Live Coding 3

Topics:

- Seeding the Database
- Pull Request
- How to Save Password:

We will see how to encrypt and create a hash out of it.

Seeding the Database:

If we see users, we see email Id and password as plain string which is not a good approach. If someone gets access of this entire database, then they will also have user password and email in plain string. Most of the users keep one single password for most of there accounts. So, if someone has access can manipulate it.

So, how to git rid this issue? Hashing.

What is a HASH?

When the user enters their credentials, we take the password and give to an algorithm (hashing algorithm). Based on the content inside, it will generate a random string. The idea is to not have the plain string of the password. But the other problem would be, if another user, also sets the same password (accidently), in such case this particular password goes to the same algorithm and will generate the same hash. If the password is alike, then it will generate the same hash.

Let’s say password is ABC. When we put it inside the hashing algorithm and it generates XYZ. Now, user2 also has ABC as password. The concept of hashing is if you give the same input, it would give the same output. So, for the second user also hashing would give XYZ. Hashing concept is not limited to password alone.

Lets say we have a file1.txt and we can provide all the data to the hashing function and it will provide a unique hash. If someone opens file1.txt and make changes to it and then pass it through the hashing function it would generate a different hash. This will tell us that there has been some tampering with the file1.txt. So, if the input content is same, than it will create the same output for all cases.

In password case, it is not necessary to always hash it because it would not complete the work.

Let's say we have 10 users and 7 have the same password. Then the hashing algorithm will generate 7 same hashes. We are not using the hashing for the email for now. if someone gets access to my database, all he needs to do is decrypt one and he will get the password of all the 7 users. Part of the problem is solved by hashing, but not the entire problem.

Hashing solves one problem; we can generate a random string which is cryptic in such a way that humans will not understand. But if users have same password, then it would generate the same hash. In such case we will provide something called salt. Salt is like a rolling-key which keeps changing.

In such cases, we have one password pass1, and we have another password pass2. We will put it in the hash algorithm along with salt. It is a simple string but it changes each time it is put in. So, it will generate a random string with salt and the hash part as well. Now, these two are unique (hash & salt output). Now, for the second user, the hash will give the same output as per the password, but the salt would generate another random string.

Lets say

```
const email = "abc@123.com"
const password = "123"
```

In real world, we will be getting this either from user form or req.body

We first generate a salt and it would be handled by the library. NodeJS has a predefined module to generate hash passwords. (bcrypt). We would import it,

```
const bcrypt = require('crypto')
```

'crypto' in itself has lot of function and hashing algorithms already written. So bcrypt uses all this internally and gives us better API's.

npm i bcrypt

Once this is imported, we will generate a salt. There is a function called, bcrypt.genSalt, which would generate a random salt.

```
const salt = bcrypt.genSalt(10)
```

Once the salt is generated, we can say,

```
const hashedPassword = bcrypt.hash()
```

This accepts two parameters. First parameter will be the password or content that we want to hash and the second parameter would be what salt it should use.

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

app.use(express.urlencoded({extended: false}))

app.get('/', async (req, res) => {

    const email = "abc@123.com"
    const password = "123"

    const salt = await bcrypt.genSalt(10)
    const hashedPassword = await bcrypt.hash(password, salt)

    console.log(salt)
    console.log(hashedPassword)

    res.send("Welcome to Yelpcamp")

})

app.listen(3000, () => {console.log('Server Initiated')})
```

In the output we can see the salt and a random password being generated. If we add another password to,

```
const email = "abc@123.com"
const pass1 = "123"
const pass2 = "123"

const salt1 = await bcrypt.genSalt(10)
const hashedPassword1 = await bcrypt.hash(pass1, salt1)

console.log(salt1)
console.log(hashedPassword1)

const salt2 = await bcrypt.genSalt(10)
const hashedPassword2 = await bcrypt.hash(pass2, salt2)

console.log(salt2)
console.log(hashedPassword2)
```

We can see that the hash part is same for both passwords, but the salt is different. Every time we refresh, bcrypt would generate a different salt. We will be storing the hash password in our database along with the salt that has been added to it. This how we generate the password using hash and salt.

How to compare?

We will try to get the user from the database base on email id that we get.

```
const user = {
    email: "abc@123.com",
    password = hashedPassword1
}
```

We have put the hashedPassword1 as we will be comparing it. this value will be coming from the database.

```
const isMatching = await bcrypt.compare()
```

We can see the compare(). The first parameter will be the password that the user has typed-in the form and sent it. So, we can wirte req.body.password and stimulating it with password. And the second parameter would be the password that is coming from the database.

```
const isMatching = await bcrypt.compare(password, user.password)
```

We have the email and password. The given password would be compared by the password that was already present in the database.

```
app.get('/', async (req, res) => {

    const email = "abc@123.com"
    const pass1 = "123"

    const salt1 = await bcrypt.genSalt(10)
    const hashedPassword1 = await bcrypt.hash(pass1, salt1)

    const user = {
        email: "abc@123.com",
        password = hashedPassword1
    }

    const isMatching = await bcrypt.compare(password, user.password)
    console.log(isMatching)

    res.send("Welcome to Yelpcamp")

})
```

Even if we change a single character, it would show **false** in the terminal. We are not going to pass salt in the compare(), because the salt is added to the hashedPassword.

Seeding:

As we are developing a project, we have a lot of functions to be tested. We cannot do everything manually. But we want the correct data in our database so that if we try to pick a bug we have sample data available to us. this is what we call seeding.

When we are developing from a clean slate in the database section, then database should be clean that is dummy data only, but it should be right format. So we will create a javascript file (seed.js) we will create a function and using whatever models we have,

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

function seedDB() {
    UserModel.remove({})
}
```

This will remove everything in our database under user model. We have a sample data as json or we can write some libraries to generate a random string. Let's say we have a sample user object.

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


async function seedDB() {
    UserModel.remove({})

    //
    const userArr = []
    for (let index = 0; index < 10; index++) {

        const user = {
            email: `abc${index++}@abc.com`,
            password: "123"
        }

        URLSearchParams.push(user)
    }

    const result = await UserModel.inserMany(user)

    console.log(result.length)

}

module.exports = { seedDB }
```

We can then import seedDB into our index.js file.

User.Model.remove({}), will remove all users and the function seedDB would create a list of sample data in the database. Once our database is clean, then we will comment the seedDB in the index.js file and we can do our testing. Again, if we want to do something else then would uncomment it and create a sample data again.

This is for user model and as many models we have, we can create it. We have multiple functions for different models. This eases up the development process.

Git Pull:

When we do pull request, we need to give a **proper pull request message**, because other's working on the same would be able to understand.

This also follows for commit messages as well.

Based on the changes, there will be merge conflicts i.e., two people made the same changes in the same line. Then Git would generate a merge conflict.

In daily practice, we have to keep updating the branches or code, in-sync with the main branch. While we are making changes in our branch, there were changes being made to the main branch as well. Let say, we have cloned from the second commit and while we are working, the main branch got another commit (third one). Now, if we create pull request, it will git will compare the third commit of the main branch with the first clone of the side branch.

Upstream:

There is an option called Fetch Upstream. Whenever we are staring new, we have to pull from the upstream. In our terminal if we enter

```
git remote -v
```

the origin is the original one. we would also have upstream, which will have reference of the original project. Upstream will have reference of the original project. So before doing a pull request, we will type,

```
git pull upstream
```

If we don't want to do in the CMD, then in our account, if we click on Fetch upstream, it would entire the main branch and then we can create the pull request.