NAME: GAURAV GAUR

REG NO: 20BCE0774

Symmetric algorithm:

AES

AES (Advanced Encryption Standard) is a widely used symmetric encryption algorithm that operates on blocks of data. It is known for its security, efficiency, and compatibility across various systems.

Here's a brief explanation of how the AES algorithm works:

1. Key Expansion: AES requires a secret key to encrypt and decrypt data. The key expansion process generates a set of round keys derived from the original secret key.

2. Initial Round: AES operates on blocks of data, typically 128 bits (16 bytes) in size. In the initial round, the input block is combined with the first round key using a process called "AddRoundKey."

3. Rounds: AES employs a series of rounds (typically 10, 12, or 14 rounds depending on the key size) to transform the data block. Each round consists of four main steps applied to the data:

a. SubBytes: Each byte in the data block is substituted with a corresponding byte from the AES S-box, a predefined substitution table.

b. ShiftRows: The bytes in each row of the data block are shifted cyclically to the left. This step provides diffusion in the data.

c. MixColumns: Each column of the data block is transformed by combining its four bytes using a matrix multiplication operation. This step introduces diffusion across columns.

d. AddRoundKey: The round key for the current round is combined with the transformed data block using a bitwise XOR operation.

4. Final Round: The final round is similar to the earlier rounds, but it lacks the MixColumns step.

5. Output: After completing all the rounds, the resulting data block is the encrypted (or decrypted) ciphertext.

The AES algorithm is designed to be resistant to various cryptographic attacks, including differential and linear cryptanalysis. It ensures the security of the encrypted data by incorporating key-dependent operations and complex mathematical transformations.

Advantages of AES:

1. Strong Security: AES is considered secure against all known practical attacks when used properly. It has a high level of resistance against various cryptographic attacks, such as brute force, differential cryptanalysis, and linear cryptanalysis.

2. Wide Adoption: AES has been adopted as the standard encryption algorithm by governments, organizations, and industries worldwide. It is used extensively in various applications, including data encryption, secure communication protocols, virtual private networks (VPNs), and more.

3. Efficiency: AES is computationally efficient, allowing for fast encryption and decryption of data. It is optimized for modern computer architectures and can be implemented efficiently in hardware and software, making it suitable for a wide range of devices and platforms.

4. Scalability: AES supports key sizes of 128, 192, and 256 bits, providing flexibility and scalability based on the required level of security. The larger the key size, the stronger the encryption.

5. Transparency: AES is an open standard, meaning its specification and implementation details are publicly available. This openness allows for extensive analysis and scrutiny by cryptographic experts, enhancing confidence in its security.

Disadvantages of AES:

1. Symmetric Encryption: AES is a symmetric encryption algorithm, which means the same key is used for both encryption and decryption. This requires the secure distribution of the encryption key to all parties involved, which can be challenging in some scenarios.

2. Key Management: The strength of AES relies heavily on the security and management of encryption keys. The generation, storage, distribution, and revocation of keys must be carefully handled to maintain the overall security of the system.

3. Limited to Encryption: AES is primarily designed for encryption purposes and does not provide built-in functionality for other cryptographic operations like digital signatures or key exchange. Additional algorithms or protocols are needed to complement AES for these purposes.

4. Vulnerabilities in Implementation: While AES itself is considered secure, vulnerabilities can arise from flaws in the implementation or surrounding systems. Side-channel attacks, such as timing attacks or power analysis, can exploit implementation weaknesses rather than targeting the algorithm itself.

5. Quantum Computing Threat: The advent of quantum computers could potentially pose a threat to AES and other symmetric encryption algorithms. Quantum computers have the potential to break current cryptographic algorithms, including AES, using algorithms like Shor's algorithm. However, this threat is not imminent and can be mitigated by using quantum-resistant algorithms.

Here are some examples of real-world usage of AES:

1. Secure Communication: AES is commonly used to secure communication channels, such as in SSL/TLS protocols used for secure web browsing (HTTPS). When you visit a website with a padlock icon in the address bar, AES encryption is likely being used to protect the data transmitted between your browser and the website's server.

2. File and Disk Encryption: AES is often employed to encrypt files and entire disk drives to protect sensitive data. Operating systems like Windows and macOS have built-in support for AES encryption, allowing users to encrypt their files or create encrypted disk images for data storage.

3. Wireless Networks: AES is used in various wireless network security protocols, such as WPA2 (Wi-Fi Protected Access 2), which provides encryption for securing Wi-Fi connections. It ensures that data transmitted over the wireless network is encrypted using AES, protecting it from eavesdropping and unauthorized access.

4. Virtual Private Networks (VPNs): AES is commonly used in VPNs to create secure tunnels for remote access and private communication. VPN protocols like OpenVPN, IPSec, and WireGuard utilize AES encryption to secure the data transmitted between the VPN client and server.

5. Secure Messaging Applications: Many messaging applications, such as Signal and WhatsApp, use AES encryption to secure messages and attachments exchanged between users. AES ensures that the content of the messages remains confidential even if intercepted by unauthorized parties.

6. Financial Transactions: AES encryption is employed in various financial systems and payment gateways to protect sensitive information during online transactions. It ensures the confidentiality of credit card numbers, banking details, and other financial data transmitted over networks.

7. Cloud Storage and Backup: AES encryption is used in cloud storage and backup services to protect data stored on remote servers. Files uploaded to services like Dropbox, Google Drive, and iCloud are often encrypted using AES to prevent unauthorized access.

Asymmetric key algorithms

RSA

The RSA (Rivest-Shamir-Adleman) algorithm is a widely used asymmetric encryption algorithm that provides secure communication and digital signatures. It is based on the mathematical concepts of prime factorization and modular arithmetic. Here's a brief explanation of how the RSA algorithm works:

1. Key Generation:

- Select two large prime numbers, p and q.

- Compute their product, n = p * q. This is the modulus used in encryption and decryption.

- Calculate Euler's totient function, ϕ(n) = (p - 1) * (q - 1). This is used to compute the private and public keys.

- Choose an integer e, known as the public exponent, such that 1 < e < ϕ(n) and e is coprime with ϕ(n).

- Compute the modular multiplicative inverse of e modulo ϕ(n). This inverse, d, is the private exponent and will be kept secret.

2. Encryption:

- To encrypt a message m, the sender converts it into a numerical representation.

- The sender uses the recipient's public key, (n, e), and applies the encryption formula: c = m^e mod n.

- The resulting ciphertext, c, is sent to the recipient.

3. Decryption:

- The recipient uses their private key, d, to decrypt the ciphertext.

- The recipient applies the decryption formula: m = c^d mod n.
- The resulting decrypted message, m, is recovered.

The security of the RSA algorithm relies on the difficulty of factoring large composite numbers into their prime factors. The private key, d, is required to decrypt the ciphertext, and it is computationally infeasible to compute d without knowing the prime factors of n. The larger the prime numbers used in the key generation, the more secure the RSA encryption becomes.

In addition to encryption, the RSA algorithm can also be used for digital signatures. The process involves signing a message using the sender's private key, and the recipient can verify the signature using the sender's public key.

The RSA (Rivest-Shamir-Adleman) algorithm has several key strengths and advantages that have contributed to its widespread usage in various cryptographic applications. Here are some of its key strengths:

1. Asymmetric Encryption: RSA is an asymmetric encryption algorithm, meaning it uses a pair of keys - a public key for encryption and a private key for decryption. This enables secure communication between parties without the need to share a secret key in advance. The public key can be freely shared, while the private key remains confidential.

2. Security: The security of RSA is based on the difficulty of factoring large composite numbers into their prime factors. The strength of RSA lies in the assumption that factoring large numbers into their prime factors is computationally infeasible within a reasonable time frame. As long as the key length is sufficiently large, RSA is considered secure against attacks using current computing resources.

3. Digital Signatures: RSA can be used for digital signatures, providing a means of verifying the authenticity and integrity of digital documents. The sender can sign a message using their private key, and the recipient can verify the signature using the corresponding public key. This ensures that the message has not been tampered with and comes from the expected sender.

4. Key Exchange: RSA can facilitate secure key exchange in symmetric encryption systems. Two parties can use RSA to securely exchange a shared secret key that can then be used for symmetric encryption, such as with AES. This allows for secure communication between parties without the need for pre-shared keys.

5. Wide Adoption: RSA has been widely adopted and standardized, making it compatible across various platforms, systems, and applications. It is supported by numerous cryptographic libraries and used in protocols such as SSL/TLS, S/MIME, and SSH.

6. Flexibility: RSA supports variable key sizes, allowing for flexibility and scalability based on the desired level of security. Larger key sizes provide stronger encryption but may require more computational resources. The choice of key size depends on the specific requirements of the application.

7. Openness and Accessibility: RSA is an openly published algorithm, making its specifications and implementations widely available for review and analysis by the cryptographic community. This transparency helps ensure its security and promotes the discovery and mitigation of potential vulnerabilities.

While RSA has notable strengths, it also has some limitations. The main drawback is its relatively slower computational speed compared to symmetric encryption algorithms, especially when encrypting large amounts of data. As a result, RSA is typically used in combination with symmetric encryption algorithms to achieve a balance of security and efficiency in practical applications.

While RSA (Rivest-Shamir-Adleman) is a widely used and secure encryption algorithm, it is not without its vulnerabilities and weaknesses. Some of the known vulnerabilities and weaknesses of the RSA algorithm include:

1. Key Length: The security of RSA is highly dependent on the length of the keys used. As computing power advances, longer key lengths are required to maintain the same level of security. Keys that are too short can be vulnerable to brute force attacks or advances in computational techniques, such as the development of quantum computers.

2. Factoring Attacks: The security of RSA is based on the difficulty of factoring large composite numbers. If a significant breakthrough in factoring algorithms or quantum computing occurs, it could potentially weaken the security of RSA. Shor's algorithm, a quantum algorithm, has the potential to efficiently factor large numbers, which would render RSA vulnerable.

3. Timing Attacks and Side Channels: RSA implementations can be vulnerable to side-channel attacks, such as timing attacks or power analysis. These attacks exploit information leaked through side channels, such as variations in execution time or power consumption, to gain insights into the secret key.

4. Random Number Generation: RSA relies on the generation of random numbers during key generation and encryption processes. Weak or predictable random number generation can undermine the security of RSA. Proper implementation and use of secure random number generators are essential to mitigate this vulnerability.

5. Padding Oracle Attacks: In some cases, RSA encryption may not provide proper padding, leading to vulnerabilities. Padding oracle attacks exploit vulnerabilities in the padding scheme, allowing an attacker to decrypt the ciphertext or recover parts of the plaintext without knowledge of the private key.

6. Implementation Flaws: Vulnerabilities can arise from flaws in the implementation of RSA or related cryptographic protocols. Programming errors or incorrect usage of RSA can lead to critical weaknesses that can be exploited by attackers.

It's important to note that while these vulnerabilities exist, their exploitation often requires significant computational resources, specialized knowledge, or breakthroughs in cryptographic techniques. The strength of RSA lies in its widespread adoption, extensive analysis, and continuous research to address potential vulnerabilities. Nonetheless, it's crucial to keep RSA implementations up to date and use appropriate key lengths to ensure the highest level of security.

The RSA (Rivest-Shamir-Adleman) algorithm is widely used in various real-world applications to ensure secure communication, digital signatures, and key exchange. Here are some common examples of where the RSA algorithm is commonly used:

1. Secure Communication: RSA is widely used in secure communication protocols such as SSL/TLS (Secure Sockets Layer/Transport Layer Security) used for secure web browsing (HTTPS). When you visit a website with a padlock icon in the address bar, RSA encryption is likely being used to establish a secure connection between your browser and the web server.

2. Digital Signatures: RSA is used for digital signatures to ensure the authenticity and integrity of digital documents. It is commonly employed in email encryption and signing standards like S/MIME (Secure/Multipurpose Internet Mail Extensions) to provide a means of verifying the identity of the sender and the integrity of the message.

3. Key Exchange: RSA is used in key exchange protocols, such as the Diffie-Hellman key exchange, to securely establish a shared secret key between two parties. This shared key can then be used for symmetric encryption to ensure secure communication.

4. Secure Shell (SSH): RSA is used in the SSH protocol for secure remote access to servers and network devices. RSA key pairs are generated and used for authentication, establishing secure connections, and protecting the integrity of data transferred between the client and server.

5. Virtual Private Networks (VPNs): RSA is often used in VPNs to establish secure tunnels for remote access and private communication. VPN protocols like OpenVPN and IPSec utilize RSA encryption for key exchange, authentication, and secure communication.

6. Certificate Authorities (CAs): RSA is commonly used in the infrastructure of Certificate Authorities, which issue digital certificates for website authentication. RSA keys are used in the creation and validation of digital certificates, providing the security infrastructure for secure web browsing.

7. Secure File Transfer: RSA encryption is used in various secure file transfer protocols and tools, such as Secure FTP (SFTP) and Pretty Good Privacy (PGP), to protect the confidentiality of transferred files and ensure secure data exchange.

These examples demonstrate the widespread adoption and versatility of RSA in securing communication, authenticating identities, and protecting data in various domains and applications. The RSA algorithm's robustness, compatibility, and established infrastructure make it a popular choice in many real-world scenarios where security is paramount.

HASH FUNCTIONS

SHA-256

SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function that generates a fixed-size output (256 bits) called a hash value or digest. Here's a brief explanation of how SHA-256 works:

1. Message Padding: SHA-256 operates on blocks of data. If the message is not already in a multiple of 512 bits, padding is added to the message. The padding ensures that the message length is a multiple of the block size.

2. Initialization: SHA-256 initializes a set of constants and initial hash values. These values are predefined and specific to the SHA-256 algorithm.

3. Breaking Message into Blocks: The padded message is divided into blocks of 512 bits each.

4. Message Schedule: For each block, the message schedule is generated. The message schedule consists of 64 32-bit words derived from the block and previous hash values.

5. Compression Function: The compression function takes the current block's message schedule and the previous hash value as inputs. It performs a series of logical and arithmetic operations, including bitwise logical functions (AND, OR, XOR), modular addition, and logical rotations.

6. Iteration: The compression function is applied iteratively for each block of the message, updating the hash value in each iteration.

7. Final Hash Value: After processing all the blocks, the resulting hash value is the final output of SHA-256. The hash value is a fixed-length representation of the input message, and even a small change in the input will produce a significantly different hash value.

The strength of SHA-256 lies in its properties, such as preimage resistance, collision resistance, and the avalanche effect. Preimage resistance ensures that it is computationally

infeasible to determine the original message from its hash value. Collision resistance means it is extremely unlikely to find two different messages that produce the same hash value. The avalanche effect implies that even a slight change in the input will result in a completely different hash value.

SHA-256 is widely used in various applications, including digital signatures, password storage, integrity verification, and blockchain technology, providing a secure and efficient means of verifying the integrity and authenticity of data.

The SHA-256 (Secure Hash Algorithm 256-bit) has several key strengths and advantages that make it a popular choice for cryptographic hash functions. Here are some of its key strengths:

1. Security: SHA-256 offers a high level of security. It is designed to be resistant to various cryptographic attacks, including preimage attacks, second preimage attacks, and collision attacks. These attacks aim to find vulnerabilities that would allow an attacker to derive the original message from its hash value or find two different messages with the same hash value. SHA-256's strength against these attacks has been extensively analyzed and tested.

2. Cryptographic Strength: SHA-256 is a 256-bit hash function, meaning it generates a fixed-size hash value of 256 bits. The large output size makes it computationally infeasible to reverse-engineer the original message or find another message with the same hash value. This provides a high level of assurance for data integrity and authenticity.

3. Efficiency: SHA-256 is computationally efficient, especially compared to other hash functions with longer output sizes. It can process large amounts of data quickly and efficiently, making it suitable for real-time applications, large-scale data processing, and resource-constrained environments.

4. Standardization: SHA-256 is a widely standardized algorithm, recognized and supported by various cryptographic libraries, systems, and protocols. Its widespread adoption ensures compatibility across different platforms and ensures interoperability among different systems and applications.

5. Broad Applicability: SHA-256 has a wide range of applications across various domains. It is commonly used for digital signatures, password storage (in hashed form), integrity

verification, certificate validation, blockchain technology (e.g., Bitcoin), and other security-sensitive applications where data integrity and authenticity are crucial.

6. Resistance to Collision Attacks: SHA-256 has a large hash output size, reducing the probability of collision attacks. A collision occurs when two different inputs produce the same hash value. The 256-bit output size of SHA-256 makes it highly unlikely to find two different messages with the same hash value, providing a strong defense against collision attacks.

7. Established and Trusted: SHA-256 is part of the SHA-2 family of hash functions, which have been extensively studied, analyzed, and tested by the cryptographic community. It has a long history of usage and has demonstrated robustness and reliability.

These strengths and advantages have contributed to the widespread adoption and trust in SHA-256 as a secure and efficient hash function for a variety of cryptographic applications. However, it's important to note that the security of SHA-256 relies on the proper implementation, key management, and overall system security practices in the specific context of its usage.

While SHA-256 (Secure Hash Algorithm 256-bit) is widely used and considered secure, it's important to be aware of potential vulnerabilities and weaknesses that have been identified. Here are some known vulnerabilities and weaknesses of the SHA-256 algorithm:

1. Collision Attacks: Although SHA-256 is resistant to collision attacks, it is not theoretically collision-proof. A collision occurs when two different inputs produce the same hash value. While the probability of finding a collision in SHA-256 is extremely low, it is not impossible. As computing power advances, the potential for finding collisions may increase.

2. Length Extension Attacks: SHA-256 is vulnerable to length extension attacks. In such attacks, an attacker who knows the hash of a message can append additional data to the original message without knowing the content of the message. This can lead to the creation of maliciously crafted messages with the same hash as the original message.

3. Quantum Cryptanalysis: While not specific to SHA-256 alone, it's worth mentioning that the emergence of quantum computers poses a potential threat to the security of all classical cryptographic algorithms, including SHA-256. Quantum computers, if developed with

sufficient computational power, could potentially break the underlying mathematical assumptions that provide the security of SHA-256.

4. Side-Channel Attacks: Side-channel attacks target vulnerabilities in the physical implementation of a cryptographic algorithm rather than its mathematical properties. Techniques such as timing analysis, power analysis, or electromagnetic analysis can potentially leak information about the internal operations of SHA-256 and compromise its security. Careful implementation and countermeasures are necessary to mitigate these attacks.

5. Limited Input Size: SHA-256 operates on fixed-size inputs, meaning that if the input message is longer than the block size (512 bits), it needs to be divided into blocks. This can introduce vulnerabilities if the padding and block division are not implemented correctly. It's important to follow recommended practices to ensure proper handling of input longer than the block size.

6. Dependence on Hash Strength: The strength of a hash function like SHA-256 relies on the entropy and randomness of the input message. If the input message has low entropy or is predictable, it can weaken the security guarantees provided by SHA-256.

7. Future Advances: Cryptographic algorithms, including SHA-256, must withstand future advances in computational power, algorithms, and attacks. Ongoing research and scrutiny are necessary to address any potential weaknesses that may arise due to advancements in technology or cryptanalysis.

It's important to note that while these vulnerabilities exist, most of them require significant computational power, specialized knowledge, or breakthroughs in cryptographic techniques to exploit. SHA-256 remains widely used and trusted in many practical applications. Nonetheless, it's essential to stay informed about the latest developments in cryptography and adopt recommended best practices for secure implementation and usage of SHA-256 and related algorithms.

The SHA-256 (Secure Hash Algorithm 256-bit) is widely used in various real-world applications where data integrity, security, and authenticity are critical. Here are some common examples of where the SHA-256 algorithm is commonly used:

1. Cryptocurrency: SHA-256 is a fundamental component of many cryptocurrencies, including Bitcoin. It is used in the process of mining, where miners compete to find a hash value that satisfies specific conditions. The SHA-256 hash function ensures the integrity and security of the blockchain, providing immutability to transactions and blocks.

2. Digital Signatures: SHA-256 is commonly used in digital signature algorithms such as ECDSA (Elliptic Curve Digital Signature Algorithm) and RSA (Rivest-Shamir-Adleman). Digital signatures provide a means of verifying the authenticity and integrity of digital documents, and SHA-256 is used to generate and verify the hash values that are used in the signature process.

3. SSL/TLS Certificates: SHA-256 is used in SSL/TLS certificates for secure communication over the web. Certificates use SHA-256 to generate and validate the certificate's digital signature, ensuring the authenticity and integrity of the certificate information and establishing secure connections between clients and servers.

4. File Integrity Checking: SHA-256 is commonly used to ensure the integrity of files during storage or transmission. By calculating the hash value of a file using SHA-256, one can verify whether the file has been altered or tampered with. Many software applications and tools use SHA-256 hashes to verify the integrity of downloaded files.

5. Password Storage: SHA-256 (sometimes in combination with salt) is used for secure password storage. Instead of storing passwords directly, systems store the hashed values of passwords. When a user logs in, the entered password is hashed with SHA-256, and the generated hash is compared with the stored hash for authentication.

6. Blockchain Applications: Beyond cryptocurrencies, SHA-256 is utilized in various blockchain applications. It ensures the integrity and security of data stored on the blockchain, and it is used in hashing transactions, blocks, and Merkle trees to maintain the integrity of the distributed ledger.

7. Digital Forensics: SHA-256 plays a vital role in digital forensics investigations. It is used to generate hash values for digital evidence, allowing investigators to verify the authenticity and integrity of data. Hash values generated using SHA-256 can be used to match and identify known files, detect duplicate files, or verify the integrity of evidence throughout the investigation process.

These examples highlight the versatility and widespread adoption of SHA-256 in various domains where data security, integrity, and authenticity are paramount. SHA-256's strong cryptographic properties and established infrastructure make it a reliable choice for ensuring the integrity and security of critical data and systems.

IMPLEMENTATION

I have implemented bcrypt for encrypting the password for login to save in the database. Here the password is encrypted using bcrypt and saved in the database.

Sign up code:

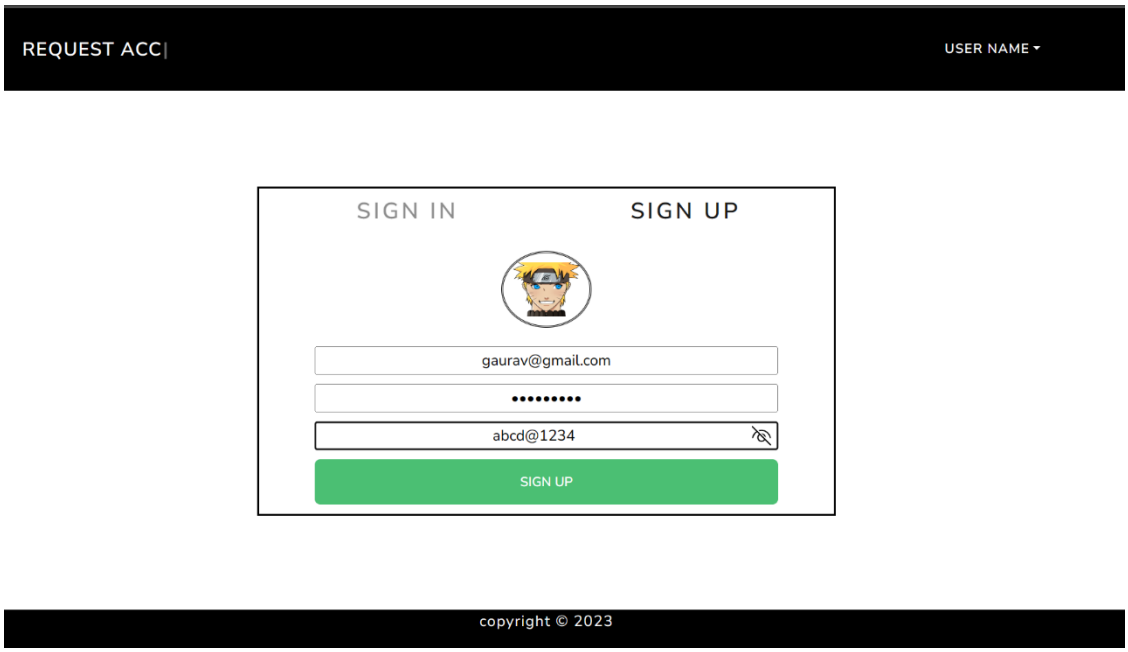
```
const asyncHandler = require("express-async-handler");
const express = require("express");
const router = express.Router();
const mongoose = require("mongoose");
const { User, createUser } = require("../models/User");
const bcrypt = require("bcrypt");
const tokenGenerator = require("../utils/jwt");

// siginup route


router.post(
  "/",
  asyncHandler(async (req, res) => {
    var { email, password } = req.body;
    email = email.toLowerCase();
    const userExists = await User.findOne({ email: email });
```

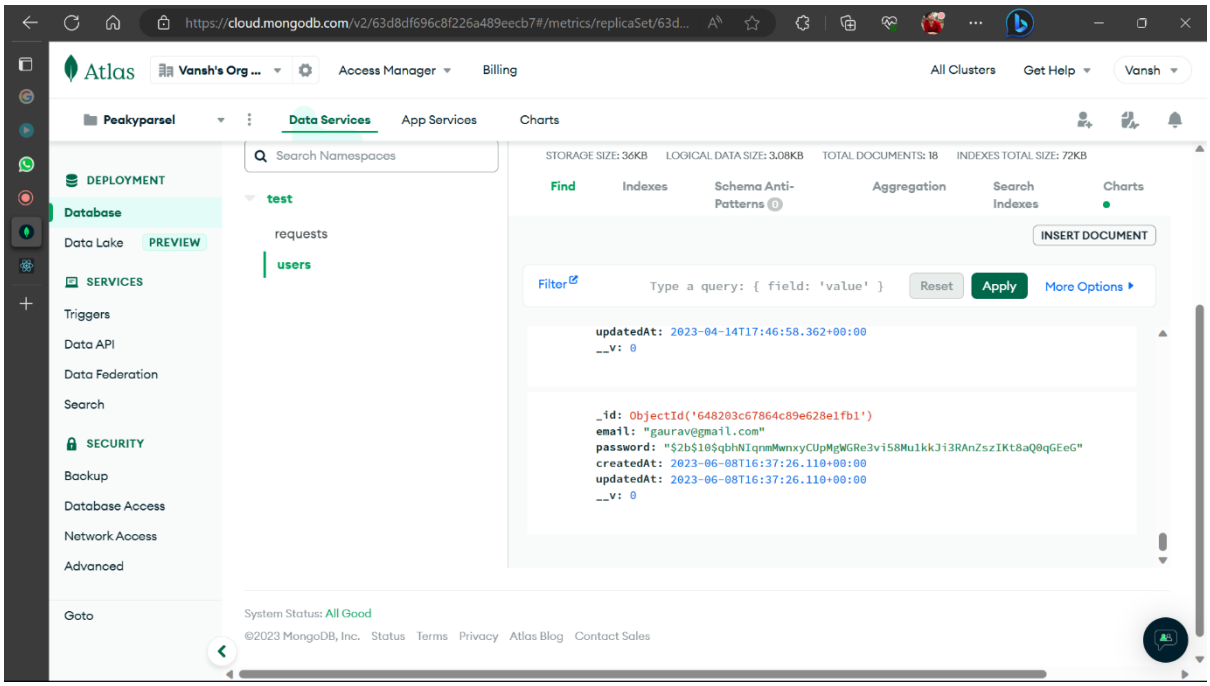
```
  if (userExists) {
    console.log("user already exists");
    res.send("User alredy exist");
  } else {
    bcrypt.hash(password, 10, function (err, hash) {
      if (err) {
        console.log(err);
      } else {
        User.create(createUser(email, hash), (err) => {
          if (err) {
            console.log(err);
          } else {
            console.log("successfully create user");
            User.findOne({ email: email }, (err, docs) => {
              if (err) {
                console.log(err);
              } else {
                // send email and jwt token
                var token = tokenGenerator(docs._id.toString());
                res.json({ email: email, token: token });
              }
            })
          }
        });
      }
    });
  }
 })
);
```

module.exports = router;



After sign up:

Saved in data base like below



Login code:

```
const asyncHandler = require("express-async-handler");
const express = require("express");
const router = express.Router();
const mongoose = require("mongoose");
const { User } = require("../models/User");
const bcrypt = require("bcrypt");
const tokenGenerator = require("../utils/jwt");

// sigin route


router.post(
  "/",
  asyncHandler(async (req, res) => {
    var { email, password } = req.body;
    email = email.toLowerCase();
    User.findOne({ email: email }, (err, docs) => {
      if (err) {
        console.log(err);
      } else {
        if (docs === null) {
            res.send("invalid credential");
        } else {
          bcrypt.compare(password, docs.password, function (err, result) {
            if (err) {
              console.log(err);
              res.send("invalid credential");
            } else {
              if (result) {
```

```
            var token = tokenGenerator(docs._id.toString());
            res.json({ email: docs.email, token: token });
          } else {
            res.send("invalid password");
          }
        }
      });
    }
  }
 });
});

module.exports = router;
```