  Mongo DB website. The main page of the Mongo DB website. It says Mongo DB atlas. Deploy a fully managed cloud database in minutes. There is an arrow in the upper right corner pointing to a button that says Try Free.  Figure 1-1: MongoDB Website  Click on the Try Free button on the MongoDB.com website, located in the top right corner of the page. This will take you to a signup screen that looks like the one below:  Mongo DB signup. Signup page for Mongo DB. There are text boxes to fill in with a username, first and last name, and password. An arrow points to the button at the bottom of the page that reads "Get started free".  Figure 1-2: MongoDB Signup  Once you have signed up, you will be directed to the MongoDB Atlas website to create your first database cluster.   * A database cluster refers to several database servers hosting a database instance. this creates redundancy since more than one server can be connected to for the same data.   Click on the button that says Build my first cluster to begin creating your first database.  Build First Cluster. Splash page for Mongo DB Atlas. It says Welcome. Mongo DB Atlas is the best way to run Mongo DB. You won't have to worry about installing, configuring, or managing your database anymore. Get started by building your first cluster. An arrow is pointing to a button that reads Build my first cluster.  Figure 1-3: Build First Cluster  Make sure to click (FREE) Shared cluster.  Cluster (FREE) Shared.  Figure NEW: Free Shared Cluster  Next, you’ll be prompted to select a region and provider to host your MongoDB instance. The default selections will have a green box around them. Make sure the selection for your region is a FREE TIER AVAILABLE selection. It’s recommended to use AQS and N. Virginia. Then click on the Create Cluster button at the bottom of the page.  Cluster select. Icons are shown for several different cloud providers and regions. Arrows point to A W S and N. Virginia in North America. Another arrow is pointing to the create cluster button at the bottom of the screen.  Figure 1-4: Cluster Selection  Now you have your first cluster created. Great job! In a later lesson, you will se up your first database using MongoDB Atlas. |
| **What is NoSQL?** | The term **NoSQL** refers to a data storage system that does not have the relational constraints and model that a traditional SQL RDBMS (relational database management system) has. NoSQL stands for “NotSQL,” “Non-SQL,” or more recently, “Not Only SQL.” It is category of storage solutions, of which there are various types. The last term, “Not Only SQL,” actually refers to a storage system that is not SQL, but it is similar to SQL when interacting with your data. The main difference between SQL and NoSQL is the presence of a predefined schema. With a SQL database, you create tables that contain only certain information and act as a blueprint for organizing your data. With NoSQL, the NoSQL databases are built to allow data insertion without a predefined schema or blueprint in SQL.  There are four categories of storage solutions for most of the systems that qualify as NoSQL. As new ones are continually coming into existence, there is some cross-pollination of ideas and functionality, so these categories are by no means absolute. However, at this time, most of the solutions can fit into these four types: **Document Database**, **Key-Value Database**, **Graph Database**, and **Object Database**.   1. **Document Database:** This is the primary version of NoSQL that you will be interested in, as MongoDB is a document-oriented database. The key concept for a document database is that data for a particular purpose of entity are grouped as a single document and stored as such. Each document in your database will have a unique identifier, which you will use to complete your CRUD operations. The most popular type of structures for documents in these databases are BSON, XML, or JSON. 2. **Key-Value Database:** Probably the most aptly named version of a NoSQL solution, a Key-Value database is simply that – a single, unique key for every set of data that you store in the database. It is sometimes referred to as a dictionary – as it work in the same fashion. In a dictionary, the word itself that you look up would be the key, and the value associated with that key would be the word’ definition. 3. **Graph Database:** This is a term you should be familiar with, as with the explosion of social networking, they have gained popularity quite a bit. Under the hood of a Graph Database, you will usually find the storage to be a Key-Value or Document Database. Still, they are structured with nodes and edges properties to represent relationships instead of containing the relationships in the data like an RDBMS does. 4. **Object Database:** An Object-Oriented database, sometimes abbreviated OODBMS, is a database where the data itself is stored in the form of objects. An object will contain all information about a specific record in the database. Think of if like a car – it has weight, color, wheels, windows, and it stops and starts. All of this information is contained in one place: an object. An object also allows for different types of data.   Now that you have an idea of what NoSQL is, it is time to dive into creating a database. |
| **Creating a Database** | Previously in this lesson, you signed up for MongoDB Atlas, which is where you will be storing all of your data in your database. Next, we need to make sure you can access your database.   1. Click on Database Access:   Database access configuration. Window showing the cluster menu. The database access option is selected. On the right side of the screen, it shows part of a sandbox with the cluster tier, region, type, and stitched app.  Figure 1-6: Database Access Configuration   1. Click on add new database user:   Add new database user. Screen in the application that says create a new database user. Set up database users, permissions, and authentication credentials to connect to your clusters. There is a large button that says Add new Database user that highlighted.  Figure 1-7: Add New Database User  The Add New User screen is where you can add user accounts and set their permsisions for accessing your databases. Enter the name of dbuser and set the password to Password1!. Then click on the Add User button on the bottom right of the screen.  Add DB User. A screen that shows SCRAM authentication with two textboxes to enter username and password. User privileges icons and menus are across the bottom of the screen. They include Atlas Admin, Read and Write to any database, which is highlighted, Only read any database and select custom role. An arrow is pointing to the add user button at the bottom right of the screen.  Figure 1-8: Add dbuser  You should now see the new dbuser account listed under the Security section of your cluster.   1. Next on your list is to Whitelist your IP address. click on Network Access:   Network Access.  Folder directory with the network access option highlighted. The username DB user and the authentication method SCRAM are shown.  Figure 1-9: Network Access   1. Then click on Add IP Address:   Add an IP Address. Screen that reads Whitelist an IP Address. Configure which IP Addresses can access your cluster. There is a large button that says Add IP Address that is highlighted.  Figure 1-10: Add IP address   1. From the Add Whitelist Entry screen, you will see two buttons. One button will state ADD CURRENT IP ADDRESS and the other will state ALLOW ACCESS FROM ANYWHERE. Click on the ALLOW ACCESS FROM ANYWHERE button. This will add all the IP addresses to the whitelist and prevent any issues connecting to your database in the future. This will set the Whitelist Entry field to a value of 0.0.0.0/0. Then click on the Confirm button.   Add access from anywhere. An arrow points to a button that says add access from anywhere on a screen titled Add Whitelist Entry. Another arrow points to a button in the bottom right corner that says confirm.  Figure 1-11: Add Access from Anywhere  The final step in the Get Started checklist is to connect your database. To do that, you will need to install the MongoDB extension for VS Code. You’ll cover that in the next topic. |
| **Connecting to MongoDB with VS Code** | Before you connect to your MongoDB database, you will need to install the official MongoDB for VS Code extension and a connection string from the MongoDB website. To connect and install the MongoDB for VS Code, follow the steps below:  ***Connection String -*** *mongodb+srv://dbuser:<password>@cluster0.zo6yg.mongodb.net/test*   1. Make sure you are on the tab labeled Databases on the left. You can then click the Connect button under your Database Deployments information.   Connect button.  Figure NEW: Connect Button   1. After you click on the Connect button, you will be presented with three options of how to connect to your MongoDB database cluster. The third option is Connect using MongoDB Compass, which is the option you want to click.   Connect Options.  FIGURE NEW: Connect Options  **Connect with MongoDB Shell** will walk you through connecting via the Mongo Shell. Some developers prefer this interface.  **Connect Your Application** will walk you through connecting a Java, Python, Ruby, or NodeJS application to the database cluster.   1. The next screen will show your connection string. You do not need to install MongoDB Compass as we will be using the VS Code extension. Copy this connection string and save it somewhere as you will need it in the next section.   Connection String.  Figure NEW: Connection String  ***Installing MongoDB for VS Code***   1. Open the Extensions view and search for MongoDB to filter the results. Select the MongoDB for VS Code extensions.   MongoDB for VS Code  Figure NEW: MongoDB for VS Code   1. Once you’ve installed the MongoDB for VS Code extension, you’ll notice there is a new MongoDB Activity Bar view. Select the MongoDB view, and you’l see the MongoDB Explorer.   MongoDB for VS Code  Figure NEW: MongoDB for VS Code   1. To connect to a MongoDB database, select Add Connection and then click the connect with a connection string link. An input area will pop up – paste the connection string we copied in the Connection String section (you may need to right-click and choose Paste).   MongoDB for VS Code  Figure NEW: MongoDB for VS Code   1. After pasting your connection string, you will need to change the section that says <password> to the password you assigned to dbuser, which should be Password1! – Once changed, press Enter to connect.   MongoDB for VS Code  Figure NEW: MongoDB for VS Code   1. VS Code should now be connected to MongoDB!   MongoDB for VS Code  Figure NEW: MongoDB for VS Code |
| **MongoDB – Playgrounds** | MongoDB Playgrounds  **MongoDB Playgrounds** is included with the VS Code extension. Playgrounds allow you to quickly test out MongoDB commands. It is a great way to learn how to use MongoDB. It is also a great way to learn how to use VS Code.  MongoDB Playgrounds provide intelligent autocomplete and commands with helpful syntax highlighting.   * MongdoDB for VS code splits your Playground window to display your Playground Result in the right-side pane.   ***Your First MongoDB Playground***   1. In the Playgrounds panel of MongoDB for VS Code, click Create New Playground.   MongoDB Playgrounds  Figure NEW: MongoDB Playgrounds   1. Due to us already setting up an active connection, your playground should already be connected. If you do NOT have an active connection, you will see Click here to add connection at the top of your playground. Click on the Add Connection button and enter the connection string we copied in the Connection String section.   MongoDB Playgrounds  Figure NEW: MongoDB Playgrounds   1. The playground will show a template with example commands. You will not be using the commands in this template, but feel free to take a look and see how they work. You can delete everything in the template to start from scratch. You are now ready to write your own commands in the following section.   MongoDB Playgrounds  Figure NEW: MongoDB Playgrounds |
| **MongoDB Collections** | **MongoDB Collections** are a grouping of MongoDB documents and are the equivalent of an RDBMS table. A collection exists within a single database and does not enforce a schema (or a blueprint of your data). Documents within a collection can have different fields. Typically, all documents in a collection have a similar or relate purpose. You can refer to this as a *flexible schema*.  Within collections are documents, which are independent units that make performance better (related data is read contiguously off the desk) and make it easier to distribute data across multiple servers while preserving its locality. Each document has a document ID, which is the unique identifier for a document. When working with Collections and Documents, you will perform CRUD (Create, Read, Update, Delete) operations again. |
| **Creating Your First Collection** | You will start with a very important acronym in the data storage universe – **CRUD**. CRUD is an acronym for Create, Retrieve ,Update and Delete. This refers to the category of operations in an RDBMS known as DML (data manipulation language). Also, in MongoDB, you can use a set of operations to do the same types of things wit the documents you will create and retrieve.  But before you can get into the CRUD operations within MongoDB, you will first need to build the construct (the collection) inside the database in which you store your data. For this example, you will use your connection to the database through the Mongo Playground to create your first collection.  A function is a predefined set of code that can call to be executed by the user. In this case, you want to execute a function called createCollection. To call this function, you use a set of open and closed parentheses (). But there’s one additional thing you need to know before you can execute the function. In order for this function to work, it needs to know the name of the collection to create. You pass this to the function through something called an argument, and the argument is inserted between the parentheses. If you have done any programming before, this will look very familiar. If you haven’t, copy the test below into your terminal and press enter to execute the function:  db.createCollection("appusers");  The above command executes the createCollection function on the database (DB is short for database). You are passing the argument of appusers as the name you want to give this collection.  To execute the function, you need to highlight the function and click the Run Selected Lines From Playground message or press the play button on the top right.  MongoDB Playgrounds  Figure NEW: MongoDB Playgrounds  After executing this function, you will get a response back from the server with some information about the result of your request.  MongoDB Playgrounds  Figure NEW: MongoDB Playgrounds  The only significant thing to note at this time is that the "ok"value is set to 1, indicating that the request was processed successfully and your new collection called appusers is created.  Now you can verify that this collection has been created on the Atlas website.  Open your browser, navigate to <https://cloud.mongodb.com>, and log back in if you aren’t logged in already.  From your Overview tab in Atlas, click on the Cluster0 active hyperlink.  Cluster Hyperlink. The clusters sandbox. An arrow is pointing to cluster 0.  Figure 1-37: Cluster Hyperlink  In your Cluster0 screen, you will find a Collections tab, click on Collections.  Collections tab. An arrow is pointing to the Collections tab in the clusters sandbox 0 windows.  Figure 1-28: Collections Tab  You should see the appusers collection listed in the collections of your database.  Collections list. The collection test app users are highlighted inside the collections tab.  Figure 1-39: Collections List  Great work! Now you have a place to store your documents in your database. Continue, and you’ll explore how to add data to these documents. |
| **Data Types** | Before you dive into inserting documents into your collection, you need to know what types of data are allowed in a NoSQL database. When creating a document, the data within it need to live somewhere. When working with MongoDB, the data lives within a JSON (pronounced jay-son) file.  ***JSON***  Below is the definition of JSON from the MongoDB website:  JavaScript Object Notation (JSON) is a lightweight data-interchange format that is easy for humans to read and write and for machines to parse and generate .Along with XML, JSON is the main format for data interchange used on the modern web. JSON supports all basic data types: numbers, strings, Booleans, in addition to arrays and hashes.  MongoDB and other document databases use JSON documents to store records, while relational databases use tables and rows to store records. Below you will find an example of a JSON document:  {  "\_id": 1,  "name": { "first": "John", "last": "Backus" },  "contribs": ["Fortran", "ALGOL", "Backus-Naur Form", "FP"],  "awards": [  {  "award": "W.W. McDowell Award",  "year": 1967,  "by": "IEEE Computer Society"  },  {  "award": "Draper Prize",  "year": 1993,  "by": "National Academy of Engineering"  }  ]  }  JSON databases return query results that can parse easily with little to no transformation. JavaScript and many other popular programming languages can parse these results. This ease of parsing reduces the amount of logic needed to build into the application layer.  ***JSON Types***  Below is a list of data types allowed within JSON:   * sring * number * object (JSON object) * Array * Boolean * Null   JSON values cannot be the following data types:   * Function * Date * undefined * Although Date data types can’t be used in JSON, their string representations can be, and are how dates are passed around in JSON. For example, “2018-02-01” (January 2, 2018) is acceptable in JSON and can be converted to a Date object.   Now it’s time for you to explore how JSON values will look.  **JSON String** – strings in json MUST BE written within quotes:  { "name": "Emily" }  **JSON Number** – Numbers in JSON MUST BE an integer and will NOT be wrapped in quotes:  { "age": 30 }  **JSON Object** – JSON values can be objects. Remember that objects are like physical objects, like a car: many components make up the entire object.  {  "customer": {  "firstName": "Carl",  "lastName": "Orff",  "age": 75,  "location": "Munich"  }  }  **JSON Array** – JSON values can be arrays. Arrays are used to store multiple values within a single variable.  {  "customers": ["Anna", "Mary", "Thomas", "Robert", "Tom", "Cora"]  }  **JSON Boolean** – JSON values can be Boolean. Booleans are values that are either set to be true or false.  { "sale": true }  **JSON Null** – JSON values can also be Null. Null is a value that does not exist but prevents the field from being empty.  { "middlename": null } |
| **Relationships** | Data within MongoDB has something called a *flexible schema*. This term means that collections within a database do not enforce a specific data structure. How the data is modeled in the collection can affect the application’s performance and database capacity. These are referred to as embedded documents and they are the best and most common way of structuring data in MongoDB.  ***One-to-One Relationship***  Say you have customers in a collection, and you want to retrieve the address of a specific customer. In a normalized data model, it would look something like below:  {  "\_id": "123",  "name": "Jeremy Houser"  }  {  "patron\_id": "123",  "street": "1012 Grove Street",  "city": "Faketown",  "state": "CA",  "zip": "12345"  }  Above, you have two different sets of data that can be linked together using the ID of “123”. In NoSQL, the better way of organizing the data is to embed the address information within the same dataset of the customer’s name, like below:  {  "\_id": "123",  "name": "Jeremy Houser",  "address": {  "street": "1012 Grove Street",  "city": "Faketown",  "state": "CA",  "zip": "12345"  }  }  Now the address information of the customer is embedded and located in the same area as the name. Now you do not need to link two different sets of data together. This allows you to retrieve all of the data of this customer with only *one* query.  ***One-to-Many Relationship***  Say your above customer has multiple addresses. In a normalized data model, you would link the multiple addresses to the customer by using the customer’s \_id:    "\_id": "123",  "name": "Jeremy Houser"  }  {  "patron\_id": "123",  "street": "1012 Grove Street",  "city": "Faketown",  "state": "CA",  "zip": "12345"  }  {  "patron\_id": "123",  "street": "8877 E. 1st Street",  "city": "Motown",  "state": "NY",  "zip": "09887"  }  Instead of the above, you can embed the multiple addresses as you did above. The difference when embedding multiple documents within one field is that they need to be contained within square brackets []:    "\_id": "123",  "name": "Jeremy Houser"  }  {  "patron\_id": "123",  "street": "1012 Grove Street",  "city": "Faketown",  "state": "CA",  "zip": "12345"  }  {  "patron\_id": "123",  "street": "8877 E. 1st Street",  "city": "Motown",  "state": "NY",  "zip": "09887"  }  Do you see how the addresses are within the square brackets []? This data structure will allow you to pull in all of a specific customer’s data with *one* query.  ***Many-to-Many Relationship***  There will come a time when you are working with a database and referencing documents will be more efficient than embedding documents. This typically happens when working with a Many-to-Many Relationship. Consider the following example:  {  "\_id": "123",  "name": "Jeremy Houser",  "address": {  "street": "1012 Grove Street",  "city": "Faketown",  "state": "CA",  "zip": "12345"  },  "supportRep": [  {  "\_id": 2435,  "title": "Support Manager",  "location": "MA"  },  {  "\_id": 3829,  "title": "Support Representative",  "location": "NY"  }  ]  }  {  "\_id": "443",  "name": "Alison Markson",  "address": {  "street": "664 Central Street",  "city": "Blurg",  "state": "MO",  "zip": "87469"  },  "supportRep": [  {  "\_id": 2435,  "title": "Support Manager",  "location": "MA"  },  {  "\_id": 3829,  "title": "Support Representative",  "location": "NY"  }  ]  }  Above, you have *two* different customers with the same *two* support representatives. In each case, you include all of the information of the same support representatives within each customer. This is allowing for repetition, which you do not want. To prevent that, you can have a collection that stores the data of all the support representatives, along with your collection of customers. You can then reference the support representatives’ collection:  // SupportRep collection  [  {  "\_id": 2435,  "title": "Support Manager",  "location": "MA"  },  {  "\_id": 3829,  "title": "Support Representative",  "location": "NY"  }  ]  // Customer collection  {  "\_id": "123",  "name": "Jeremy Houser",  "address": {  "street": "1012 Grove Street",  "city": "Faketown",  "state": "CA",  "zip": "12345"  },  "supportRep\_id": [2435, 3829];  }  {  "\_id": "443",  "name": "Alison Markson",  "address": {  "street": "664 Central Street",  "city": "Blurg",  "state": "MO",  "zip": "87469"  },  "supportRep\_id": [2435, 3829];  }  You store the support representative’s \_id within each customer so that both supportReps are easily found and referenced.  Great! Now that you understand the data structure in MongoDB, it is time to explore how to insert documents into your database. |
| **Inserting Documents** | Earlier in this lesson, you learned what documents look like within a MongoDB collection. Each document has a field and a value. That value can then be an embedded document if needed.  ***Insert A Single Document***  Before inserting documents, make sure you have a playground up, running, and connected to your database.  To insert a single document into a collection, you would use insertOne(). The generic syntax for inserting is as follows:  db.collectionName.insertOne([document\_schema])  The [document\_schema] is the document itself and includes the fields and values separated by commas (,) within curly braces ({}). When inserting documents, you must make sure that each document has a unique identifier, usually an appusers. MongoDB will automatically create one for you if you insert a document without an appusers field. If you would not like to do that, you can identify the document ID yourself. You MUST make sure that the documentID does not already exist, or you will have an error thrown when you try to run the insert. since your appusers collection is empty, you should add a new user to your collection, as shown below:  db.appusers.insertOne({\_id:1,lastName:"Pumpernickel", firstName:"Georgina", middleName:"Sasquatch", age:27, gender:"f"})  Go ahead and run the above query in the playground. Other than the fact that poor Georgina has an unfortunate name, you should have also noticed the addition of the \_id field, and you’ve assigned it a value of 1. This will succeed, and MongoDB will return an "acknowledged" : true message , with an "insertedId" : 1along with it. The critical part here is to make sure that you don’t duplicate when you specify the ID yourself, or your insert will fail. If you want to be sure that your data insert will succeed, you can let MongoDB generate that unique value for you and use the value that it provides in your code moving forward.  Now add another user, leveraging the flexible schema you learned about earlier:  db.appusers.insertOne({\_id:2,lastName:"Wilson", firstName:"Coderboy", age:18, favoriteColor:"blue"})  You should run this command and get the same acknowledgment message as before, except this time you will see that the InsertedId is 2. Notice the differences: Georgina has a middle name and gender specified, while Coderboy does not – yet Coderboy has a favorite color specified. Ideally, you’d have an initial set of fields that you would deem acceptable as entries into your “persons” documents, and a subset of those are what you would be populating. This will be important later as you create indexes in your MongoDB database.  Now, if you want to see all documents within your database, you can run a query to see both of your documents. See below:  db.appusers.find({})  The above query will find all documents within the appusers collection. You should now see both documents you have inserted and all of the fields included.  ***Inserting Many Documents***  The .insertMany() query is the companion to insertOne, where you can insert several documents into the same collection at once. The syntax is somewhat similar:  db.collectionName.insertMany( [ {document1}, {document2}, … {documentN}])  The specification of having an \_id field or not will work exactly the same as the insertOne command. You can specify one if you want, but make sure that it’s not already in use. Now add a few more rows to your original appusers collection, and see how this works:  db.appusers.insertMany([{lastName: "Anstruther", firstName: "Jimmy", age: 21}, {lastName: "Stevens", firstName: "Amelia", age: 28},{lastName: "Hopkins", firstName: "Fred", age: 35, favoriteColor: "Green"}])  Notice how the last document has an extra field “favoriteColor” that the first two documents in the query do not have. Also, you did not include id’s for the three new users like you did with the first two users added. Because of the flexible schema, you can do this. Go ahead and run the above query. You should get the following message:  {  "acknowledged": true,  "insertedIds": {  "0": {  "$oid": "6102f0fc086d5ec1c2162cd6"  },  "1": {  "$oid": "6102f0fc086d5ec1c2162cd7"  },  "2": {  "$oid": "6102f0fc086d5ec1c2162cd8"  }  }  }  Again, run the find query to see all of your documents:  db.appusers.find({})  Look carefully at the returned documents. The \_id field that you created will now have the $oid value that was automatically created when the documents were inserted. You should see something similar to the code below, but your $oid values may be different:  {  "\_id": {  "$oid": "6102f0fc086d5ec1c2162cd6"  },  "lastName": "Anstruther",  "firstName": "Jimmy",  "age": 21  },  {  "\_id": {  "$oid": "6102f0fc086d5ec1c2162cd7"  },  "lastName": "Stevens",  "firstName": "Amelia",  "age": 28  },  {  "\_id": {  "$oid": "6102f0fc086d5ec1c2162cd8"  },  "lastName": "Hopkins",  "firstName": "Fred",  "age": 35,  "favoriteColor": "Green"  }  Now that you have gone through plenty of inserting, return to your Collections tab in Atlas. Refresh the page, and you should use that your appusers collection now has five documents. You can click on the collection and see each document and the fields and values included.  ***Create Collection When Inserting***  You’ve seen how to use the db.createCollection function to create a collection, but there is a shortcut. If you run an insert query with a collection name that does not already exist, it will automatically create that collection for you. Time to try it!    Run the following query:  db.movies.insertMany([{title: "The Shawshank Redemption", director: "Frank Darabont", yearReleased: 1994}, {title: "The Dark Knight", director: "Christopher Nolan", yearReleased: 2008}, {title: "It's a Wonderful Life", director: "Frank Capra", yearReleased: 1946}])  You should get the "acknowledged" : true message. If you go into Atlas again, you will see a collection called movies with three documents.  Great work during this lesson! Time for you to do a lesson review of the key terms and then jump into your first MongoDB project. |