the Master Course

{CODENATION}

Backend Development

Hashing and Bcrypt

{CODENATION}

{CODENATION}

Learning Objectives

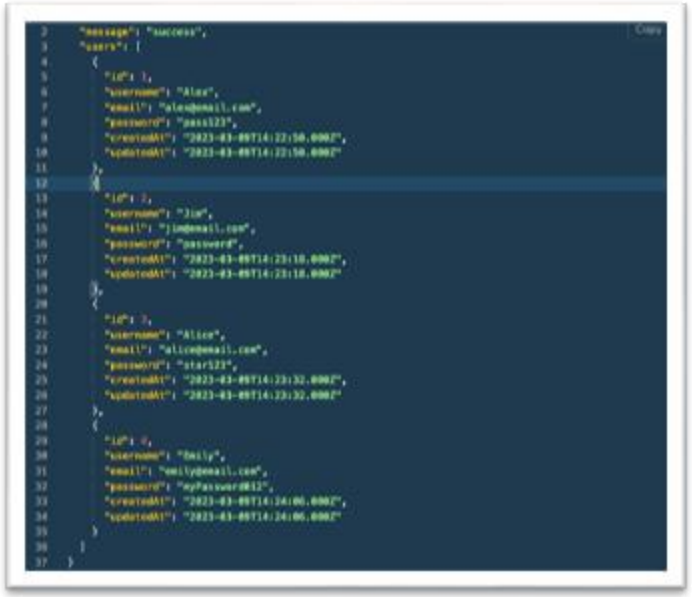
To understand the need for hashing database passwords

To use the ‘bcrypt’ NPM library to hash passwords

Let’s look at the entries we have stored in our user table.

HASHING

What if an attacker gained access to our database? Our user's plain text passwords are now exposed.



{CODENATION}

HASHING

How can we store passwords more securely in our database?

That’s where hashing comes into play

Password hashing is defined as putting a password through a hashing algorithm to turn a plaintext password into an unintelligible series of numbers and letters.

HASHING

hashing

```
-------
Plain Text Word - password
-------
Hashed Version - 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
-------
Plain Text Word - hello
-------
Hashed Version - 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
-------
Plain Text Word - random
-------
Hashed Version - a441b15fe9a3cf56661190a0b93b9dec7d04127288cc87250967cf3b52894d11
-------
```

security

hashed passwords

HASHING

Password hashing is a key step to protecting your users on the backend, but it’s not infallible because it hashes in a consistent way. This means it is predictable and can be beaten.

HASHING

Below is an example of a few words going through the hashing and salting process.

The first 22 characters is the salting which has been added to the start of each plain text password.

In the event of a security breach, compromised hashed passwords are unintelligible to an attacker. As a result, the theft of this information is considerably more difficult.

How do we make this even more secure ?

We can also specify the number of times our plain text passwords go through the salting and hashing process, this is known as salt rounds or the cost factor

The higher the number of salt rounds: the more secure the hash will be.

However, there is a trade off …

HASHING

```
Plain text password - password
number of salt rounds 10, time to hash: 81.25ms
hashed and salted password - Xr7z4KZewc3bLChhgEM1VeHtVAxgGWqYD23knY9/hGq748eERZVza
Plain text password - password
number of salt rounds 11, time to hash: 152.886ms
hashed and salted password - ubcirPOTR/PAhCd1/AhuJ.ZowOrNCxmNogf2RyB9fX.shM.138Pl.
Plain text password - password
number of salt rounds 12, time to hash: 283.777ms
hashed and salted password - gg3dhNambokowGEjkhTXvOQVdbSRwY1wohb3WnVibhl7mBw3g6TFy
Plain text password - password
number of salt rounds 13, time to hash: 549.329ms
hashed and salted password - 6nbUskpFbxeJbo06Iada7uRfdDei8TmpAfonwzCdNEq3w6nzzfm3y
Plain text password - password
number of salt rounds 14, time to hash: 1.148s
hashed and salted password - UYvTDkxyQlpdCmwFmChl/.BSRvxzNUMlhUVNf/WbS.mmUIHS4HeNa
Plain text password - password
number of salt rounds 15, time to hash: 2.886s
hashed and salted password - 0UKrltny6mI5CEQnTbPdou.0V3XDGxXX1dFb1GVUh2hx9FrjXtxfS
Plain text password - password
number of salt rounds 16, time to hash: 5.177s
hashed and salted password - 2JqpWBvFKEtVhSN3DxqYreJ1yTPeVBEDA6KCBZwwTYjNGu.Mt8cA.
Plain text password - password
number of salt rounds 17, time to hash: 9.217s
hashed and salted password - szY17yV5Mdu0mQVP6z/QSe0kttGwZ0C2P1BR91yJBMYp6G4wvDi66
Plain text password - password
number of salt rounds 18, time to hash: 19.612s
hashed and salted password - wlE2br2oZYw9O7MwfBWt9eoVripPglHXJ49/Gyh6/n2drP0I1jaOS
Plain text password - password
number of salt rounds 19, time to hash: 38.436s
hashed and salted password - VJCLYxr5Lw5bRqc/Q7rwPeRRGbHexkBolm2siqi4btuACIBSgORpq
Plain text password - password
number of salt rounds 20, time to hash: 1:15.315 (m:ss.mmm)
hashed and salted password - ugi7Bq66djY.fIvIbLStyedo6nMpyeUV9SXd0NovH1WB0G/mjSTLy
```

The more salt rounds number also effects how long the hash and salting process takes to complete. The output (left) shows the time it took to hash and salt a plain text password 10 times. With the number of salt rounds starting at 10 and increasing by 1 until 20 is reached.

As you can see the processing time increases as the number of salt rounds does. That's why it's important when deciding on the number of salt rounds to find the right balance between security and usability. Increasing the number of salt rounds increases computation time.

HASHING

As a common rule of thumb for deciding on the number of salt rounds, is to tune the cost so that the hashing process runs as slow as possible without affecting the users' experience.

What is bcrypt?

It is an NPM library to help you to hash passwords.

It uses a password-hashing function that is based on the Blowfish cipher.

HASHING

First, let's install bcrypt …

```
admin2@CN18s-MBP hashing-example % npm i bcrypt
```

Once installed, bcrypt should appear as a dependency in the project package.json file

```
  "dependencies": {
    "bcrypt": "^5.1.0",
    "cors": "^2.8.5",
    "dotenv": "^16.0.3",
    "express": "^4.18.2",
    "jsonwebtoken": "^9.0.0",
    "mysql2": "^3.2.0",
    "nodemon": "^2.0.21",
    "sequelize": "^6.29.0"
  }
}
```

Now, we need to import bcrypt into the specific files where needed

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

bcrypt .hash() Method

This method is used to hash and salt plain text passwords asynchronously.

HASHING

Here we have an asynchronous function that hashes the value of the plainTextPassword variable using the bcrypt .hash() method.

.hash() takes two values:

1. A password string to hash.
2. The number of salt rounds to use, e.g., 10

Once our hashPass function is called, the plain text password is hashed, salted then stored in the hash variable.

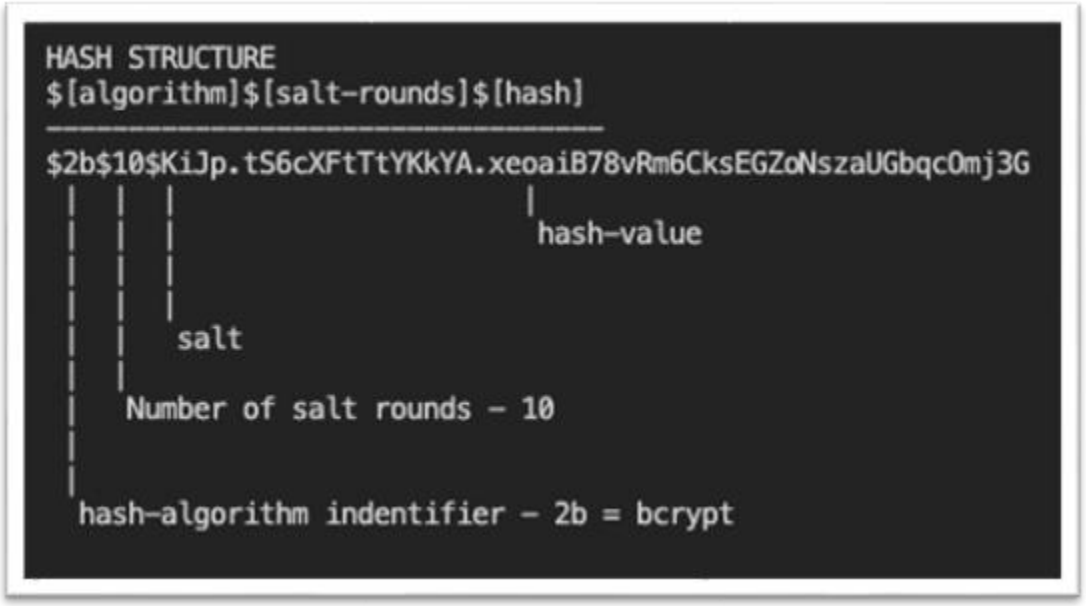
The resulting hash is then logged in the terminal.

```
const plainTextPassword = "DFGh5546*%^__90";
const saltRounds = 10

const hashPass = async () => {
  let hash = await bcrypt.hash(plainTextPassword, saltRounds)
  console.log(hash)
}

hashPass()
```

```
admin2@UNKNOWN salt-rounds % node index.js
Hashed Password - $2b$10$xQDFh2HweU.CaY5HRiAl9e58Jwi6Wj8hDoZLjziuB80wdena9sBTK
```

HASHING

Resultant hashes will be 60 characters long and, as follows.

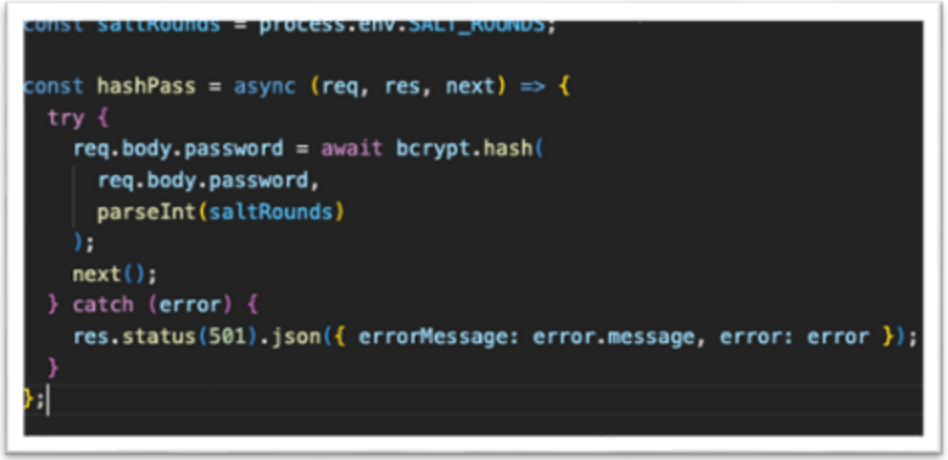


They will include the salt and other parameters

HASHING

Here is another example of the .hash() method being used in a hash password middleware function, in a REST API.

This time the plain text password that is passed in the body of the request is overwritten with the hashed version.



```
const saltRounds = process.env.SALT_ROUNDS;

const hashPass = async (req, res, next) => {
  try {
    req.body.password = await bcrypt.hash(
      req.body.password,
      parseInt(saltRounds)
    );
    next();
  } catch (error) {
    res.status(501).json({ errorMessage: error.message, error: error });
  }
};
```

The hash.() method is called and the plain text password stored in the body of the request along with the salt rounds, which are loaded from the environment variables and converted into a number are passed to it. next() is then called once the hash has been successful.

bcrypt .compare() Method

This method is used to compare a plain text password with a hashed version.

HASHING

Here we have an asynchronous function that compares the value of a previously hashed password with the plain text version, using the bcrypt .compare() method.

```
const plainTextPassword = "DFGh5546*%^__90";
const hashedPassword = "$2b$10$6cR7I/9l4pFvTVNiGRHzZu0Li2k8GV5LRlb6zlWDRYwFKZqAVokC."
const otherPassword = "hello"
const compareHash = async () => {
    let match = await bcrypt.compare(plainTextPassword, hashedPassword)
    // match = true
    if (match) {
      console.log("It matches!")
    } else {
      console.log("Invalid password!");
    }
}
compareHash()
```

.compare() takes two values:
1. the plain text password.
2. the hashed version of the plain text password.

Once the compareHash function is called, the plain text password is compared with the hashed version. If they match the method will return true.

```
admin2@CN18s-MBP salt-rounds % node index.js
It matches!
```

Here is the same function again, but this time the otherPassword is compared with the original hashedPassword.

```
const plainTextPassword = "DFGh5546*%^__90";
const hashedPassword = "$2b$10$6cR7I/9l4pFvTVNiGRHzZu0Li2k8GV5LRlb6zlWDRYwFKZqAVokC.'
const otherPassword = "hello"
const compareHash = async () => {
    let match = await bcrypt.compare(otherPassword, hashedPassword)
    // match = false
    if (match) {
      console.log("It matches!")
    } else {
      console.log("Invalid password!");
    }
}
compareHash()
```

These don't match so our .compare() method returns false

```
admin2@UNKNOWN salt-rounds % node index.js
Invalid password!
```

.compare() method

First the user's credentials are found in the database using the username passed in the body of the request as a filter. We now have access to the users credentials, stored in an object called user, that we add to the request body

Compare the plain text password with the hashed version.

If the passwords match,then next() is then called and the login process can continue. Otherwise report an error.

```
const comparePass = async (req, res, next) => {
  try {
    // find user
    req.user = await User.findOne({ where: { username: req.body.username } });

    // req.body.password = plain text password
    // req.user.password = hashed password stored in the database
    const match = await bcrypt.compare(req.body.password, req.user.password);

    // if no match - respond with 500 error message "passwords do not match"
    if (!match) {
      const error = new Error("Passwords do not match");
      res.status(500).json({ errorMessage: error.message, error: error });
    }

    // if match - next function
    next();
  } catch (error) {
    res.status(501).json({ errorMessage: error.message, error: error });
  }
};
```

{CODENATION}

Learning Objectives

To understand the need for hashing database passwords

To use the ‘bcrypt’ NPM library to hash passwords