Jagadeesh Palaniappan

Tech superstars  United States of America

MEAN –Stack

Contents

[NodeJS Intro 2](#_Toc447837926)

[NodeJS Module Export 3](#_Toc447837927)

[**‘module.export’** vs **‘export’** 3](#_Toc447837928)

[MongoDB Intro 5](#_Toc447837929)

[Chapter-2 (Mongoose) 7](#_Toc447837930)

[Mongoose Intro 7](#_Toc447837931)

[Mongoose -Sample Code 8](#_Toc447837932)

[MongoDB –Important Concepts -Summary 9](#_Toc447837933)

[MongoDB -Schema Design 9](#_Toc447837934)

[1. Store What You Query For 10](#_Toc447837935)

[2. Principle Of Least Cardinality 11](#_Toc447837936)

[MongoDB Indexes 12](#_Toc447837937)

[3. Retail Application Schema (Product) 14](#_Toc447837938)

[4. Retail Application Schema (Category) 15](#_Toc447837939)

[5. Retail Application Schema (User and Cart) 17](#_Toc447837940)

[6. Advanced Mongoose Features (Virtual) -or- (Virtual Field) -or- (Virtual Property) 18](#_Toc447837941)

[7. Advanced Mongoose Features (Custom Setters + Sorting By Currency) 19](#_Toc447837942)

# NodeJS Intro

## NodeJS Module Export

### ‘module.export’ vs ‘export’

rocker.js

exports.printMyName = function() {  
 console.log('My name is Lemmy Kilmister');  
};

index.js

var rocker = require('./rocker.js');  
rocker. printMyName(); // 'My name is Lemmy Kilmister'

**module.exports is the real deal**.

exports is just module.exports's little helper.

* All exports does is collect properties and attach them to module.exports properties --if module.exports doesn't have something on it already.
* If there's something assigned to module.exports already, everything on exports is ignored.

Put the following in rocker.js:

module.exports = 'ROCK IT!';  
exports.name = function() {  
 console.log('My name is Lemmy Kilmister');  
};

And this in another file, and run it:

var rocker = require('./rocker.js');  
rocker.name(); // TypeError: Object ROCK IT! has no method 'name'

* The rocker module completely ignored exports.name, and returned a string 'ROCK IT!'.
* If you don't set module.exports to anything explicitly, the properties of exports and attached to it and returned.
* Your modules can be any legal JavaScript object - boolean, number, date, JSON, string, function, array, and so on. Your module is whatever you set module.exports to.

In this case, your module is a class:

module.exports = function(name, age) {  
 this.name = name;  
 this.age = age;  
 this.about = function() {  
 console.log(this.name +' is '+ this.age +' years old');  
 };  
};

and you'd use it this way:

var Rocker = require('./rocker.js');  
var r = new Rocker('Ozzy', 62);  
r.about(); // Ozzy is 62 years old

In this case, your module is an array:

module.exports = ['Lemmy Kilmister', 'Ozzy Osbourne', 'Ronnie James Dio', 'Steven Tyler', 'Mick Jagger'];

and you may use it this way:

var rocker = require('./rocker.js');  
console.log('Rockin in heaven: ' + rocker[2]); //Rockin in heaven: Ronnie James Dio

So you get the point now - if you want your module to be of a specific object type, use module.exports;

if you want your module to be a typical module instance, use exports.

The result of attaching properties to module.exports is akin to attaching properties to exports. For example this:

module.exports.name = function() {  
 console.log('My name is Lemmy Kilmister');  
};

does the same thing as:

exports.name = function() {  
 console.log('My name is Lemmy Kilmister');  
};

But note that, they are not the same thing.

As I said earlier module.exports is the real deal, exportsis just its little helper.

Having said that,

* exports is the recommended object unless you are planning to change the object type of your module from the traditional 'module instance' to something else.

As long are you don't overwrite the module.exports object with an assignment operation, anything attached to module.exports and exports will be available in the 'required' module.

If this is the content of your module:

module.exports.age = 68;  
exports.name = 'Lemmy Kilmister';

The following code would work fine:

var rocker = require('./rocker.js');  
console.log('%s is %s', rocker.name, rocker.age); // Lemmy Kilmister is 68

BUT

if you overwrite module.exports with anything in your module, it will fail:

module.exports = 'LOL';  
module.exports.age = 68;  
exports.name = 'Lemmy Kilmister';

or

module.exports.age = 68;  
exports.name = 'Lemmy Kilmister';  
module.exports = 'WTF';

the order doesn't matter, rocker.age and rocker.name will now be undefined.

Also, note: just because module.exports.age and exports.name are exported, does not mean you should use a combination of both. My recommendation is to stick to exports.\*, and be aware ofmodule.exports.\*.

# MongoDB Intro

#### Core MongoDB-NodeJS-Driver

|  |
| --- |
| **var *mongodb*** = **require**(**'mongodb'**); **var *uri*** = **'mongodb://localhost:27017/movies'**;   ***mongodb***.MongoClient.**connect**(***uri***, **function**(error, db) {   **if** (error) {  *//Mongodb Connection Failed* **console**.log(error);  process.exit(1);  }   *//Mongodb Connection Succeded* **var** doc = {  **"title"**: **"Bahubali"**,  **"year"**: 2015,  **"director"**: **"Raj Mouli"** };   db.**collection**(**'movies'**).insert(doc, **function**(error, result) {   **if** (error) {  *//Insert Failed* **console**.log(error);  process.exit(1);  }   *//Insert Succeded    //Find that All "Raj Mouli" Movie docs from Collection* **var** query = {  **"director"**: **"Raj Mouli"** }  db.**collection**(**'movies'**).find(query).toArray(**function**(error, docs) {  **if** (error) {  *//Find Movies Failed* **console**.log(error);  process.exit(1);  }   *//Find Movies Succeded* docs.forEach(**function**(doc) {  *//Print All "Raj Mouli" Moview Records* **console**.log(**JSON**.stringify(doc));  });   process.exit(0);   });   *//JNOTE ::   /\*  db.collection('movies').find(query) //return 'cursor' //record by record   //That's the reason, they have helper method '.toArray' --which provides all docs.   db.collection('movies').find(query).toArray(...) //returns all docs   \*/* });  }); |

* Added MongoDB Node Module ("mongodb": "2.0.27")
  + require('mongodb');
* Database Connection
  + ***mongodb***.MongoClient.**connect**(***uri***, **function**(error, db) {..})
  + callback function will have the
    - (error) as first parameter (if “Connection” failed)
    - database connection (db) as second parameter
* Find All / Select All
  + db.**collection**(**'movies'**).find( ) (**function**(error, cusror) {..})
    - find() only provides cursor of each record / doc
    - That’s why we need .toArray() to get the all documents
      * --instead of fetching one by one record with cursor
  + db.**collection**(**'movies'**).find( ).toArray(**function**(error, docs) {..})
  + callback function will have the
    - (error) as first parameter (if “Find” failed)
    - docs //contains all found documents (movie docs)
* Find Only Specific / Selecting only particular
  + db.**collection**(**'movies'**).find({ **"director"**: **"Raj Mouli"** }).toArray(**function**(error, docs) {..})
  + callback function will have the
    - (error) as first parameter (if “Find” failed)
    - docs //contains all found documents (movie docs)
* Insert
  + **var** doc = {…..};
  + db.**collection**(**'movies'**).insert(doc, **function**(error, result) {…})
  + callback function will have the
    - (error) as first parameter (if “Insert” failed)
    - result //some meta data

# Chapter-2 (Mongoose)

## Mongoose Intro

|  |  |
| --- | --- |
|  | Mongoose (ODM) (Object Data Mapping)  * It helps, [NodeJS Object] – [MongoDB Document] mapping * Mongoose provides features like   + Schema Validation   + Pseudo Joins,..etc.. |
| C:\JUNIVERSE\WORKSPACE\NODEJS\INTERNET-PROJECTS\EDX\jlearn\ch1\2\mongoose-intro.png | |

## Mongoose -Sample Code

|  |
| --- |
| Mongoose Schema Example |
| var mongoose = require('mongoose');  module.exports = new mongoose.Schema({  {  name: {type: String, required: true},  email: {  type: String,  required: true,  match: /.+@.+\..+/,  lowercase: true  }  loggedInCount: {type: Number, default: 0}  }); |

|  |
| --- |
| Mongoose Example |
| var mongoose = require('mongoose');  var userSchema = require('./userSchema');  mongoose.connect('mongodb://localhost:27017/myApp');  var User = mongoose.model('User', userSchema, 'users');  var myUser = new User({name: 'Jagadeesh', email: 'jagadeeshtechgeek@gmail.com'});  //Handy .save() Function  myUser.save(function(error){  if (error) {  //Insert Failed  console.log(error);  process.exit(1);  }  //Insert Succeeded  User.find({email: 'jagadeeshtechgeek@gmail.com'}, function(error, userDocs){    if (error) {  //Find User Failed  console.log(error);  process.exit(1);  }  //Find User Succeded  console.log(JSON.stringify(docs[0]));    process.exit(0);  });  }); |

|  |
| --- |
| **MongoDB –Important Concepts -Summary** |
| MongoDB Schema Design Principles :   1. Store What You Query For    * MongoDB -Schema should closely match the Data you want to display to the End-User.    * Sometimes, even store Pre-Computed values -instead of computing each request 2. Principle Of Least Cardinality   This principle provides you a guideline for,   * + how to resolve one to many and many to many relationships   + “Arrays That Grow Without Bound” –will increases the Document size –it is bad for MongoDB |
| 1. Mongo Indexes    * Need not to scan each and every Document in Collection    * Instead, scan only the Index created Map –check where to look for matching Documents    * Multi-key Index that -- keeps track of [array - values] |
| Retail Application Schema:   1. Product    * Implicit '\_id' //Need not to specify    * Auto Type Casting (2 --> "2")    * required, enum, match 2. Category    * “Category” Document has been embedded into each “Product” Document    * Category is duplicated in Products 3. User and Cart    * User and Product have a “Many-To-Many” Relationship    * User Schema has Sensitive information (Oauth ID, Cart)    * User schema needs, Field specific “Access Control”    * MongDB doesn't have any "Notion Of Access Control"    * MongDB -Queries have a "Notion Of Projection"    * Also, Prevent “Editing” Sensitive Information -by allowing only required “Sub-Documents” |
| Advanced Mongoose Features :   1. Virtual    * Dynamic field //Not stored in Database    * Similar to helper function,    * But No function calls required, also includes when we convert Mongo Object to JavaScript Object 2. Custom Setters + Sorting By Currency    * Added one additional field with fixed currency USD    * Updating that variable whenever the value price value changes //By overriding the ‘price’ setter function    * Sorting the By Priced –based on the additional field |

# MongoDB -Schema Design

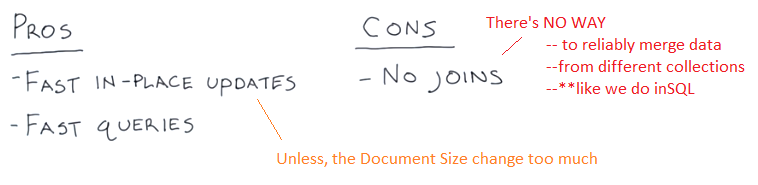
#### MongoDB is Schema Less Database

* Which means we can store ANYTHING as we want
  + whatever the Type / whatever the Document / whatever the Collection
* Just because, Wen can store anything in document
  + It doesn’t mean that, We SHOULD
* In order to get “Good Productivity” from Mongo DB –from Developer perspective
  + We should think carefully about **“Schema Design”**

## Store What You Query For

#### The first Schema Design --Principle you will learn about,

* Result of these below Three – MongoDB - Characterstics



Examples

|  |  |
| --- | --- |
|  | * Each ‘Review’ tracks which ‘User’ Posted it   --   * In order to get a User and their corresponding Reviews, * you have to run 2 queries, one for User & and one for each of their reviews. * There's no way to do this with one query and MongoDB * With this Schema, * there is no way to Sort- Users by their average Review -Score   -- |
|  | * We have to **Pre-Compute** the [Average Review Score] * and store it in the User Document * if you want to sort by Review Score   So, In General :  In order to take advantage of MongoDB strengths,   * MongoDB -Schema should closely match the Data you want to display to the End-User.   For instance,   * if we want to display the average review score for a user, * we should track the average review score in the user document * Rather than re-computing it every time you load a user by loading each and every individual review.   **In One Word:**  “Store What You Query For”  In SQL databases terms, This is NOT a “NORMALIZED FORM”.  However, Storing exactly what you query for   * is Easier to understand, * also “Less-Data-Transformation” between client & database. * It's also better “Performance”,   + because reading a single MongoDB document requires fewer different non-sequential hard drive reads   + than executing an SQL query with multiple joins. |

## Principle Of Least Cardinality

|  |  |
| --- | --- |
|  | This principle provides you a guideline,   * How to resolve “One To Many” & “Many To Many” Relationships --with MongoDB |

#### Note: ”Arrays That Grow Without Bound” –is bad for Mongo DB

|  |  |
| --- | --- |
| MongoDB documents can contain arrays.   * There's nothing to stop you from including every single Review a user posts in the User documen. * Depending on your use case, that may be a decent idea.   But,… | But beware,  [Arrays] that Grow Without Bound are a   * Very bad “MongoDB- Anti-Patten”   If you can't Limit the number of Reviews a User can post,   * your Document size will become massive * This is bad for MongoDB performance * This is bad because of MongoDB's 16 megabyte document size limit * Also It is bad because it wastes network throughput.   As a web developer, you should defend your network throughput  from unnecessary data because network throughput is most scarcest resource. |
|  | Good Design :   * in case of unbounded number of reviews * Lets 'Review'   + to track which user posted them   + rather than listing out all the reviews in the User doc.   This is the principle of **“Least Cardinality”**   * Tracking the User in the Review document results in smaller array sizes than * Tracking the Review documents in the User documents.   In this scenario, User can post an unlimited number of reviews. |
|  | This principle can also be applied to Many To Many relationships.  Lets assume a MongoDB Design for meetup.com,   * Users register for events with a capped number of attendees.   In SQL the schema design would be,   * You would have a MAPPING-TABLE for USERS and EVENTS. This table rows contain a USER\_ID and an EVENT\_ID.   In MongoDB the solution is not very simple,   * If you expect millions of attendees -or- unlimited attendees * Just like SQL we need a **“Mapping Collection"** * if you expect few hundred attendees -or- limited attendees, * We can de-normalize this many to many relationship * by keeping a list of attendees UserIDs in the Event document itself.   This is because you assume that the number of attendees for any given event is never going to grow without bound and will be couple hundred.  The right MongoDB schema design always   * depends on the “Store What You Query” -Principle.   But always remember,   * “Arrays That Grow Without Bound” are always a bad choice in MongoDB. |

## MongoDB Indexes

* Indexes are a way to get Consistent Performance from your queries --as your data grows.
* Indexes are key to get good performance from MongoDB.
* MongoDB indexes are pretty similar to SQL databases

Indexes helps MongoDB to pre-compute the results of a query,

* Just like, Book [Index Page] helps to find certain Topic - without searching the entire book.
* Indexes helps query to find MongoDB Document quicker.

|  |  |
| --- | --- |
| Without Index (Small Datasets)   * Query runs faster * Ok for Small Datasets   As your data grows,   * Collection Scan will become more and more expensive (operation). | Why Because?  This query search each and every single Document in the Users Collection   * to find documents that matches criteria * looks for <'name': 'John'> in each and every Document and fetches only the matched Documents |
|  | Index Creation :  When you create an index,   * MongoDB creates a Data Structure that maps the values of the “name” field to Documents that have that value of the name field.   So now when you ask MongoDB for Documents where 'name' is equal to 'John',   * MongoDB doesn't need to search every single Document * It just reads from the map. |
|  | |

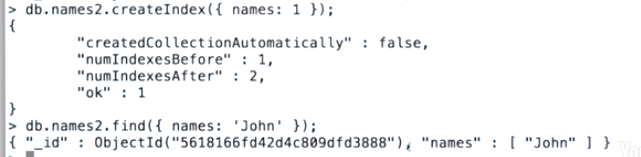
#### MongoDB [Multi-Key –Index]

* Multi-Key Index is an index that -- keeps track of [array - values] in a field
* Multi-Key indexes speed up queries that require scanning through arrays

If we not keeping Multi-Key Index on Array-- which we have to find that field frequently

* Collection scan would have to scan through every single [array element] in every single document in Collection

Creating a Multi-Key index is as same as normal index,



Multi-Key indexes are very powerful,

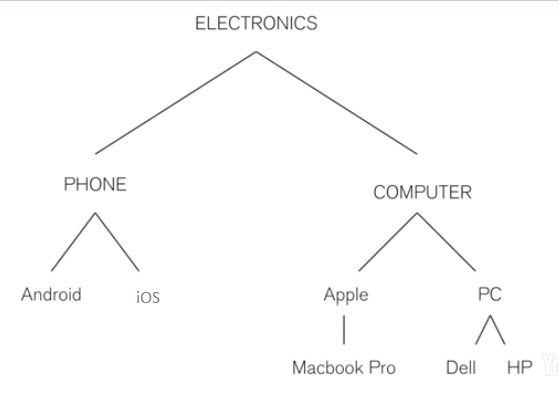
But, once again, be aware of “Arrays That Grow Without Bound”

* Make the Document Size larger.
* Large Documents take up more bandwidth
* Multi-Key-Indexes on large arrays have some significant performance

## Retail Application Schema (Product)

|  |
| --- |
| **var** mongoose **=** require**(**'mongoose'**);**  **var** Category **=** require**(**'./category'**);**  **var** productSchema **=**  **{**  name**:** **{**type**:** String**,** required**:** **true},**  pictures**:** **[{**type**:** String**,** match**:** **/^http:\/\//i}],**  price**:** **{**  amount**:** **{**type**:** Number**,** required**:** **true},**  currency**:{**  type**:** String**,**  **enum:** **[**'USD'**,**'EUR'**,**'GBP'**],**  required**:** **true**  **}**  **},**  category**:** Category**.**categorySchema //Embedding the 'Plain JavaScript Object' [Not Mongoose Schema]  **};**  module**.**exports **=** **new** mongoose**.**Schema**(**productSchema**);** //Mongoose Schema Object  module**.**exports**.**productSchema **=** productSchema**;** //Plain JavaScript Object |
| This schema represents, what you will display on an individual product view,   * product's name, * a list of pictures, * how much the product costs, * and the category that the product belongs to.   **Mongoose Implicit \_id Specification:**   * If we didn't specify an '\_id' field in this schema, (Mongoose will implicitly add \_id with type MongoDB ObjectId)   **Mongoose Auto Type Cast :**   * Mongoose will automatically try to "Cast” values to the specified Type   For instance, (2 🡪 "2")   * if you try to set the 'name' field to the Number type value //2 * Mongoose will convert two to a String for you. //"2" * Mongoose will refuse to save if there was an error casting a value.   For instance, ("Hi" 🡪 ERROR)   * if you set String value to price.amount, which is expected to be a Number * Mongoose will return an ERROR when validating the document   -----  **required -Property** in Product Schema: name, price.amount, currency   * Mongoose will not allow you to save a product that doesn't have a name, an amount,or a currency. * By default, all fields are Optional unless you specify required,   **enum -Property** [Only Allowed Values]   * Mongoose will only allow [USD, EUR, and GBP] values for price.currency * 'enum' property is available for String type.   **match -Property** specifies regex (regular expression) that the string must match   * In this case, image URL start with http://   **Category field is important for the Product Schema :**   * because you'll want to query for products based on the category's properties.   This is an instance/exmaple of “Store What You Query For”   * Category is a separate Collection, * Since we care about "Querying For Products By Category", * we should inline Category Object with Product for optimal query performance,   + especially since there is a one-to-many relationship of categories to products.   **Data Duplication** (Category Document might be duplicated and embedded in each Product Document)  Product’s Category is not going to change often,   * You will probably change the Product's Category as part of an expensive Database Migration.   But you will Query by category often,   * which is why in lining category is a good idea in this case * and thus an instance of "Store What You Query For". |

## Retail Application Schema (Category)



**You “Schema Design” should answer this question**

* Which Products are there in “Electronics”?
* What Category “iOS” is Sub Category of? //Useful When Users, Browse the Category
  + nothing but, What is iOS immediate Parent Category

|  |
| --- |
| **var** categorySchema **=**  **{**  \_id**:** **{**type**:** String**},**  **parent:** **{**type**:** String**,** ref**:** 'Category'**},**  **ancestors:** **[{**type**:** String**,** ref**:** 'Category'**}]** // With multi-key indexes  **}** |
| * **parent // Immediate Parent Category** * **ancestors // All Top Parent Category** |
| Data Looks Like this, {\_id: "Electronics", ancestors: ["Electronics"] }  {\_id: "Phones", parent: "Electronics", ancestors: ["Electronics", "Phones"] }  {\_id: "Android", parent: "Phones", ancestors: ["Electronics", "Phones", "Android"] }  {\_id: "iOS", parent: "Phones", ancestors: ["Electronics", "Phones", "iOS"] } |
| Find “Electronics” -All Sub Categories db.categories.find({ancestors: 'Electronics'});  {\_id: "Electronics", ancestors: ["Electronics"] }  {\_id: "Phones", parent: "Electronics", ancestors: ["Electronics", "Phones"] }  {\_id: "Android", parent: "Phones", ancestors: ["Electronics", "Phones", "Android"] }  {\_id: "iOS", parent: "Phones", ancestors: ["Electronics", "Phones", "iOS"] } Find “Phones” -Immediate Child Categories db.categories.find({parent: 'Phones'});  {\_id: "Android", parent: "Phones", ancestors: ["Electronics", "Phones", "Android"] }  {\_id: "iOS", parent: "Phones", ancestors: ["Electronics", "Phones", "iOS"] }   Find “Android” -All Parent Categories db.categories.find({\_id: ' Android'});  {\_id: "Android", parent: "Phones", ancestors: ["Electronics", "Phones", "Android"] } |

#### **“Category”** Document has been embedded into each **“Product”** Document

|  |
| --- |
| **var** mongoose **=** require**(**'mongoose'**);**  **var** categorySchema **=**  **{**  \_id**:** **{**type**:** String**},**  **parent:** **{**type**:** String**,** ref**:** 'Category'**},**  **ancestors:** **[{**type**:** String**,** ref**:** 'Category'**}]**  **}**  module**.**exports **=** **new** mongoose**.**Schema**(**categorySchema**);** //Mongoose Schema Object  module**.**exports**.**categorySchema **=** categorySchema**;**  //Plain JavaScript Object |

|  |
| --- |
| **var** mongoose **=** require**(**'mongoose'**);**  **var** Category **=** require**(**'./category'**);**  **var** productSchema **=**  **{**  name**:** **{**type**:** String**,** required**:** **true},**  pictures**:** **[{**type**:** String**,** match**:** **/^http:\/\//i}],**  price**:** **{**  amount**:** **{**type**:** Number**,** required**:** **true},**  currency**:{**  type**:** String**,**  **enum:** **[**'USD'**,**'EUR'**,**'GBP'**],**  required**:** **true**  **}**  **},**  category**:** Category**.**categorySchema //Embedding the 'Plain JavaScript Object' [Not Mongoose Schema]  **};**  module**.**exports **=** **new** mongoose**.**Schema**(**productSchema**);** //Mongoose Schema Object  module**.**exports**.**productSchema **=** productSchema**;** //Plain JavaScript Object |

#### The Good Features with this design is,

#### **Finding All Products -**under the 'Phones' category is Very Simple / Efficient / Better Performance.

|  |
| --- |
| db**.**products**.**find**({**'category.ancestors'**:** 'Phones'**});**  {  \_id: "ObjectId(...)",  name: "iPhone 6s",  category:{  \_id: "iOS",  parent: "Phones",  ancestors: ["Electronics", "Phones", "iOS"]  }    },  {  \_id: "ObjectId(...)",  name: "Nexus 5p",  category:{  \_id: "Android",  parent: "Phones",  ancestors: ["Electronics", "Phones", "Android"]  }  }  {  \_id: "ObjectId(...)",  name: "ASUS 2x",  category:{  \_id: "Android",  parent: "Phones",  ancestors: ["Electronics", "Phones", "Android"]  }  } |

## Retail Application Schema (User and Cart)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **User** and **Product** have a “Many-To-Many” Relationship  |  |  | | --- | --- | |  | * One User can have multiple Products in their Cart. * One Product can be in many of the Users Cart |   Users will typically have a small number of items in their cart.  --  But our Retail App,   * will have millions of Users buying millions of Products, * a single Product may be in many of their User carts at the same time.   By The Principle Of Least Cardinality,   * Assuming that User will have around 5 to 10 Products in their cart, * we can embed the list of Product IDs in the cart sub documents. |
|  | User schema needs, Field specific “Access Control”  * User schema have sensitive information like, Oauth ID, Cart information. * We shouldn’t give access to other User to see your Oauth ID field & Cart field * Other Users shouldn’t know what you buy.   Product and Category schemas, has no sensitive information, So we didn’t think about Access Control.  Unlike SQL databases,  MongDB doesn't have any **"Notion Of Access Control"**   * There's no way to tell MongoDB that only this User has Access to specific fields in the Collection.   But,  MongDB -Queries have a **"Notion Of Projection"**   * This enables you to hide specific fields from the output |
|  | Also, We wanted to prevent “Editing” Sensitive Information  * If you don’t wish to allow User to edit their Oauth Id mistakenly * For Example, Allow user to only Edit their “Profile” Information   We can achieve this by allowing only “Sub-Documents” during Update   * While Updating User Document, Allow only (user.profile) sub-document * Then there is no way User can modify Oauth ID or Cart information. |

## Advanced Mongoose Features (Virtual) -or- (Virtual Field) -or- (Virtual Property)

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  |  | | --- | --- | |  | * Virtuals are properties that are typically computed from other properties. * They are not persisted to the database, * but they can be accessed just like any other property. |   Declare a Virtual Property like this   * In this case, the getter function converts the currency into a Symbol and then concatenates the Price.   We can achieve the same effect with Helper function()  However, virtuals have some nice properties that make them a more convenient choice than helper functions.  Mongoose's methods **toObject()** and **toJSON()**  Mongoose's methods for converting,   * [Mongoose Document] into a plain old [JavaScript Object] * Both toObject and toJSON does same operation. * by default, toObject and toJSON do not include virtuals. |
|  | Lets see what make ‘virtual’ more convenient choice than ‘Helper function()’ : Those nice properties of virtual,   1. There is no function() calls 2. JSON.stringify(), product.toObject(),product.toJSON() also includes Virtual Property along with the other Product properties.  |  |  |  | | --- | --- | --- | | JSON.stringify(product) | product.toObject() | product.toJSON() | | JavaScript's  Built-In function | **Mongoose Function** | **Mongoose Function** | | converts,  [JavaScript Object] into  [JSON -String] | converts,  [Mongoose Document] into [JavaScript Object] | converts,  [Mongoose Document] into [JavaScript Object] | |

## Advanced Mongoose Features (Custom Setters + Sorting By Currency)

|  |  |
| --- | --- |
|  | Our App is having multiple currencies [USD, EUR,..]   * How will you Sort Products By price when products have different currencies? * Also products should be sorted using a recent exchange rate.   Solution:   * We want one additional Numeric Field (approximatePriceUSD), * this represents the Price of the Product in a fixed currency (USD)   Whereever we trying to change ‘amount’ -or- ‘currency’ property in Document,   * This **internal.approximatePriceUSD** will be updated automatically\*\* --based on the current Exchange rate ##  \*\* How this Automatic Update happen? By Overriding the setter property of ‘amount’ & ‘currency’,   * This seter will ba called whenever we set value for amount -or- currency field * Same value 'v' is getting retuned. NO\_CHANGE   But additionaly, one more PROCESS also happened,   * Populating the fixed curreny value (USD) internal.approximatePriceUSD  ## How it fetches current Exchange market Price?  * During Population it also fetches current USD exchange price * By calling fx() function, we will be getting the current Exchange price. |