MongoDB

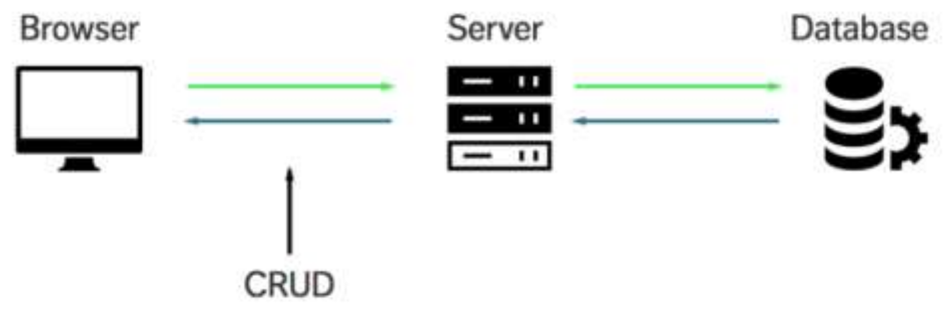
MongoDB is a database.

This is the place where you store information for your websites (or applications).

CRUD

- CRUD is an acronym for Create, Read, Update and Delete.
- It is a set of operations we get servers to execute (POST, GET, PUT and DELETE requests respectively). This is what each operation does:

- Create (POST) - Make something
- Read (GET)- Get something
- Update (PUT) - Change something
- Delete (DELETE)- Remove something



Installation

- Install MongoDB
  - https://docs.mongodb.com/manual/installation/
- Mongosh Install
  - https://docs.mongodb.com/mongodb-shell/install/

- **Prerequisites**
- To use the MongoDB Shell, you must have a MongoDB deployment to connect to.
- For a free mo deployment, you can use MongoDB Atlas.
- To learn how to run a local MongoDB deployment, see Install MongoDB.

In the terminal

```
mongosh
```

- This will open the shell and show details relating to mongoDB version, mongosh version,…

Basic Commands

Please refer to the cheat sheet

Mongoose

- A wrapper around mongoDB
- Mongoose is a Node. js-based Object Data Modeling (ODM) library for MongoDB.
- The problem that Mongoose aims to solve is allowing developers to enforce a specific schema at the application layer. In addition to enforcing a schema, Mongoose also offers a variety of hooks, model validation, and other features aimed at making it easier to work with MongoDB.
- To start using mongoose: `npm i mongoose`

Connect to DB

```
const mongoose=require(‘mongoose’)

mongoose.connect("mongodb://localhost/db“)
```

3 concepts to understand

- Schema: The schema just defines what the structure of your data looks like
- Model: A model is just the schema in a form that you can use
- Query: simply a query you are making against the mongoDB database.

Creating the schema

- You can create all schemas in one file
- But generally you have a file for each schema (ex: users.js)

```
const mongoose=require('mongoose')
const userSchema=new mongoose.Schema({
//this is where you put all the fields for your schema
 name: String,
 age: Number,
……
})
```

Creating the schema….continued

```
try{
            const user=await User.create({
           name:"Sara",
           age:55,
           hobbies:["weight lifting","bowling"],
           address:{
               street:"main street",
           },
})
console.log(user)
 } catch (e) {
       console.log(e.message)
       }}
```

More mongoose schema types

- String
- Number
- Date
- Mongoos.SchemaTypes.ObjectId //another object: the id of another object
- [String] //an array: example an array of string
- { key:type,

  key:type

}   //nested object

- **If in the script, the values are not compatible with the specified datatypes, it will give an n error.**
- **This error can be caught by wrapping the code inside a try/catch ..**

Creating the model

- In the same User.js schema file, create and export a model and let is use the userSchema built earlier.

```
module.exports=mongoose.model('User', userSchema)
```

- The first argument is the *singular* name of the collection your model is for.

Two ways to deal with nested objects in the schema

```
const userSchema=new mongoose
.Schema({

………

address: {street:String,

                city:String}

…………})
```

```
const adressSchema=new mongoo
se.Schema({

               street:String,

                city:String})

const userSchema=new mongoose
.Schema({

……..

address:addressSchema,

…………..

})
```

To create a new user object

- In the script.js file, import the Users module.

```
const mongoose = require('mongoose');

const User=require("./User")

const user=new User({
        name:"Sara",
        age:55})

//to save in DB call the user.Save()which is an
 asynchronour func

user.save().then(()=>console.log('user saved')
)
```

```
…….
run()
async function run(){
             const user=new User({
            name:"Sara",
            age:55,
              ….
            })
await user.save()
console.log(user)
 }
```

```
…….
run()
async function run(){
           const
user=await  User.create({
            name:"Sara",
            age:55,
              ….
            })
 user.name="Sally"
await user.save()
console.log(user)
 }
```

Field validation

- You need to start by passing an object instead of just a type.
- Example: Adding the required flag, the lowercase check, and the minlength

```
email:{
        minlength:10,
        lowercase:true,
        type:String,
        required:true
    },
```

Setting min and max

- Example: Adding a min and max

```
age:{
      type:Number,
      min:1,
      max:100,
},
```

Adding custom validation

- Example: Adding custom validation to check that the age is even

```
 age:{
     type:Number,
     min:1,
     max:100,
     //add custom validation
     validate:{
         //add a validate object..pass it a validator function that runs to check it this value is valid
          validator: v => v % 2===0,

          //specific a message. The message takes props object, and this props contains the value

            message:props=>`$(props.value) is not an even number`
     }
},
```

Note:

- When adding the validation on the model itself, you don’t have to worry about writing the validation somewhere else. Its all in one place.
- This custom validation will only run when you use the create or save method.

How to use findById()

```
…….
run()
async function run(){
          try{
           const user=async User.findById("578345934534")
             console.log(user)
               }
            catch( e)=>{
           console.log(e.message)(
 }
```

How to use find ()

```
…….
run()
async function run(){
          try{
           const user=async User.find ({name:”Sara”})
             console.log(user)
               }
            catch( e)=>{
           console.log(e.message)(
 }
```

How to use deleteOne()/deleteMany()

```
…….
run()
async function run(){
          try{
           const user=async User.deleteOne({name:"Sara")
             console.log(user)
               }
            catch( e)=>{
           console.log(e.message)(
 }
```

How mongoose deals with queries

- Mongoose implemented something known as queries. It allows you to write .where, allowing you to create your own query (your own find syntax) based on helper methods.

```
const user=await User.where("name").equals("kyle")
Console.log(user)
```

```
const user=await User.where("age").gt("2");
```

```
const user=await User.where("age").gt("2").where("name").equals("sara");
```

```
const user=await User.where("age").gt("2").where("name").equals("sara").limit(2)
```

```
const user=await User.where("age").gt("2").where("name").equals("sara").limit(2).select("age")
```

Adding methods to the schema (available on the instances)

```
const userSchema=new
mongoose.Schema({
……..
})

userSchema.methods.sayHi=function(){
Console.log(`Hi, my name is
${this.name}`)
}
```

```
Try{

const user=await
User.findOne(name:"Sara")
console.log(user)
user.sayHi()

}
…
```

Defining static methods (available on the model)

```
const userSchema=new
mongoose.Schema({
……..
})

userSchema.statics.findByName=functi
on(name){
return this.where({name:new
RegExp(name,'i')})

}
```

```
Try{
const user=await
User.findByName(name:"Sara")
console.log(user)
User.sayHi()

}
…
```

Adding onto a query

```
const userSchema=new
mongoose.Schema({
……..
})

userSchema,query.byName=function(n
ame){
return this.where({name:new
RegExp(name,'I')})



}
```

```
Try{
const user=await
User.find().byName("Sara")
console.log(user)



}
…
```

userSchema.virtual

```
const userSchema=new
mongoose.Schema({
……..
})
userSchema.virtual(‘namedEmail’).get(f
unction(){
Return `$(this.name)<${this.email}`>

})
}
//a property that now exists on
individual users
```

```
Try{
const user=await
User.findOne({name:”Sara”})
console.log(user)
console.log(user.namedEmail)




}
…
```

Middleware in mongoose

- Allows you to insert code in between different actions
- Middleware for saving, validating, removing
- userSchema.pre(“save”) , userSchema.pre(“validate”), userSchema.pre(“remove”)

```
const userSchema=new
mongoose.Schema({
……..
})
userSchema.pre(‘save’,function(next
){
          this.updatedAt=Date.now()
          Next()
})
```

```
Try{
const user=await
User.findOne({name:”Sara”})
console.log(user)
Await user.save()
console.log(user)

}
```

Middleware in mongoose

```
const userSchema=new
mongoose.Schema({
……..
})
userSchema.post(‘save’,function(doc
,next){
            doc.sayHi()
            next()
})
```

```
Try{
const user=await
User.findOne({name:”Sara”})
console.log(user)
Await user.save()
console.log(user)

}




}
…
```

Rest

- Rest (Representation State Transfer)
- a way of saying that a server responds to create, read, update, and delete requests in a standard way.
- The idea behind REST is to treat all server URLs as access points for the various resources on the server.

Example

- For example in this URL, http://abc.com/users, users represents the resource that the server is exposing
- The following URLs are used to create,read,update, and delete recources.
  - http://example.com/users
  - http://example.com/users
  - http://example.com/users/1
  - http://example.com/users/1
  - http://example.com/users/1
- The URLs that do not have an ID, act on the entire user's resource, while the URLs that have an ID act on only a single user resource.

- But as you may notice there are only two distinct URLs
- REST uses the four basic HTTP actions, GET, POST, PUT, and DELETE to determine what to do with each URL.
- If we add in those actions to the URLs it is much easier to see what each of the URLs do.
  - [GET] http://example.com/users
  - [POST] http://example.com/users
  - [GET] http://example.com/users/1
  - [PUT] http://example.com/users/1
  - [DELETE] http://example.com/users/1

http status codes

- the idea of status code is to give much information to the browser without having to do too much work
- https://www.restapitutorial.com/httpstatuscodes.html
- You have 5 categories 1xx-5xx
- 1XX: informational   (not relevant for building web APIs or websites)
- 2XX: success
- 3XX redirection
- 4XX client error
- 5XX server error

2XX success

- 200: ok (a very general request message)
- 201: created   (all post requests to create something)
- 204: no content (everything went well but you have nothing to return. Example when you delete something)

3XX: Redirection

- 304: Not modified (nothing has changed)
- A way to save bandwidth..the server sends you than nothing has changes so you can pull it from cache

4XX: Client error

- An error from the client side. Example user working with your API sent you some bad information
- 400: bad request. general error, you don’t know the exact reason. Example: sending wrong parameters
- 401: unauthorized. Accessing something that requires authentication and you didn’t pass it or it was wrong.
- 403: forbidden. The user did pass a key but what he is trying to access requires different permissions. Example: A basic user accessing an admin feature.
- 404: not found

5XX: Server Error

- Something broke on the serve (example: database down or the server side code throws an error)
- 500: Internal Server Error. A way of saying something broke on the server and that what the user is doing is not wrong