Fixa varning:

“To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the MongoClient constructor.”

Använd i mongoose.connect :

const options= {

useUnifiedTopology:true,

useNewUrlParser:true

}

mongoose.connect("mongodb://localhost/mydatabase",options)

.then(()=>{

console.log("connected to mydatabase database")

})

Sätta validering reglar i schema :

Man kan inte lämna name, price, isAvailable fältet tömt :

const productSchema = new mongoose.Schema({

**name:{type:String, required:true},**

**price: {type:Number, required: true},**

tags:[ String],

date: {

type: Date, default:Date.now

},

**isAvailable: {type:Boolean, required:true}**

})

Om man prövar utan kommer man se error och ingen data kommer att skapas tömt.

\*\*\*\*Prova att skapa en produkt och spara den i databasen utan att name eller price variabel.

Skriv logiska valideringsregel:

price: {type:Number, required: function(){return this.isAvailable}},

Price, name ska vara ett obligatorisk fält när det finns tillgänglig i lager.

Mongoose validator: <https://mongoosejs.com/docs/validation.html>

|  |
| --- |
| Kopierat från mongoose sidan:  Mongoose has several built-in validators.   * All [SchemaTypes](https://mongoosejs.com/docs/schematypes.html) have the built-in [required](https://mongoosejs.com/docs/api.html#schematype_SchemaType-required) validator. The required validator uses the [SchemaType's checkRequired() function](https://mongoosejs.com/docs/api.html#schematype_SchemaType-checkRequired) to determine if the value satisfies the required validator. * [Numbers](https://mongoosejs.com/docs/api.html#schema-number-js) have [min and max](https://mongoosejs.com/docs/schematypes.html#number-validators) validators. * [Strings](https://mongoosejs.com/docs/api.html#schema-string-js) have [enum, match, minlength, and maxlength](https://mongoosejs.com/docs/schematypes.html#string-validators) validators. |

i vår app: Vi kan till ex. skriva om schema validering:

const productSchema = new mongoose.Schema({

name:{type:String,

minlength:3,

maxlength:30,

required:true},

price: {type:Number,

**min:300,**

**max:10000,**

required: function(){return this.isAvailable}},

tags:[ String],

date: {

type: Date, default:Date.now

},

isAvailable: {type:Boolean, required:true},

category: {

type:String,

required:true,

enum: ["skor","kläder", "kostym" ]

}

})

const product =mongoose.model("product", productSchema);

async function createProduct(){

const js = new product({

name:"nike",

price:1000,

tags:[ "nike model 19"],

isAvailable: true,

category:"skor"

})

try{

const res = await js.save()

console.log(res);

}

catch(e){

console.log(e)

}

}

createProduct();

|  |
| --- |
| There are more options: like uppercase, lowercase, trim and they have boolean value.  for instance: if you use: **uppercase : true** to product name field , product name will be saved in uppercase format in database even though you are using lowercase when you have saved in database.  name:{type:String,  minlength:3,  maxlength:30,  uppercase:true  required:true}, |

|  |
| --- |
| get and set properties: <https://mongoosejs.com/docs/tutorials/getters-setters.html>  Till ex: för att ge 30% rea på produkten kan man skapa en set options och i set options kan man en skapa en funktion som beräknar priset efter rean.  price: {type:Number,  min:300,  max:10000,  set: (p)=> p - p \*.30,  get :  till ex. för att få rund produkt pris :  get:(p)=> Math.round(p), |

Custom validator:

If the built-in validators aren't enough, you can define custom validators to suit your needs.

Till ex: om vi vill skapa en schema validering för tags :

tags:{

type:Array,

validate:{

validator: function(tag){

return tag.length >0;

},

message: "Lägg till tags för att göra produkterna SEO vanlig"

}

},

Övning:

Skapa schema validering för alla era fälten.

Update validator: lägg runValidators:true i update methoden:

{runValidators:true}

ejs.layout :

<%- include("mappensomharheadfilen/head.ejs")%>

OBS: efter <% “ - ” tecken inte “ = ” .

* minus för att lägga till html element i ejs filen.

Sass: i express :

<https://www.npmjs.com/package/node-sass-middleware>

i din main.js/index.js/app.js:

var sassMiddleware = require('node-sass-middleware')

app.use(sassMiddleware(

{src:path.join(\_\_dirname,"scss"),

dest:path.join(\_\_dirname,"public")

}

));

app.use(express.static(path.join(\_\_dirname, "public")));

---------------------------------------------------- -------------------------

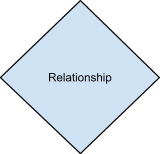
Pagination: med hjälp av skip, limit i mongoose find.

ER diagram:

# **ER-diagram**

****

Entity = en “sak”; något som har relationer och attribut vi är intresserade av. Personer, bilar, lägenheter etc.



Relationship = en “relation”, ett förhållande mellan två entities. Beskriver hur de hänger ihop, t.ex “äger”, “bor ihop med”, “betalar”.



Attribute = en “egenskap” en Entity har. Attribut bör hållas generella; “färg”, “namn”, “adress”.

**Att tänka på:**

Två entities kan bara kopplas samman via ett relationship, aldrig direkt med varandra.

Beskriv hur saker hänger ihop snarare än vad som händer.

Vad är intressant? Vad ska vi *använda* databasen till?

Relation mellan collection/modeller:

|  |
| --- |
| Till ex. Vilken användare har skrivit vilken kommentarer.  MongoDB/NoSql har inte någon funktionalitet som avgör att objekten kommer att ha relation mellan varandra. Vi får skriva själva relation. |

Normalisation : Man använder referenser

|  |
| --- |
| Till ex . document users:  {  name: “someone”  }  document comments  {  comments:”bla bla words…”  referens\_Id: id //id av someone  } |

DeNormalisation: Man använder embedded/inbyggd document.

|  |
| --- |
| document comments:  {  user: { name: “someone”}  } |

Vilket sätt ska man välja :

consistency(enkel att redigera): normalization

prestanda (till ex. Snabbsökning): denormalizaton.

Eller både två: Man kan använda hybrid approach.

normalisation approach:

|  |
| --- |
| const User = mongoose.model('User', new mongoose.Schema({  name: String,  email: String  }));  const Comment= mongoose.model('Comment', new mongoose.Schema({  name: String,  user: {  // Using user id as a type  type:mongoose.Schema.Types.ObjectId,  ref:"User"  }  }));  async function createUser(name, email) {  const user = new User({  name,  email  });  const result = await user.save();  console.log(result);  }  async function createComment(name, user) {  const comment = new Comment({  name,  user  })  const result = await comment.save();  console.log(result);  }  //createUser("rakib","mail@y.se")  //createComment("somecomments","UserID" ) |