Live Coding 2

We have forked the repo and are able to run, but not able to see the database. So, we have created own cluster by changing the DB_URL. If we have 10 people working in a team, and they all want to contribute, then we need to create our own test database and add or delete something to the project so that the company's database will be safe. Git will be tracking all these changes and will register them. when we compare between the two branches everyone will have their own database with their own username and password.

How to get rid of this issue? Based on which system it is running, if this could have be initialized then it would be amazing. In order to do that, we have something called 'env' files. Environment variable are maintained by the operating system based on where it has been defined and it is accessible to all the process. We will just utilize the concept and will use the package 'dotenv'.

We will utilize the concept and use a npm package, **dotenv**. This allows us to create environment variables and use it in our system and wherever the project is running in the system it will get all the data.

We will first install it.

```
npm i dotenv
```

This is added to the package.jsons file. Now, we can create a file called '**`.env`**'. Git is not tracking this file, because it is mentioned in our '**`.gitignore`**' file. Instead of defining the variable string in the app.js file, we can write in '**`.evn`**' file

DATABASE_URL = 'mongodb+srv://express-attainu:qAmxNDPVztn2rSUG@cluster0.bg4zd.mongodb.net/yelpcamp-attainu?retryWrites=true&w=majority'

NOTE: It's a convention to have our environment variables as CAPS.

You'll define the name of the variable and then will give the value without the quotes. Next step is to call the .env package and load these environment variables inside the NodeJS application. It will be the same process for everyone and therefore everyone can make changes to the .env but not to the link in the app.js file.

All we have to do is config(), it automatically goes to the directory and finds the .env file. This is the default behavior.

The way to access it is little different and so we won't be accessing the variable inside the javascript. This .evn file has to be in the root level meaning, it should be in the same folder as the app.js file.

```
require('dotenv').config()
```

Once this is incorporated, we need,

```
process.env.PORT
```

We did something in Heroku as well. Because Heroku know on what port it will be severing. So, we put in the variable PORT and based on which system it is running its value will change.

```
console.log(process.env)
```

Apart from whatever we have defined, there are so many environmental variables attached to our system. All this information is added by operating system and maintains it. '**dotenv**' package goes through the '**env**' file and adds the variables inside it as well.

As git is not tracking it, we don't have to change it every time if someone new comes in wants to contribute.

Now in place of **DB_URL** we will use **process.env.DATABASE_URL**. That is done by the dotenv package. This package goes through the entire list of environment variable and converts it into an object. which mean, even we can de-structure it. There will not be any error, but it will override it.

Mongo Client Vs Mongoose:

In our Mongo Client, we were creating any json object and if we call insert, it would insert it irrespective of the structure. There is some new developer is created a json in a wrong way. Instead of password he wrote pass, and sent to the database. A section of database will have password and another section will have it as pass.

Although this is the main feature, this is susceptible to people creating **bugs**. So, for each next user who is signing up the property name will get changed. MongoDB does not complain, whatever json object is given it will store. This might affect the other functionalities. The login logic would be looking for the password, but it was stored as pass. Even though the data is present, the user will not be able to login.

The second issue is, we have to put in a callback with a variable **client**. Then we have given the collection name on which we are supposed to work.

Mongoose is a new wrapper or few lines of extra code, around the same mongo client package itself.

The benefit is, we will be able to define a schema.

Class is a blue print for my objects. whichever object we are creating using the structure, will always have those fields. Similarly, the same analogy is used. We can define a schema AKA class, which it will use to enforce this behavior. SO, if anyone adds a new field, it will not accept it as it is not already present in the schema.

In this example, it is the responsibility of the developer to enforce, if the password is a string, then it has to be entered as a string. But if we use mongoose, then we can define the schema and we can enforce, the password would be number type or string type and if someone tries to enter anything else, it would show an error.

Model:

We create a model based on Schema. When we create a model, we pass in the particular schema. Model knows about the schema and it does not need to use the client. We can directly say model.insert and it will directly insert in the proper collection. Once we define the model and schema, it will never happen that we are accidently using user model.insert, it will never insert in the other model. We are coupling it to a model so that functionally of add or removing it will always exist on the model itself.

If we try to insert something in a particular collection, then MongoDB will automatically create it for you. We will create a small route. We want to use mongoose in order to store user related data. We will start by creating a ‘model’ folder because there will be multiple collection so that organizational wise we will have clarity and user-understandable. The convention is to name the files in model folder with the first letter being a capital. All the schema related data is inside mongoose so we need to import mongoose package. Now, we want to define a schema.

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

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

```
    email: String,
    password: String

})
```

We have defined a schema (mongoose.Schema) and gave it a javascript object defining what my structure would look like in the database. This schema gets created and gets saved in the const User. The first step is completed.

Once we create a particular schema, we want to create a model out of it.

```
const User = mongoose.model('User', UserSchema)

module.exports = User
```

Once this done, although mongoose is smart enough to understand the schema, it is a good practice to name the collection name in the first parameter of (mongoose.model()). For this collection name User, use the particular schema (second parameter – UserSchema) and give us a model which is getting saved in the User variable.

For each of the collection that we have we will store it and export it. Wherever you want to use inert or delete operation we will be importing this model and do it.

```
app.get('/', (req, res) => {
    const newUser = {
        email: "abce@abce.com",
        password: "asdasd"
    }
})
```

After we get the particular text, we will create a document out of the model. Using the UserModel we are creatinga new document which is going to be stored in the database. It will go through this object all property and if there are any errors, then it will not allow you to add it.

This entire thing is creating a newUser document from my model.

```
const userDoc = new UserModel()
```

We have created the document but have not saved it in the document. So we will write,

```
userDoc.save()
```

We can see multiple options in the UserModel rather than client variable we previously had. We don't have specify what is the collection name, it will automatically figure it out based on the collection name and schema that we have given here.

Save() is an asycnhrnous operation and it returns a promise.

```
app.get('/', async (req, res) => {
    const newUser = {
        email: "abce@abce.com",
        password: "asdasd"
    }

    const userDoc = new UserModel()

    const result = await userDoc.save()
    console.log(result)

    res.send('Added to the database')
})
```

Review: I have created a simple object which can be created by req.body or form input from the user. We are important the UserModel on top from User.js file. Then we are creating a new user document from that model. Basically, we are creating a new object from a class. Using the schema, it creates a userDoc for the MongoDB and we can call the save operation. This data will be saved in our database. This is an asynchronous operation as it has to go through the network of internet, it will create a request to tell the MongoDB server to save this data and then it will send a response back stating that the given data has been saved.

The mongoose and NodeJS application have been connected. In the browser if we give localhost:5000, it would show 'Added to the database'. In our database we can see the details being saved as document.

The same data can be seen in the terminal as well. So, if this particular result has an _id attached to it, that means data has been inserted and we can add it to the database. If there is an error, then we can use the try-n-catch block.

```
const expHbs = require('express-handlebars')

app.engine('hbs', expHbs({extname: 'hbs'}))
app.set('view engine', 'hbs')
```

In the layouts we have our main.hbs as the entry level. If the pages need anything extra we can include it. We will create a signUp.hbs in the views folder. We define a route for it to render that page. We will not do it directly everything, as we have to define separate routes and the entire project file would become too huge. So, we will use the router to split the routes separately.

So, we will create a foldr called routes and we will add a user.js file.

```
const express = require('express')
const router = express.Router()
```

We are expecting a /signUp route where we will render the form.

```
const express = require('express')

const router = express.Router()

router.get('/signUp', (req, res) => {
    res.sender('signup')
})
```

Finally, we need to export this file, so

```
module.export = router
```

In the app.js we would use,

```
const userRouter = require('./routes/user')

app.use('/user', userRouter)
```

The GET part is done. this will be posting data, for which we need to create a handler.

```
router.post('/signUp', (req, res) => {

})
```

We will add the express.urlencoded in the app.js file. We have to import the user model,

```
const User = require('../models/User')
```

We will add a try-catch block for error handling. We will create a variable newUserDoc and will put the UserModel(req.body) value in it. We will await newUserDoc.save() in the saveUserDoc. If there is an error it will come to the error block. We want to redirect to my home page. So in app.js we will change the GET

route message to Welcome to Yelpcamp and in the catch block we log the error and then send an error message to the page.

```
router.post('/signUp', async (req, res) => {
    try {
        const newUserDoc = new UserModel(req.body)
        const saveUserDoc = await newUserDoc.save()
        res.redirect('/')

    } catch(error){
        console.log(error)
        res.send('Error Occured')
    }
})
```

Let's say email address is required in the hbs file. The browser will not allow me to click on the submit button. But someone who is smart enough can go the inspect and remove it. So, for this we added extra frontend javascript in the public folder and linked it. but if I'm able to define the constrain inside the model or schema itself, then a developer has forgotten to add that validation then database will not accept it.

For this purpose we can use mongoose. Right now, in our schema, we said that we need to have email and password, but to add this extra security or process, then we can change our schema.

We will make the email into an object and will add **type** and **required** property inside the object.

```
email: {
    type: String,
    required: true,
    lowercase: true,
    trim: true
},
```

Similarly in the password as well.

```
email: {
    type: String,
    required: true,
    lowercase: true,
    trim: true
},
password: {
    type: String,
```

```
    required: true,
},
createdAt: {
    type: Date,
    default: Date.now
}
```

It says user validation has failed: Path ‘email’ is required, Path ‘password’ is required. Notice that the server is not crashed and ‘**nodemon**’ is still running. ‘**Nodemon**’ has not restarted, because in the handler we are catching the error. So in the try-catch block we can make changes as,

```
} catch(error){
    console.log(error)
    res.send(`Internal Error Occured: ${error._message}`)
}
```

Now, if I signUp, it says, ‘**Internal Error Occurred: User validation failed**’