Encryption and Hashing

Encryption is the process of encoding information. This process converts the original representation of the information, known as plaintext, into an alternative form known as ciphertext. Ideally, only authorized parties can decipher a ciphertext back to plaintext and access the original information. Encryption does not itself prevent interference but denies the intelligible content to a would-be interceptor. Encryption is a two-way function; what is encrypted can be decrypted with the proper key.

Hashing is the process of converting a given key into another value. A hash function is used to generate the new value according to a mathematical algorithm. Once hashing has been done, it should be impossible to go from the output to the input.

Encryption

Node.js provides a built-in crypto module that you can use to encrypt and decrypt strings, numbers, buffers, streams, and more. Nest itself does not provide any additional package on top of this module to avoid introducing unnecessary abstractions.

As an example, let's use AES (Advanced Encryption System) `'aes-256-ctr'` algorithm CTR encryption mode.

```
import { createCipheriv, randomBytes, scrypt } from 'crypto';
import { promisify } from 'util';

const iv = randomBytes(16);
const password = 'Password used to generate key';

// The key length is dependent on the algorithm.
// In this case for aes256, it is 32 bytes.
const key = (await promisify(scrypt)(password, 'salt', 32)) as Buffer;
const cipher = createCipheriv('aes-256-ctr', key, iv);

const textToEncrypt = 'Nest';
const encryptedText = Buffer.concat([
  cipher.update(textToEncrypt),
  cipher.final(),
]);
```

Now to decrypt `encryptedText` value:

```
import { createDecipheriv } from 'crypto';

const decipher = createDecipheriv('aes-256-ctr', key, iv);
const decryptedText = Buffer.concat([
  decipher.update(encryptedText),
  decipher.final(),
]);
```

Hashing

For hashing, we recommend using either the bcrypt or argon2 packages. Nest itself does not provide any additional wrappers on top of these modules to avoid introducing unnecessary abstractions (making the learning curve short).

As an example, let's use `bcrypt` to hash a random password.

First install required packages:

```
$ npm i bcrypt
$ npm i -D @types/bcrypt
```

Once the installation is complete, you can use the `hash` function, as follows:

```
import * as bcrypt from 'bcrypt';

const saltOrRounds = 10;
const password = 'random_password';
const hash = await bcrypt.hash(password, saltOrRounds);
```

To generate a salt, use the `genSalt` function:

```
const salt = await bcrypt.genSalt();
```

To compare/check a password, use the `compare` function:

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

You can read more about available functions here.