Semester	T.E. Semester VI – Computer Engineering
Subject	Cryptography and cyber security
Subject Professor In-charge	Prof. Amit Nerurkar
Assisting Teachers	Prof. Amit Nerurkar
Laboratory	M312B

Topic	Mini Project Report
Group Members	Omkar Patil 21102A0003 Pranav Redij 21102A0005 Sahil Pokharkar 21102A0006 Deep Salunkhe 21102A0014
TE Division	A

Title: Hasher(Hashing library)

Implementation:

encrypt.js

```
function encrypt(message) {
    // Constants defined by the SHA-256 algorithm
    const K = [
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3,
```

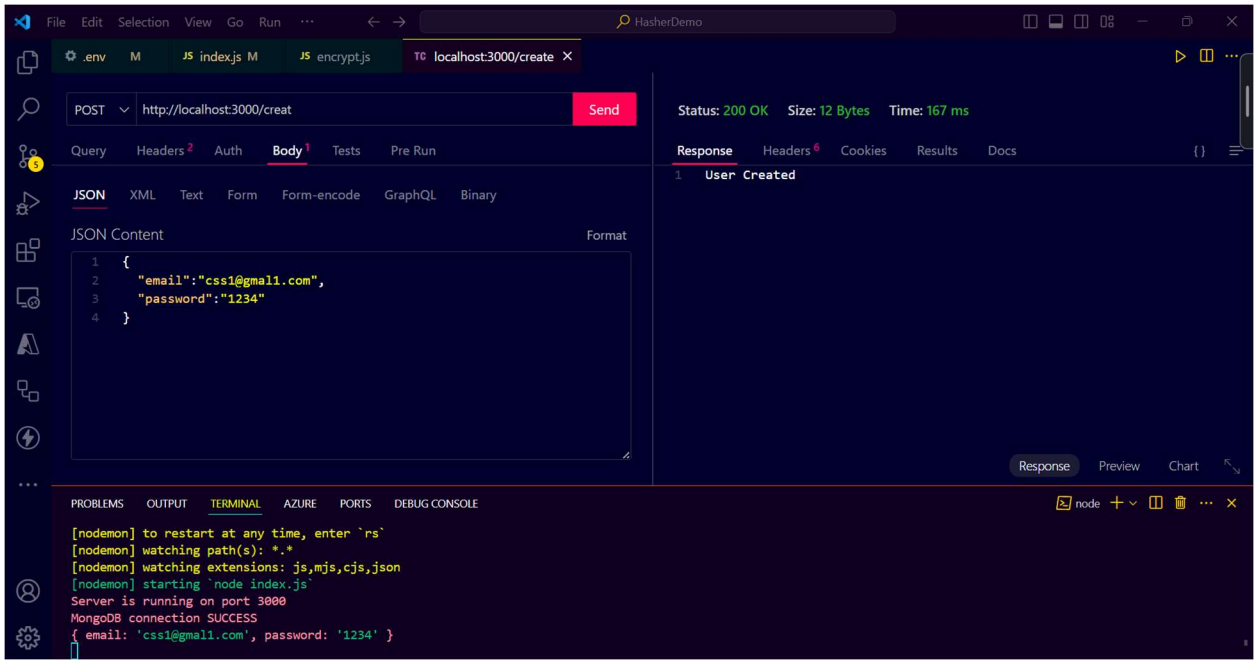
Title: Mini Project

```
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
0xbef9a3f7, 0xc67178f2
    ];

    // Initial hash values (first 32 bits of the fractional parts of the square
roots of the first 8 primes)
    const H = [
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c,
0x1f83d9ab, 0x5be0cd19
    ];

    // Helper functions

    // Bitwise rotation functions
    //inputs: x: number, n: number
    //Tasks: Rotate x right by n bits
    function rotr(x, n) { return (x >>> n) | (x << (32 - n)); }

    // Bitwise logical functions
    //inputs: x: number, n: number
    //Tasks: Shift x right by n bits
    function shr(x, n) { return x >>> n; }

    // SHA-256 functions

    //These are the function which are used in each round of the SHA-256
algorithm
    function sig0(x) { return rotr(x, 2) ^ rotr(x, 13) ^ rotr(x, 22); }
    function sig1(x) { return rotr(x, 6) ^ rotr(x, 11) ^ rotr(x, 25); }
    function S0(x) { return rotr(x, 7) ^ rotr(x, 18) ^ shr(x, 3); }
    function S1(x) { return rotr(x, 17) ^ rotr(x, 19) ^ shr(x, 10); }

    // Pre-processing
    // Convert message to array of 64-byte chunks
    //why: The message is divided into 64-byte(512 bits) chunks, and each chunk
is further divided into 4-byte words
    let m = [];
    for (let i = 0; i < message.length; i += 64) {
        let chunk = message.slice(i, i + 64);
        m.push(new Uint8Array([...chunk].map(c => c.charCodeAt(0))));
    }

    // Padding
```

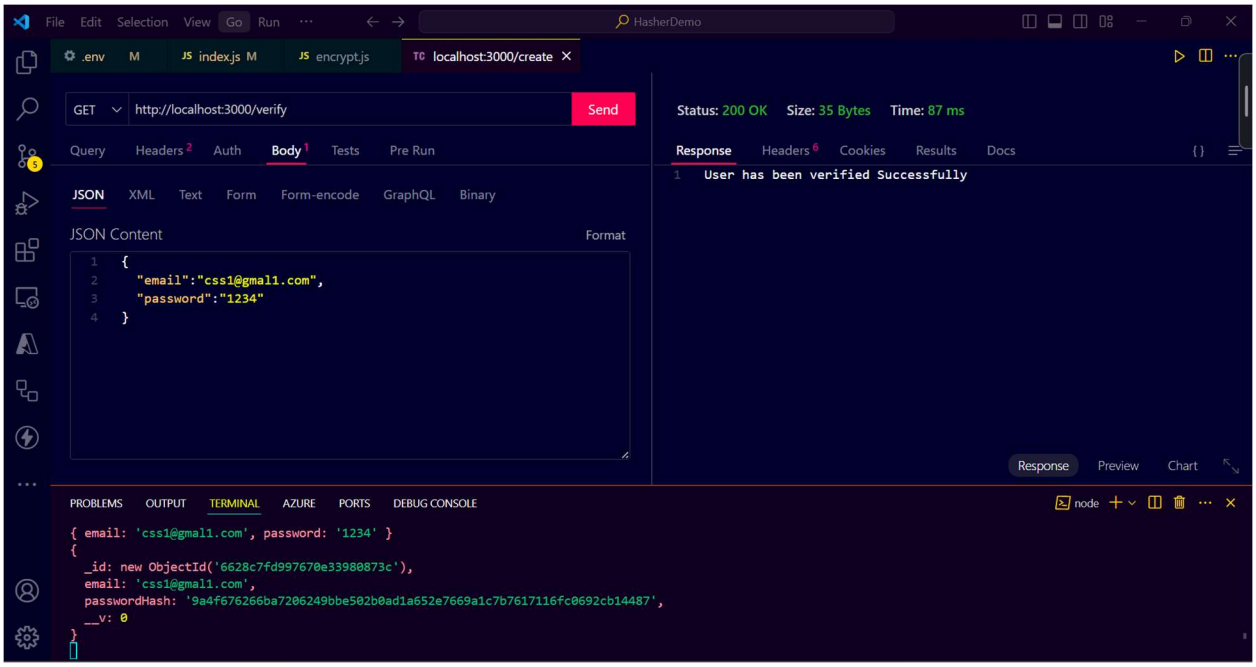
```
    m.push(new Uint8Array([0x80]));
    while (m.length % 64 !== 56) {
        m.push(new Uint8Array([0x00]));
    }
    const originalLength = message.length * 8;
    const lengthBytes = new ArrayBuffer(8);
    const dv = new DataView(lengthBytes);
    dv.setBigUint64(0, BigInt(originalLength));
    const lengthBytesArray = new Uint8Array(lengthBytes);
    m.push(lengthBytesArray);

    // Process each 512-bit block
    for (let i = 0; i < m.length; i += 64) {
        //w is an array of 64 32-bit words
        const w = new Uint32Array(64);

        // Prepare message schedule
        for (let t = 0; t < 16; t++) {
            w[t] = (m[i / 4][t * 4] << 24) | (m[i / 4][t * 4 + 1] << 16) | (m[i /
4][t * 4 + 2] << 8) | m[i / 4][t * 4 + 3];
        }
        for (let t = 16; t < 64; t++) {
            w[t] = (S1(w[t - 2]) + w[t - 7] + S0(w[t - 15]) + w[t - 16]) >>> 0;
        }

        // Initialize hash value for this chunk
        let [a, b, c, d, e, f, g, h] = H;

        // Main loop
        for (let t = 0; t < 64; t++) {
            const T1 = (h + sig1(e) + ((e & f) ^ (~e & g)) + K[t] + w[t]) >>> 0;
            const T2 = (sig0(a) + ((a & b) ^ (a & c) ^ (b & c))) >>> 0;

            h = g;
            g = f;
            f = e;
            e = (d + T1) >>> 0;
            d = c;
            c = b;
            b = a;
            a = (T1 + T2) >>> 0;
        }
```

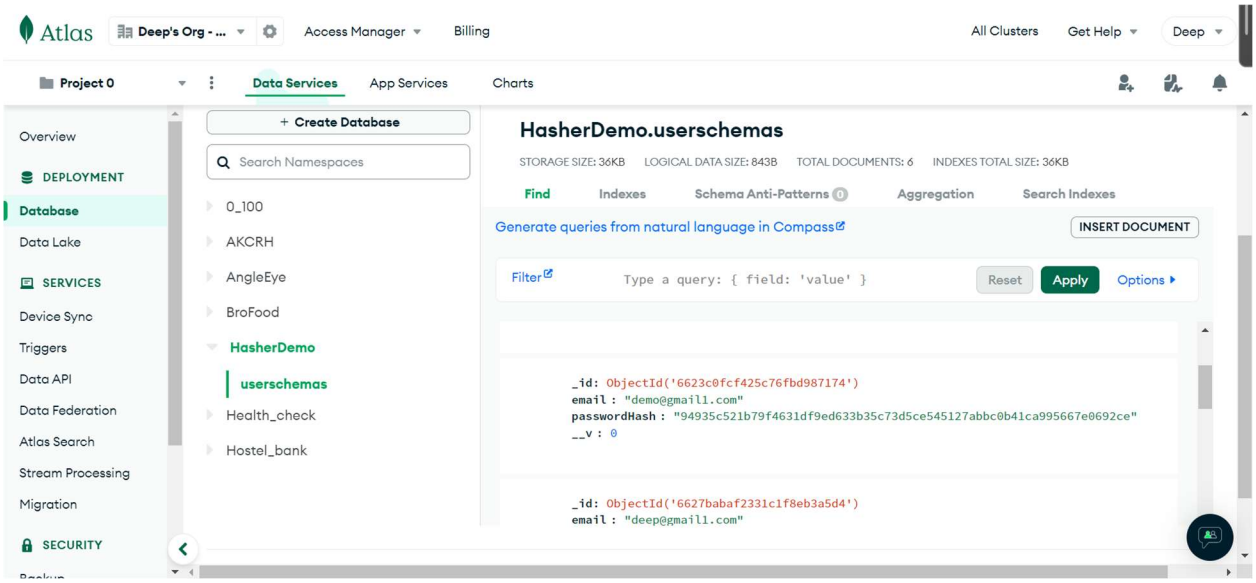
```
        // Update hash values
        H[0] = (H[0] + a) >>> 0;
        H[1] = (H[1] + b) >>> 0;
        H[2] = (H[2] + c) >>> 0;
        H[3] = (H[3] + d) >>> 0;
        H[4] = (H[4] + e) >>> 0;
        H[5] = (H[5] + f) >>> 0;
        H[6] = (H[6] + g) >>> 0;
        H[7] = (H[7] + h) >>> 0;
    }

    // Convert hash values to hexadecimal string
    let hashHex = '';
    for (let i = 0; i < H.length; i++) {
        hashHex += ('00000000' + H[i].toString(16)).slice(-8);
    }

    return hashHex;
}

module.exports={encrypt};
```

Output:



Title: Mini Project





Steps:

1. **Message Padding**: If the message's length is not already a multiple of the block size (512 bits), padding is applied to ensure it is. The padding typically includes a single '1' bit followed by the necessary number of '0' bits, and finally, the length of the original message in binary representation.
2. **Message Parsing**: The padded message is divided into blocks of 512 bits each.

Title: Mini Project

3. **Initialization**: SHA-256 uses eight initial hash values (often called "constants" or "IVs"). These values are derived from the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. These initial hash values are often represented in hexadecimal as:

Copy code

6a09e667, bb67ae85, 3c6ef372, a54ff53a, 510e527f, 9b05688c, 1f83d9ab, 5be0cd19

4. **Message Schedule**: For each 512-bit block, a message schedule of 64 32-bit words (W[0], W[1], ..., W[63]) is created.
5. **Compression Function**: The compression function takes the current hash value (initialized hash values in the first round) and the message schedule as inputs and produces a new hash value. This process involves 64 iterations (called rounds) of mixing and transforming the current hash value.
6. **Finalization**: After processing all blocks, the final hash value is obtained by concatenating the hash values produced from each block.
7. **Output**: The final hash value (256 bits) represents the SHA-256 digest of the input message.

Hosted Link:

https://www.npmjs.com/package/@deepsalunkhee/hasher

Conclusion:
In the realm of modern cryptography, SHA-256 emerges as a stalwart guardian, providing robust security, integrity, and trust in digital communications and transactions. Through its intricate design and rigorous security properties, SHA-256 exemplifies the pinnacle of cryptographic hash functions, offering a reliable foundation for a myriad of applications spanning from password hashing to blockchain technology.

As we navigate the complexities of an increasingly digitized world, SHA-256 stands as a beacon of assurance, safeguarding against the ever-looming threats of unauthorized access, data tampering, and forgery. Its steadfast resilience and unwavering cryptographic strength underscore its significance as a cornerstone of modern cybersecurity practices.

While SHA-256 continues to play a pivotal role in securing our digital infrastructure, the landscape of cryptography remains dynamic, with new challenges and opportunities on the horizon. As researchers and practitioners explore emerging trends and technologies, the legacy of SHA-256 endures, serving as a testament to the enduring power of cryptographic principles in preserving the confidentiality, integrity, and authenticity of our digital interactions.

In essence, SHA-256 embodies the essence of trust in the digital age, forging a path towards a more secure and resilient future where data privacy and security are paramount. As we reflect on the journey thus far, we can take solace in the knowledge that SHA-256 stands as a steadfast ally in the ongoing quest for cryptographic excellence and digital empowerment.