**1.What is MongoDB and why we use this and explain the setup steps of MongoDB?**

MongoDB is an open-source document-oriented database that is designed to store a large scale of data and also allows you to work with that data very efficiently. It is categorized under the NoSQL (Not only SQL) database because the storage and retrieval of data in the MongoDB are not in the form of tables.

MongoDB is a document database built on a scale-out architecture that has become popular with developers of all kinds who are building scalable applications using agile methodologies.

MongoDB was built for people who are building internet and business applications who need to evolve quickly and scale elegantly. If you are doing that, you should consider MongoDB.

Companies and development teams of all sizes use MongoDB because:

* The document data model is a powerful way to store and retrieve data that allows developers to move fast.
* MongoDB’s horizontal, scale-out architecture can support huge volumes of both data and traffic.
* MongoDB has a great user experience for developers who can install MongoDB and start writing code immediately.
* MongoDB can be used everywhere by anyone:
  + *For free through the open source community edition*
  + *In the largest data centers through the enterprise edition*
  + *In any of the major public clouds through MongoDB Atlas*
* MongoDB has developed a large and mature platform ecosystem, which means:
  + *MongoDB has a worldwide community of developers and consultants, so it is easy to get help.*
  + *MongoDB works on all types of computing platforms, both on-premise and in the cloud (both private, and public clouds such as*[*AWS*](https://www.mongodb.com/mongodb-on-aws)*,*[*Azure*](https://www.mongodb.com/mongodb-on-azure)*, and*[*Google Cloud*](https://www.mongodb.com/cloud/atlas/mongodb-google-cloud)*)*
  + *MongoDB can be used from all major languages.*
  + *MongoDB can be accessed from all major ETL and*[*data management systems*](https://www.mongodb.com/database-management-system)*.*
  + *MongoDB has enterprise-grade support.*

Why use MongoDB? Simply to go further and faster when developing software applications that have to handle data of all sorts in a scalable way.

Thousands of companies like Bosch, Barclays, and Morgan Stanley run their businesses on MongoDB, and use it to handle their most demanding apps in areas like IoT, Gaming, Logistics, Banking, e-Commerce, and Content Management.

MongoDB is a great choice if you need to:

* Represent data with natural clusters and variability over time or in its structure
* Support rapid iterative development.
* Enable collaboration of a large number of teams
* Scale to high levels of read and write traffic.
* Scale your data repository to a massive size.
* Evolve the type of deployment as the business changes.
* Store, manage, and search data with text, geospatial, or time series dimensions.

MongoDB as a company has grown because the number of use cases with these characteristics keep growing.

Step 1: installation

You can find the free open source edition ‘Community Server’ as well as the commercial enterprise solution available on the download center of the official MongoDB website. The first step is to find the right installation file (binary) for your system and to download it. Since MongoDB is available across a selection of platforms, there are a number of different variants for Windows and Linux as well as for OS X and Solaris.

If you’re using a Windows operating system, the database is simply installed into your folder of choice with help from the downloaded file. Windows 10 users can use the Windows Server 2008 (64-bit) version. Users of Linux and co. will have to download an archive file, unzip it and then install it with the help of the package manager. Depending on your distribution, you may first need to import the MongoDB Public GPG Key. For Ubuntu, this authentication key is required, which means you have to implement the following command:

sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927

Then you should update the list of the package manager:

sudo apt-get update

… and install MongoDB with useful management tools included:

sudo apt-get install -y mongodb-org

Step 2: starting the MongoDB server

You have the option to change the standard installation folder /var/lib/mongodb and the log folder /var/log/mongodb in the configuration file/etc/mongod.conf. The following command will start the database:

sudo service mongod start

If you use the parameter stop instead of the start command, the database will be shut down; the restart command can be used to reboot the database. To test to see whether MongoDB was successfully started, just look up the log file /log/mongodb/mongod.log: The line

[initandlisten] waiting for connections on port <port>

tells you that the database server is running and waiting for incoming connections to the port defined by the configuration file (<port>). The default is port 27017.

Step 3: starting the client

Once the MongoDB server is running, you can start the respective client. We’re using the included command line client Mongo Shell, which is based on JavaScript and may be used for administration of the database as well as accessing and updating the data. You can start the client with a simple terminal command on the same system that the MongoDB application is running:

mongo

The Mongo Shell will automatically connect to the MongoDB server that’s already running on the local host and the port 27017. You can of course always customize these default connection settings with the appropriate parameters as well. These parameters and other important options are listed in the following table:

Parameter

Function description

--shell

Activates the Shell Interface, where you can use the interactive mongo shell after executing a command

--nodb

Disables the connection between the Mongo Shell and a database

--port <port>

Defines the port for the connection

--host <hostname>

Defines the host for a connection

--help or -h

Shows you all possible options

--username <username> or -u <username>

If access rights are defined, you can use this command to log in with your corresponding user name (<username>)

--password <password> or -p <password>

If access rights are defined, you can use this command to enter your corresponding password (<password>)

The brackets included here are not part of the respective parameter and so shouldn’t appear in the final command. For example, when defining port 40000 instead of the standard port 27017 your entry should look like this:

mongo --port 40000

Step 4: creating a database

As soon as MongoDB and the client are running, you can dedicate your time to data management and editing. But first, you should create a database – otherwise your collections and documents will be saved to the automatically generated test database. You can generate a database through the use command. So if you want to create a database with the name mydatabase, for example, then the command will look like this:

use mydatabase

The command use also selects an existing MongoDB database that you want to call up for data processing; by using the short command db you can check which database is currently selected.

Step 5: creating a collection

Your next step is to create your first collection, a binder for various BSON documents in which you will store the data later. The basic syntax follows this pattern:

db.createCollection(<name>, { options } )

The createCollection command contains two parameters: name (title of the collection) and options (optional options to configure in the collection). In the options, you can determine whether the collection should be limited in size (capped: true), and if so, to what number of bytes (size: <number>) or additionally to what number of documents (max: <number>), for example.

A collection with the name mycollection, a byte limit of 6,142,800 and a maximum of 10,000 documents permitted could be created using the following command (the whitespace is simply for clarity):

db.createCollection ("mycollection", { capped: true,

size: 6142800,

max: 10000 } )

Step 6: add documents to a collection

After the binder has been created you can populate your collection with documents. There are three different means of doing this:

.insertOne()

.insertMany()

.insert()

The commands above allow you to insert either one document (.insertOne), several documents at once (.insertMany), or simply to insert as many as you list (.insert). The following example shows a simple database entry, containing three pieces of information: name, age, and sex. This document will be stored in the mycollection binder that we created in step 5:

db.mycollection.insertOne(

{

Name: "Name",

Age: 28,

Sex: "female"

}

)

MongoDB automatically generates a unique ID for this entry and every subsequent entry in this corresponding collection.

Step 7: manage documents

The final step of our MongoDB tutorial concerns the basic management of these created documents. Before you can make changes to a document, you have to call it up first. This action can be performed with the find command and can be expanded with the optional parameters query filter and projection (specification of the displayed result). So to call up the document from the previous step, we’d need the following command:

db.mycollection.find( { Name: "Name", Age: 28 } )

If you want to update this document, you’ll need to perform the update function. Here, you have to define the value you wish to change, choose an update operator, and then enter the new value. So if you want to change the age field in our example, you need the operator $set:

db.mycollection.update(

{ Age: 28 },

{

$set: { Age: 30 }

}

)

The other update operators can be found here.

To delete a document from a collection, you’ll need to use the remove command:

db.mycollection.remove()

You can also remove individual documents from the collection by defining criteria like the ID or exact values, thus signalizing which database entries are in question to MongoDB. The more specific you are, the more exact the removal process by the database system can be. The command

db.mycollection.remove( { Age: 28 } )

will remove all entries for the value 28 in the Age field. But you can also specify further, telling the database to only remove the first entry that meets this description. This involves using what’s known as a justOne parameter (1):

db.mycollection.remove( { Age: 28 }, 1 )

Further information on user administration, security settings, creation of replicas, or the sharing of data across several systems can be found in the official documentation on the mongodb.com website as well as through the more detailed MongoDB tutorial on tutorialspoint.com.

**2.What is data modeling and explain briefly different types of data modeling in MongoDB?**

Data modeling is the process of creating a visual representation of either a whole information system or parts of it to communicate connections between data points and structures. The goal is to illustrate the types of data used and stored within the system, the relationships among these data types, the ways the data can be grouped and organized and its formats and attributes.

**Data Model Design**

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

**Embedded Data Model**

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal\_details, Contact and, Address, you can embed all the three documents in a single one as shown below −

{

\_id: ,

Emp\_ID: "10025AE336"

Personal\_details:{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26"

},

Contact: {

e-mail: "radhika\_sharma.123@gmail.com",

phone: "9848022338"

},

Address: {

city: "Hyderabad",

Area: "Madapur",

State: "Telangana"

}

}

**Normalized Data Model**

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

{

\_id: <ObjectId101>,

Emp\_ID: "10025AE336"

}

Personal\_details:

{

\_id: <ObjectId102>,

empDocID: " ObjectId101",

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26"

}

Contact:

{

\_id: <ObjectId103>,

empDocID: " ObjectId101",

e-mail: "radhika\_sharma.123@gmail.com",

phone: "9848022338"

}

Address:

{

\_id: <ObjectId104>,

empDocID: " ObjectId101",

city: "Hyderabad",

Area: "Madapur",

State: "Telangana"

}

**3.What is aggregation in MongoDB explain aggregation pipeline, map-reduce function,single purpose aggregation?**

Aggregation can be defined as the operation that is used for processing various types of data in the collection, which returns a calculated result. The concept of aggregation mainly clusters out your data from multiple different documents which are then used and operates in lots of ways (on these clustered data) to return a combined result which can bring new information to the existing database. You can relate aggregation to that of the count(\*) along with the 'group by' used in SQL since both are equivalent in terms of the working.

**Aggregation Pipeline**

MongoDB's aggregation framework is modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result. For example:

db.orders.aggregate([

{ $match: { status: "A" } },

{ $group: { \_id: "$cust\_id", total: { $sum: "$amount" } } }

])

First Stage: The $match stage filters the documents by the status field and passes to the next stage those documents that have status equal to "A".

Second Stage: The $group stage groups the documents by the cust\_id field to calculate the sum of the amount for each unique cust\_id.

The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document.

Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use operators for tasks such as calculating the average or concatenating a string.

The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB.

The aggregation pipeline can operate on a sharded collection.

The aggregation pipeline can use indexes to improve its performance during some of its stages. In addition, the aggregation pipeline has an internal optimization phase. See Pipeline Operators and Indexes and Aggregation Pipeline Optimization for details.

**Map-Reduce**

As of MongoDB 5.0 the map-reduce operation is deprecated.

An aggregation pipeline provides better performance and usability than a map-reduce operation.

Map-reduce operations can be rewritten using aggregation pipeline operators, such as $group, $merge, and others.

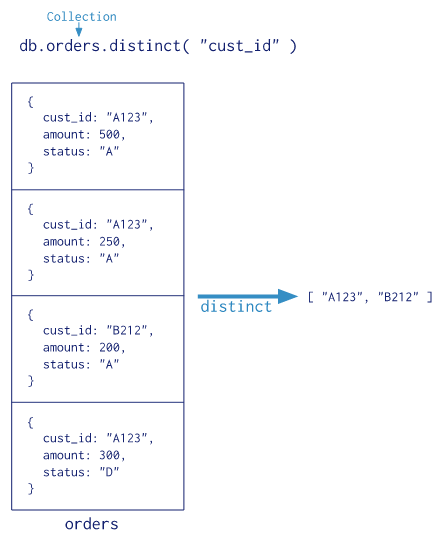
For map-reduce operations that require custom functionality, MongoDB provides the $accumulator and $function aggregation operators starting in version 4.4. Use these operators to define custom aggregation expressions in JavaScript.

For examples of aggregation pipeline alternatives to map-reduce operations, see Map-Reduce to Aggregation Pipeline and Map-Reduce Examples.

Single Purpose Aggregation Operations

MongoDB also provides db.collection.estimatedDocumentCount(), db.collection.count() and db.collection.distinct().

All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of an aggregation pipeline.



**4.Explain the indexing in MongoDB and how we retrieve the data using indexes?**

MongoDB uses indexing in order to make the query processing more efficient. If there is no indexing, then the MongoDB must scan every document in the collection and retrieve only those documents that match the query. Indexes are special data structures that stores some information related to the documents such that it becomes easy for MongoDB to find the right data file. The indexes are order by the value of the field specified in the index.

**Creating an Index :**  
MongoDB provides a method called createIndex() that allows user to create an index.

**Syntax –**

db.COLLECTION\_NAME.createIndex({KEY:1})

The key determines the field on the basis of which you want to create an index and 1 (or -1) determines the order in which these indexes will be arranged(ascending or descending).

Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. ... An index normally includes a "key" or direct link to the original row of data from which it was copied, to allow the complete row to be retrieved efficiently

**5. Explain about mongo Client and mongoose in detail?**

MongoDB Client

The MongoClient class is where the initial connection to the MongoDB server is defined.

Create a new Mongo client connection:

|  |
| --- |
| let client = try! MongoClient(uri: "<mongodb://localhost>") |

Closing the Connection

Once the connection is established and the database and collections have been defined, set the connection to close once completed using defer. This is done in reverse order: close collections, then databases, and then finally the client connection.

|  |
| --- |
| defer {      collection.close()      db.close()      client.close()  } |

Create Database Reference

getDatabase returns the specified MongoDatabase using the current connection.

|  |
| --- |
| let db = client.getDatabase(      databaseName: <String>      ) |

Parameters

* **databaseName:** String name of database to be used

Create Collection Reference

getCollection returns the specified MongoCollection from the specified database using the current connection.

|  |
| --- |
| let collection = client.getCollection(      databaseName: <String>,      collectionName: <String>      ) |

Parameters

* **databaseName:** String name of database to be used
* **collectionName:** String name of collection to be retrieved

Get Current Mongo Server Status

serverStatus returns: a Result object representing the server status.

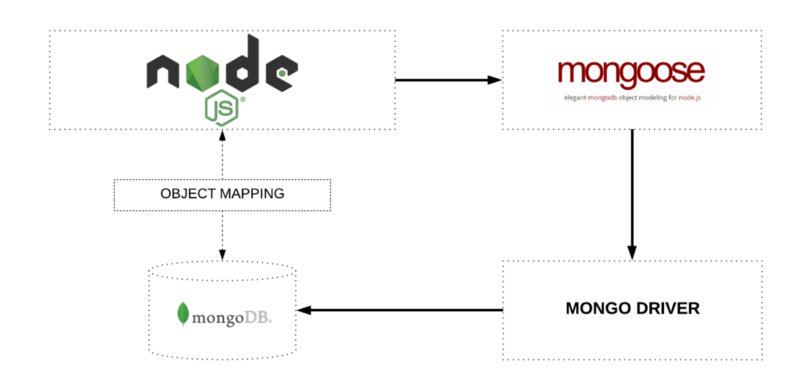
|  |
| --- |
| let status = client.serverStatus() |

Return String Array of Current Database Names

Use databaseNames to build a string array of current database names:

|  |
| --- |
| let dbnames = client.databaseNames() |

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.

Object Mapping between Node and MongoDB managed via Mongoose

**6. Explain match and set in detail?**

$match (aggregation)

Definition

$match

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

The $match stage has the following prototype form:

{ $match: { <query> } }

$match takes a document that specifies the query conditions. The query syntax is identical to the read operation query syntax; i.e. $match does not accept raw aggregation expressions. Instead, use a $expr query expression to include aggregation expression in $match.

Behavior

Pipeline Optimization

Place the $match as early in the aggregation pipeline as possible. Because $match limits the total number of documents in the aggregation pipeline, earlier $match operations minimize the amount of processing down the pipe.

If you place a $match at the very beginning of a pipeline, the query can take advantage of indexes like any other db.collection.find() or db.collection.findOne().

Restrictions

The $match query syntax is identical to the read operation query syntax; i.e. $match does not accept raw aggregation expressions. To include aggregation expression in $match, use a $expr query expression:

{ $match: { $expr: { <aggregation expression> } } }

You cannot use $where in $match queries as part of the aggregation pipeline.

You cannot use $near or $nearSphere in $match queries as part of the aggregation pipeline. As an alternative, you can either:

Use $geoNear stage instead of the $match stage.

Use $geoWithin query operator with $center or $centerSphere in the $match stage.

To use $text in the $match stage, the $match stage has to be the first stage of the pipeline.

Views do not support text search.

Examples

The examples use a collection named articles with the following documents:

{ "\_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }

{ "\_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }

{ "\_id" : ObjectId("55f5a192d4bede9ac365b257"), "author" : "ahn", "score" : 60, "views" : 1000 }

{ "\_id" : ObjectId("55f5a192d4bede9ac365b258"), "author" : "li", "score" : 55, "views" : 5000 }

{ "\_id" : ObjectId("55f5a1d3d4bede9ac365b259"), "author" : "annT", "score" : 60, "views" : 50 }

{ "\_id" : ObjectId("55f5a1d3d4bede9ac365b25a"), "author" : "li", "score" : 94, "views" : 999 }

{ "\_id" : ObjectId("55f5a1d3d4bede9ac365b25b"), "author" : "ty", "score" : 95, "views" : 1000 }

Equality Match

The following operation uses $match to perform a simple equality match:

db.articles.aggregate(

[ { $match : { author : "dave" } } ]

);

The $match selects the documents where the author field equals dave, and the aggregation returns the following:

{ "\_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }

{ "\_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }

Perform a Count

The following example selects documents to process using the $match pipeline operator and then pipes the results to the $group pipeline operator to compute a count of the documents:

db.articles.aggregate( [

{ $match: { $or: [ { score: { $gt: 70, $lt: 90 } }, { views: { $gte: 1000 } } ] } },

{ $group: { \_id: null, count: { $sum: 1 } } }

] );

In the aggregation pipeline, $match selects the documents where either the score is greater than 70 and less than 90 or the views is greater than or equal to 1000. These documents are then piped to the $group to perform a count. The aggregation returns the following:

{ "\_id" : null, "count" : 5 }

$set (aggregation)

$set

Adds new fields to documents. $set outputs documents that contain all existing fields from the input documents and newly added fields.

The $set stage is an alias for $addFields.

Both stages are equivalent to a $project stage that explicitly specifies all existing fields in the input documents and adds the new fields.

$set has the following form:

{ $set: { <newField>: <expression>, ... } }

Specify the name of each field to add and set its value to an aggregation expression. For more information on expressions, see Expressions.

Behavior

$set appends new fields to existing documents. You can include one or more $set stages in an aggregation operation.

To add field or fields to embedded documents (including documents in arrays) use the dot notation. See example.

To add an element to an existing array field with $set, use with $concatArrays. See example.

Examples

Using Two $set Stages

Create a sample scores collection with the following:

db.scores.insertMany([

{ \_id: 1, student: "Maya", homework: [ 10, 5, 10 ], quiz: [ 10, 8 ], extraCredit: 0 },

{ \_id: 2, student: "Ryan", homework: [ 5, 6, 5 ], quiz: [ 8, 8 ], extraCredit: 8 }

])

The following operation uses two $set stages to include three new fields in the output documents:

db.scores.aggregate( [

{

$set: {

totalHomework: { $sum: "$homework" },

totalQuiz: { $sum: "$quiz" }

}

},

{

$set: {

totalScore: { $add: [ "$totalHomework", "$totalQuiz", "$extraCredit" ] } }

}

] )

The operation returns the following documents:

{

"\_id" : 1,

"student" : "Maya",

"homework" : [ 10, 5, 10 ],

"quiz" : [ 10, 8 ],

"extraCredit" : 0,

"totalHomework" : 25,

"totalQuiz" : 18,

"totalScore" : 43

}

{

"\_id" : 2,

"student" : "Ryan",

"homework" : [ 5, 6, 5 ],

"quiz" : [ 8, 8 ],

"extraCredit" : 8,

"totalHomework" : 16,

"totalQuiz" : 16,

"totalScore" : 40

}

Adding Fields to an Embedded Document

Use dot notation to add new fields to embedded documents.

Create a sample collection vehicles with the following:

db.vehicles.insertMany([

{ \_id: 1, type: "car", specs: { doors: 4, wheels: 4 } },

{ \_id: 2, type: "motorcycle", specs: { doors: 0, wheels: 2 } },

{ \_id: 3, type: "jet ski" }

])

The following aggregation operation adds a new field fuel\_type to the embedded document specs.

db.vehicles.aggregate( [

{ $set: { "specs.fuel\_type": "unleaded" } }

] )

The operation returns the following results:

{ \_id: 1, type: "car", specs: { doors: 4, wheels: 4, fuel\_type: "unleaded" } }

{ \_id: 2, type: "motorcycle", specs: { doors: 0, wheels: 2, fuel\_type: "unleaded" } }

{ \_id: 3, type: "jet ski", specs: { fuel\_type: "unleaded" } }

Overwriting an existing field

Specifying an existing field name in a $set operation causes the original field to be replaced.

Create a sample collection called animals with the following:

db.animals.insertOne( { \_id: 1, dogs: 10, cats: 15 } )

The following $set operation overrides the cats field:

db.animals.aggregate( [

{ $set: { "cats": 20 } }

] )

The operation returns the following document:

{ \_id: 1, dogs: 10, cats: 20 }

It is possible to replace one field with another. In the following example the item field substitutes for the \_id field.

Create a sample collection called fruits contains the following documents:

db.fruits.insertMany([

{ "\_id" : 1, "item" : "tangerine", "type" : "citrus" },

{ "\_id" : 2, "item" : "lemon", "type" : "citrus" },

{ "\_id" : 3, "item" : "grapefruit", "type" : "citrus" }

])

The following aggregration operation uses $set to replace the \_id field of each document with the value of the item field, and replaces the item field with a string "fruit".

db.fruits.aggregate( [

{ $set: { \_id : "$item", item: "fruit" } }

] )

The operation returns the following:

{ "\_id" : "tangerine", "item" : "fruit", "type" : "citrus" }

{ "\_id" : "lemon", "item" : "fruit", "type" : "citrus" }

{ "\_id" : "grapefruit", "item" : "fruit", "type" : "citrus" }

Add Element to an Array

Create a sample scores collection with the following:

db.scores.insertMany([

{ \_id: 1, student: "Maya", homework: [ 10, 5, 10 ], quiz: [ 10, 8 ], extraCredit: 0 },

{ \_id: 2, student: "Ryan", homework: [ 5, 6, 5 ], quiz: [ 8, 8 ], extraCredit: 8 }

])

You can use $set with a $concatArrays expression to add an element to an existing array field. For example, the following operation uses $set to replace the homework field with a new array whose elements are the current homework array concatenated with another array containing a new score [ 7 ].

db.scores.aggregate([

{ $match: { \_id: 1 } },

{ $set: { homework: { $concatArrays: [ "$homework", [ 7 ] ] } } }

])

The operation returns the following:

{ "\_id" : 1, "student" : "Maya", "homework" : [ 10, 5, 10, 7 ], "quiz" : [ 10, 8 ], "extraCredit” : 0 }