Notes

- What is **Authentication?**
  - It is just a way of identifying who someone is? ⇒ **Identification**.
- What is **Authorisation?**
  - It is a way of determining what access does someone have? ⇒ **Permission**.
- Without `Authentication` I can not buy a product on **Amazon**, even if i am **authenticated**, I cannot update or delete the product details.
- Most common `Authentication` .
  - Signup/Register/Create Account
  - Login.
- If `Email` and `Password` is correct only then `Login`.
- We have to match it with the data that is there in the database.
- Create a backend project now.
- `npm i mongoose express nodemon dotenv`

```
//server.js
const express=require("express")
const {connection}=require("./config/db")

const app=express()
app.use(express.json())

app.post("/register",async (req,res)=>{
  try{
    const user= new UserModel(req.body)
    await user.save()
  }catch(err){
    console.log(err)
  }
  res.send("Registered")
})

app.post("/login",async (req,res)=>{
  const {email,pass}=req.body
  try{
```

```
    const user=await UserModel.find({email,pass})
    if(user.length>0){
      res.send("Login Successful")
    } else {
      res.send("Login Failed")
    }
  } catch(err){
    console.log(err)
  }
})

app.listen(8080,async ()=>{
  try{
    await connection
  }cartch(err){
    console.log(err)
  }
  console.log("Running at 8080 Port")
})
```

```
//congig/db.js

const mongoose=require("mongoose")

const connection=mongoose.connect("mongodb_url")

module.exports={
  connection
}
```

```
//models/User.model.js

const userSchema=mongoose.Schema({
  email:Srtring,
  pass:String,
  name:String,
  age:Number
})

const UserModel=mongoose.model("user",userSchema)
```

Hashing

Hashing is done just to protect the password.

- Before storing the `password` in `database` while `signup`, we should hash it and then store it.
- **WHY? ⇒** Because in the database the password is visible and it can be miss used in case of `data dump` .
- So, the best way would be hashing it while storing in the database and then change it back to the plain text password while `login`.
- We are going to use a package for this, `bcrypt` .

Bcrypt

`Bcrypt` is a password hashing function designed to be secure and resistant to attacks such as brute-force and rainbow table attacks. It is often used in web applications and is widely regarded as a secure way to store user passwords.

`Bcrypt` is a one-way hash function, which means that it takes a plain text password as input and produces a fixed-length output that cannot be reversed to obtain the original password. This makes it difficult for attackers who gain access to a user database to determine the actual passwords of the users.

Overall, `bcrypt` is considered to be a strong password hashing function and is widely used by developers to ensure the security of their users' passwords.

- To understand this `npm` package in a better way just go through the documentations.

```
//while registering
app.post("/register",async (req,res)=>{
    const {name,email,pass,age}=req.body
    try{
        bcrypt.hash(pass, 8, async (err, hash)=>{
          const user=new UserModel({name,email,pass:hash,age})
          await user.save()
          res.send("Registered")
        });
    }catch(err){
        res.send("Error in registering the user")
        console.log(err)
    }
})
```

- **myPlainTextPassword** ⇒ This is just the password that the user is passing while `registering`.
- **saltRounds** ⇒ `Bcrypt` is also designed to be slow, which makes it computationally expensive for attackers to try out large numbers of passwords. This is done by applying a cost factor to the hash function, which determines how many times the password is hashed before the output is generated. The higher the cost factor, the slower the function, and the more secure the password storage. This is basically the `saltRounds`
- **hash** ⇒ It is basically the hashed password.

```
//while Login
app.post("/login",async (req,res)=>{
    const {email,pass}=req.body
    try{
        const user=await UserModel.find({email}
        if(user.length>0){
            bcrypt.compare(pass, user[0].pass, function(err, result) {
              if(result){
                  const token = jwt.sign({ course: 'backend' }, 'masai');
                  res.send({"msg":"Login Successfull","token":token})
                } else {res.send("Wrong Credntials")}
            });

        } else {
            res.send("Wrong Credntials")
        }
    } catch(err){
        res.send("Something went wrong")
        console.log(err)
    }

})
```

- You can also use `node-argon2`
- It is pretty similar to `bcrypt` only.

Authorization

- Now we are able to `register` and `login`, now we have to make some `authenticated endpoints` as well.

    - `/data` endpoint.
    - `/userdetails` endpoint
- Cannot be done at `frontend` as always they are not going to use `API` endpoint using `FE`. `Thunderclient` or `Postman` can also be used
- What should be done here?
    - **1st Way:** We can make them login at protected endpoints. (send email and password with request when they are accessing this endpoint)⇒Not the best way⇒Because this could be really annoying.⇒ Every time we have to go through the data and check it.
    - **2nd Way:** Access Token can be an option. Give them a token when they are logged in.
- Only check for that token while visiting the restricted endpoint.

```
//send token while login
app.post("/login",async (req,res)=>{
  const {email,pass}=req.body
  try{
    const user=await UserModel.find({email,pass})
    if(user.length>0){
      res.send({"msg":"Login Successful","token":"abc123"})
    } else {
      res.send("Login Failed")
    }
  } catch(err){
    console.log(err)
  }
})


app.get("/data",(req,res)=>{
  if(req,query.token==="abc123"){
    res.send("Loggen in and can access the data")
  } else {
    res.send("Not Loggen in, login first")
  }
})

app.get("/cart",(req,res)=>{
  if(req,query.token==="abc123"){
    res.send("Loggen in and can access the cart")
  } else {
    res.send("Not Loggen in, login first")
```

```
  }
}
```

Lets us look at better way to do this

- There is something that we can use, which is unique, and verifiable as well.
- **JWT⇒JSON WEB TOKEN**
- It is made up of three different things.
  - Algorithm to encrypt the string.
    - It can be decrypted.
  - Payload as well, any data, which will also be encrypted in the string.
  - A secret Key to decrypt it.

Install the JWT token

- `npm i jsonwebtoken`

```
const jwt=require("jsonwebtoken")

//while logging in
const token=jwt.sign({"course":"backend"},"masai")
```

- This token will be used in restricted routes now.

```
//while visiting /data or /userdetails
var token=req.headers.authorization
jwt.verify(token,"batman" (err,decoded )=>{
  if(err){
    res.send("Invalid Token")
    console.log(err)
  } else {
    res.send("Whatever you want can be sent over here")
  }
})
```

- We are not going to pass it in `queries`, we are going to take it from `headers`.

💡 **Resources**

1. https://jwt.io/introduction
2. https://www.npmjs.com/package/jsonwebtoken
3. https://www.npmjs.com/package/bcrypt