bcrypt is a widely used library in Node.js for hashing passwords securely. It incorporates a salting mechanism to protect against dictionary attacks and brute force attacks, making it a strong choice for storing sensitive data like user passwords.

**Key Concepts**

1. **Hashing**:
   * Converts plain text data into an irreversible fixed-length string.
   * Ensures that stored passwords cannot be reversed to their original form.
2. **Salting**:
   * Adds a random string (the "salt") to the password before hashing.
   * Prevents identical passwords from generating the same hash, protecting against rainbow table attacks.
3. **Cost Factor**:
   * Determines how computationally expensive the hashing process is (default: 10).
   * Higher cost factor increases security but takes more time to compute.

**Installation**

Install bcrypt using npm:

npm install bcrypt

**Basic Usage**

**1. Hashing a Password**

import bcrypt from 'bcrypt';

const hashPassword = async (password) => {

    const saltRounds = 10; // Default cost factor

    const hashedPassword = await bcrypt.hash(password, saltRounds);

    console.log('Hashed Password:', hashedPassword);

    return hashedPassword;

};

hashPassword('mySecurePassword');

* **bcrypt.hash**:
  + Asynchronously generates a hash for the given password.
  + Takes the password and the number of salt rounds as arguments.

**2. Comparing Passwords**

const comparePasswords = async (password, hashedPassword) => {

    const isMatch = await bcrypt.compare(password, hashedPassword);

    console.log('Passwords Match:', isMatch);

    return isMatch;

};

const hashedPassword = await hashPassword('mySecurePassword');

comparePasswords('mySecurePassword', hashedPassword); // true

comparePasswords('wrongPassword', hashedPassword);    // false

* **bcrypt.compare**:
  + Asynchronously compares a plain text password with a hashed password.
  + Returns true if they match, false otherwise.

**3. Generating a Salt Manually**

While bcrypt.hash automatically generates a salt internally, you can generate one manually:

const manualHashing = async (password) => {

    const saltRounds = 10;

    const salt = await bcrypt.genSalt(saltRounds);

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

    console.log('Salt:', salt);

    console.log('Hashed Password:', hashedPassword);

};

manualHashing('mySecurePassword');

* **bcrypt.genSalt**:
  + Generates a salt value based on the given number of rounds.

**Best Practices**

1. **Use Async Functions**:
   * Always prefer the asynchronous methods (hash, compare, genSalt) to avoid blocking the event loop.
2. **Reasonable Cost Factor**:
   * Start with 10 as the cost factor, and adjust based on performance benchmarks for your application.
   * Higher cost = slower hash generation, but more secure.
3. **Never Store Plain Text Passwords**:
   * Only store the hashed version in your database.
4. **Validate User Input**:
   * Ensure the password meets complexity requirements before hashing (e.g., length, special characters).

**Real-World Example**

Here’s a simple example of how bcrypt might be used in a user registration and login flow:

**User Registration**

import bcrypt from 'bcrypt'

userSchema.pre('save',

    async function(next){

        if (!this.isModified('password')) return

        this.password = await bcrypt.hash(this.password,8)

        next()

    }

)

securely hash a user's password before saving it to the database. Let’s break it down step by step:

**Key Components**

1. **userSchema.pre**:
   * A Mongoose middleware hook that executes before a specific operation (in this case, the save operation).
   * Used here to hash the password before saving the user document.
2. **bcrypt.hash**:
   * Hashes the password with a cost factor of 8 (indicating the number of salt rounds).
   * Ensures the password is securely stored in the database.
3. **isModified('password')**:
   * Checks if the password field has been modified.
   * Prevents re-hashing the password if it's not modified (e.g., when updating other fields of the user).

**Best Practices**

1. **Choose an Appropriate Salt Rounds Value**:
   * Start with 8-10 for a good balance of security and performance.
2. **Avoid Plain Text Passwords in Logs**:
   * Never log the plain text password or even the hash.
3. **Always Use Secure Storage**:
   * Store the hashed password in the database, never the plain text password.

**Summary**

This code snippet demonstrates a robust and secure way to handle passwords using **bcrypt** in a **Mongoose model**:

* Hashes the password before saving.
* Avoids re-hashing if the password hasn't been modified.
* Provides a reusable method to verify passwords during authentication.