AES Advanced Encryption Standard

AES is a 'symmetric block cipher' for encrypting texts which can be decrypted with the original encryption key.

For many purposes, a simpler encryption algorithm such as TEA is perfectly adequate – but if you suspect the world's best cryptographic minds, and a few million dollars of computing resource, might be attempting to crack your security, then AES, based on the *Rijndael* algorithm, is the tightest security currently available (approved by the US government for classified information up to 'Secret' – and in in 192 or 256 key lengths, up to 'Top Secret'). AES was adopted by NIST in 2001 as FIPS-197, and is the replacement for DES which was withdrawn in 2005.

I developed this JavaScript implementation to to illustrate the original AES standard (NIST FIPS-197) as closely as possible. It is intended as an introduction for people seeking to learn something about implementing encryption, not an authoritative implementation – cryptography experts will already know more than I present here. The emphasis is on transparency and fidelity to the standard rather than efficiency.

This script also includes a wrapper function which implements AES in the 'Counter' mode of operation (specified in NIST SP 800-38A) to encrypt arbitrary texts – many descriptions of AES limit themselves to the Cipher routine itself, and don't consider how it can be used to encrypt texts.

This is principally a learning exercise, and I am not a cryptographic expert. I can provide no warranty or guarantees if you choose to use this code in production environments.

Functional demo

Password	L0ck it up ŝaf3
Plaintext	pssst ... đon't tell anyøne!
Encrypted text	cwJ/gPY3EWbjMJvdzHy8WizV4LofqDYknNswF3Rj7BXlYaC2OHzwqw== 0.600ms
Decrypted text	pssst ... đon't tell anyøne! 0.300ms

Much of the Rijndael algorithm is based on arithmetic on a *finite field*, or *Galois field* (after the mathematician). Regular arithmetic works on an infinite range of numbers – keep doubling a number and it will get ever bigger. Arithmetic in a finite field is limited to numbers within that field. The Rijndael algorithm works in GF(2^8), in which arithmetic results can always be stored within one byte – which is pretty convenient for computers. I can't begin to understand the maths (considering that addition and subtraction are the same thing – an XOR operation – and multiplication is performed 'modulo an irreducible polynomial': doubling 0x80 in GF(2^8) gives 0x1b).

The Rijndael algorithm lends itself to widely differing implementations, since the maths can be either coded directly, or pre-computed as lookup tables – directly parallel to using log tables for arithmetic. Different implementations can have varying pay-offs between speed, complexity, and storage requirements. Some may barely resemble each other. In this implementation, I have followed the standard closely; as per the standard, I have used a lookup table ('S-box') to implement the multiplicative inverse (i.e. 1/x) within a finite field (used for the SubBytes transformation), but other calculations are made directly rather than being pre-computed.

If you want to convince yourself that the Cipher function is working properly internally (and you should!), NIST provide test vectors for AES (appendix C.1 of the standard). Click

128-bit Test Vector | 195-bit Test Vector | 256-bit Test Vector

and the cipher output block should be

- 128-bit: *69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a*
- 192-bit: *dd a9 7c a4 86 4c df e0 6e af 70 a0 ec 0d 71 91*
- 256-bit: *8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89*

(In counter mode, a text could decrypt correctly even if the cipher routine was flawed).

The Inverse Cipher is largely a mirror of the Cipher routine, with parallel functions for Cipher, SubBytes and ShiftRows. The MixColumns routine is slightly more complex in the inverse. I have not implemented the inverse cipher here as it is not required in counter mode.

Counter mode of operation: the AES standard concerns itself with numeric or binary data (Rijndael, along with most other encryption algorithms, works on a fixed-size block of numbers – in the case of AES, each block is 128 bits or 16 bytes).

In order to make use of it to encrypt real things (such as texts), it has to be used within a certain 'mode of operation'. This is the interface between text or files, and the purely numerical encryption algorithm. See NIST Special Publication SP800-38A for more details and test vectors.

The simplest mode of operation ('electronic codebook') encrypts a text block-by-block – but since the same block of plaintext will always generate the same block of ciphertext, this can leave too many clues for attackers.

In the 'counter mode' used in this implementation, a counter which changes with each block is first encrypted, and the result is bitwise XOR'd with the plaintext block to get the ciphertext block (so the plaintext is not actually directly encrypted). A unique 'nonce' is incorporated in the counter to ensure different ciphertexts are always generated from the same plaintext every time it is encrypted; this number is stored at the head of the ciphertext to enable decryption. A combination of seconds since 1 Jan 1970, a millisecond-timestamp, and a sub-millisecond random number gives a very effective nonce. (To resist cryptographic attacks, the nonce does not need to be secret or unpredictable, but it is imperative that it is unique). In this implementation, the initial block holds the nonce in the first 8 bytes, and the block count in the second 8 bytes. Since JavaScript can represent integers up to 2^{53}, this allows a message size up to 2^{57} (c. 10^{17}) bytes – unlikely to be a limitation! Note that the nonce in counter mode is the equivalent of the initialisation vector (IV) in other modes of operation.

A curious quality of counter mode is that decryption also uses the cipher algorithm rather than the inverse-cipher algorithm. Though simple to implement, it has been established to be very secure.

Encrypting texts or files require not just the mode of operation. When implementing AES, you have to consider

- mode of operation; here the Counter (CTR) mode of operation – both simple to implement, and very secure
- conversion of text (including multi-byte Unicode texts) to binary/numeric data; here multi-byte Unicode characters are converted to UTF8, then the numeric character codes are used to pass to the cipher routine
- conversion of encrypted data to values which can be stored or transmitted without problem; here the binary encrypted texts are encoded in Base64, which is a very safe 7-bit encoding with no control codes or other troublesome characters.

The **key** in this script is obtained by applying the Cipher routine to encrypt the first 16/24/32 characters of the password (for 128-/192-/256-bit keys) to make the key. This is a convenient way to obtain a secure key within an entirely self-contained script (in a production environment, as opposed to this essentially tutorial code, the key might be generated as a hash, e.g. simply `key = Sha256(password)`). In more detail, the supplied password is converted to to UTF-8 (to be byte-safe), then the first 16/24/32 characters are converted to bytes. The resulting pwBytes is used as a seed for the Aes.keyExpansion() which is then used as the key to encrypt pwBytes with Aes.cipher(). Examples of keys generated in this way from (unrealistically) simple passwords:

'a' (U+0061): pwBytes = `61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`

key = `60 84 dd 49 14 7b 5d 05 7a e3 f8 81 b9 0e e7 dd`

```
'b' (U+0062):  pwBytes = 62 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                   key = b4 1a 83 4f da 4b aa 41 76 62 be d6 2c 66 83 6d
'☺' (U+263a): pwBytes = e2 98 ba 00 00 00 00 00 00 00 00 00 00 00 00 00
                   key = d1 0c cd fd 44 45 54 ef 59 aa f8 dc 78 8e 9a 7c
```

Even with a single bit difference between two passwords ('a' and 'b'), the key is entirely different.

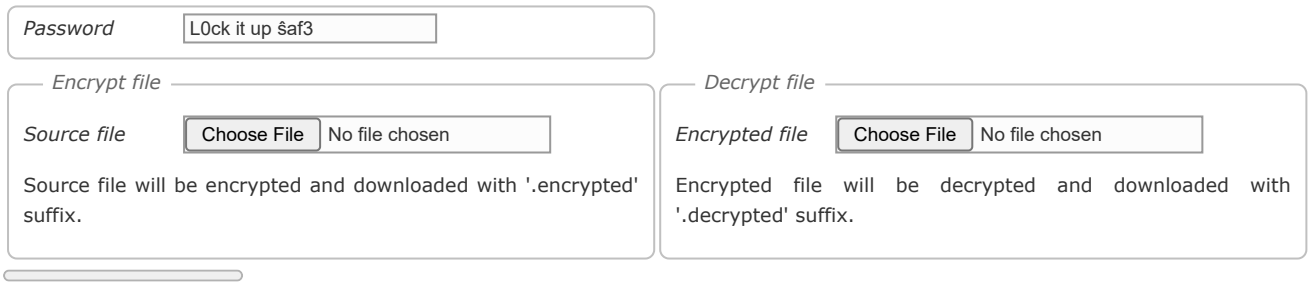
Usage: this implementation would be invoked as follows:

```
import AesCtr from './aes-ctr.js';
const password = 'L0ck it up saf3';
const plaintext = 'pssst ... đon't tell anyøne!';
const ciphertext = AesCtr.encrypt(plaintext, password, 256);
const origtext = AesCtr.decrypt(ciphertext, password, 256);
```

Note that there are no standards for data storage formats of AES encryption mode wrapper functions, so this is unlikely to inter-operate with standard library functions. It inter-operates between client-side JavaScript and Node.js: see gist.github.com/chrisveness/b28bd30b2b0c03806b0c. I have also written a matching PHP version.

Tests: if you want to be confident the AES-CTR encryption/decryption is working properly (and you should!), a set of mocha/chai tests based on NIST test vectors & a range of encryptions/decryptions of different lengths is available at www.movable-type.co.uk/scripts/test/aes-test.html.

It is also quite simple to encrypt files, by using web workers, FileReader & Blob objects, and Eli Grey's `saveAs()`:



Note that AesCtr.encrypt expects a string: as binary files may include invalid Unicode sequences if treated as strings, I treat the file contents as a byte-stream, converting it to single-byte characters before passing it to AesCtr.encrypt.

Does it make sense to implement AES in JavaScript? This is really intended as a reference implementation to help understand the AES standard, but sometimes JavaScript can be used for real-world cryptographic applications (particularly web-based ones). A JavaScript implementation such as this can also provide an easy starting-point for implementation in other languages – though I still think that TEA is generally good enough for simple applications, and a great deal simpler to use (as well as significantly faster, according to tests by Tom Doan, thanks Tom). For production use, it's always a good idea to make use of a standard library where possible, in preference to home-grown solutions.

In other languages: I've developed a PHP version which directly mirrors this JavaScript version; it differs in that PHP has Base64 encoding and UTF-8 encoding built-in, and has no unsigned-right-shift operator(!), but is otherwise a straightforward port. In other languages, be sure to use 64-bit integers/longs, either unsigned or with unsigned right-shift operators; you may need to take into consideration the way different languages handle bitwise ops, and of course standard issues such as array handling and strict typing. I'm not aware of any other issues.

I'm not familiar with Python, but there is a Python version available at wiki.birth-online.de/snippets/python/aes-

rijndael.

Speed: as mentioned, this is not an optimised implementation – using Chrome on a low-to-middling 2014 machine (Core-i5), this processes around 1MB/sec [still some 100× faster than back in 2008!).

For more information, have a look at

- Daemen & Rijnael's AES proposal for Rijndael Block Cipher
- Wikipedia article
- article from John Savard's Cryptographic Compendium

For some security applications, a cryptographic hash is more appropriate than encryption – if you are interested in a hash function, see my implementations of SHA-1 and SHA-256.

October 2009: I have updated the formulation of these scripts to use JavaScript namespaces for better encapsulation of function names.

Note that these scripts are intended to assist in studying the algorithms, not for production use. For production use, I would recommend the *Web Cryptography API* for the browser (see *example*), or the *crypto* library in Node.js – though if you are using this code, I emphasise that I have no reason at all to doubt its integrity.

See below for the source code of the JavaScript implementation, also available on GitHub. §ection numbers relate the code back to sections in the standard.

With its untyped C-style syntax, JavaScript reads remarkably close to pseudo-code: exposing the algorithms with a minimum of syntactic distractions. These functions should be simple to translate into other languages if required, though can also be used as-is in browsers and Node.js.

I offer these scripts for free use and adaptation to balance my debt to the open-source info-verse. You are welcome to re-use these scripts [under an MIT licence, without any warranty express or implied] provided solely that you retain my copyright notice and a link to this page.

If you would like to show your appreciation and support continued development of these scripts, I would most gratefully accept donations.

Donate

If you have any queries or find any problems, contact me at scripts-enc@movable-type.co.uk.

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* AES implementation in JavaScript                                   (c) Chris Veness 2005-2019  */
/*                                                                                   MIT Licence  */
/* www.movable-type.co.uk/scripts/aes.html                                                        */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */


/**
 * AES (Rijndael cipher) encryption routines reference implementation,
 *
 * This is an annotated direct implementation of FIPS 197, without any optimisations. It is
 * intended to aid understanding of the algorithm rather than for production use.
 *
 * While it could be used where performance is not critical, I would recommend using the ‘Web
 * Cryptography API’ (developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/encrypt) for the browser,
 * or the ‘crypto’ library (nodejs.org/api/crypto.html#crypto_class_cipher) in Node.js.
 *
 * See csrc.nist.gov/publications/fips/fips197/fips-197.pdf
 */
class Aes {

    /**
     * AES Cipher function: encrypt 'input' state with Rijndael algorithm [§5.1];
     *   applies Nr rounds (10/12/14) using key schedule w for 'add round key' stage.
     *
```

```
     * @param   {number[]}   input - 16-byte (128-bit) input state array.
     * @param   {number[][]} w - Key schedule as 2D byte-array (Nr+1 × Nb bytes).
     * @returns {number[]}   Encrypted output state array.
     */
    static cipher(input, w) {
        const Nb = 4;               // block size (in words): no of columns in state (fixed at 4 for AES)
        const Nr = w.length/Nb - 1; // no of rounds: 10/12/14 for 128/192/256-bit keys

        let state = [ [], [], [], [] ];  // initialise 4xNb byte-array 'state' with input [§3.4]
        for (let i=0; i<4*Nb; i++) state[i%4][Math.floor(i/4)] = input[i];

        state = Aes.addRoundKey(state, w, 0, Nb);

        for (let round=1; round<Nr; round++) {
            state = Aes.subBytes(state, Nb);
            state = Aes.shiftRows(state, Nb);
            state = Aes.mixColumns(state, Nb);
            state = Aes.addRoundKey(state, w, round, Nb);
        }

        state = Aes.subBytes(state, Nb);
        state = Aes.shiftRows(state, Nb);
        state = Aes.addRoundKey(state, w, Nr, Nb);

        const output = new Array(4*Nb);  // convert state to 1-d array before returning [§3.4]
        for (let i=0; i<4*Nb; i++) output[i] = state[i%4][Math.floor(i/4)];

        return output;
    }


    /**
     * Perform key expansion to generate a key schedule from a cipher key [§5.2].
     *
     * @param   {number[]}   key - Cipher key as 16/24/32-byte array.
     * @returns {number[][]} Expanded key schedule as 2D byte-array (Nr+1 × Nb bytes).
     */
    static keyExpansion(key) {
        const Nb = 4;            // block size (in words): no of columns in state (fixed at 4 for AES)
        const Nk = key.length/4; // key length (in words): 4/6/8 for 128/192/256-bit keys
        const Nr = Nk + 6;       // no of rounds: 10/12/14 for 128/192/256-bit keys

        const w = new Array(Nb*(Nr+1));
        let temp = new Array(4);

        // initialise first Nk words of expanded key with cipher key
        for (let i=0; i<Nk; i++) {
            const r = [ key[4*i], key[4*i+1], key[4*i+2], key[4*i+3] ];
            w[i] = r;
        }

        // expand the key into the remainder of the schedule
        for (let i=Nk; i<(Nb*(Nr+1)); i++) {
            w[i] = new Array(4);
            for (let t=0; t<4; t++) temp[t] = w[i-1][t];
            // each Nk'th word has extra transformation
            if (i % Nk == 0) {
                temp = Aes.subWord(Aes.rotWord(temp));
                for (let t=0; t<4; t++) temp[t] ^= Aes.rCon[i/Nk][t];
            }
            // 256-bit key has subWord applied every 4th word
            else if (Nk > 6 && i%Nk == 4) {
                temp = Aes.subWord(temp);
            }
            // xor w[i] with w[i-1] and w[i-Nk]
            for (let t=0; t<4; t++) w[i][t] = w[i-Nk][t] ^ temp[t];
        }

        return w;
    }


    /**
     * Apply SBox to state S [§5.1.1].
     *
     * @private
```

```
     */
    static subBytes(s, Nb) {
        for (let r=0; r<4; r++) {
            for (let c=0; c<Nb; c++) s[r][c] = Aes.sBox[s[r][c]];
        }
        return s;
    }


    /**
     * Shift row r of state S left by r bytes [§5.1.2].
     *
     * @private
     */
    static shiftRows(s, Nb) {
        const t = new Array(4);
        for (let r=1; r<4; r++) {
            for (let c=0; c<4; c++) t[c] = s[r][(c+r)%Nb];  // shift into temp copy
            for (let c=0; c<4; c++) s[r][c] = t[c];         // and copy back
        }          // note that this will work for Nb=4,5,6, but not 7,8 (always 4 for AES):
        return s;  // see asmaes.sourceforge.net/rijndael/rijndaelImplementation.pdf
    }


    /**
     * Combine bytes of each col of state S [§5.1.3].
     *
     * @private
     */
    static mixColumns(s, Nb) {
        for (let c=0; c<Nb; c++) {
            const a = new Array(Nb);  // 'a' is a copy of the current column from 's'
            const b = new Array(Nb);  // 'b' is a•{02} in GF(2^8)
            for (let r=0; r<4; r++) {
                a[r] = s[r][c];
                b[r] = s[r][c]&0x80 ? s[r][c]<<1 ^ 0x011b : s[r][c]<<1;
            }
            // a[n] ^ b[n] is a•{03} in GF(2^8)
            s[0][c] = b[0] ^ a[1] ^ b[1] ^ a[2] ^ a[3]; // {02}•a0 + {03}•a1 + a2 + a3
            s[1][c] = a[0] ^ b[1] ^ a[2] ^ b[2] ^ a[3]; // a0 • {02}•a1 + {03}•a2 + a3
            s[2][c] = a[0] ^ a[1] ^ b[2] ^ a[3] ^ b[3]; // a0 + a1 + {02}•a2 + {03}•a3
            s[3][c] = a[0] ^ b[0] ^ a[1] ^ a[2] ^ b[3]; // {03}•a0 + a1 + a2 + {02}•a3
        }
        return s;
    }


    /**
     * Xor Round Key into state S [§5.1.4].
     *
     * @private
     */
    static addRoundKey(state, w, rnd, Nb) {
        for (let r=0; r<4; r++) {
            for (let c=0; c<Nb; c++) state[r][c] ^= w[rnd*4+c][r];
        }
        return state;
    }


    /**
     * Apply SBox to 4-byte word w.
     *
     * @private
     */
    static subWord(w) {
        for (let i=0; i<4; i++) w[i] = Aes.sBox[w[i]];
        return w;
    }


    /**
     * Rotate 4-byte word w left by one byte.
     *
     * @private
```

```
     */
    static rotWord(w) {
        const tmp = w[0];
        for (let i=0; i<3; i++) w[i] = w[i+1];
        w[3] = tmp;
        return w;
    }


}


// sBox is pre-computed multiplicative inverse in GF(2^8) used in subBytes and keyExpansion [§5.1.1]
Aes.sBox = [
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16,
];


// rCon is Round Constant used for the Key Expansion [1st col is 2^(r-1) in GF(2^8)] [§5.2]
Aes.rCon = [
    [ 0x00, 0x00, 0x00, 0x00 ],
    [ 0x01, 0x00, 0x00, 0x00 ],
    [ 0x02, 0x00, 0x00, 0x00 ],
    [ 0x04, 0x00, 0x00, 0x00 ],
    [ 0x08, 0x00, 0x00, 0x00 ],
    [ 0x10, 0x00, 0x00, 0x00 ],
    [ 0x20, 0x00, 0x00, 0x00 ],
    [ 0x40, 0x00, 0x00, 0x00 ],
    [ 0x80, 0x00, 0x00, 0x00 ],
    [ 0x1b, 0x00, 0x00, 0x00 ],
    [ 0x36, 0x00, 0x00, 0x00 ],
];


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

export default Aes;
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* AES counter-mode (CTR) implementation in JavaScript                (c) Chris Veness 2005-2019  */
/*                                                                                 MIT Licence  */
/* www.movable-type.co.uk/scripts/aes.html                                                    */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

/* global WorkerGlobalScope */


import Aes from './aes.js';


/**
 * AesCtr: Counter-mode (CTR) wrapper for AES.
 *
 * This encrypts a Unicode string to produces a base64 ciphertext using 128/192/256-bit AES,
 * and the converse to decrypt an encrypted ciphertext.
 *
 * See csrc.nist.gov/publications/detail/sp/800-38a/final
 */
class AesCtr extends Aes {
```

```
/**
 * Encrypt a text using AES encryption in Counter mode of operation.
 *
 * Unicode multi-byte character safe.
 *
 * @param   {string} plaintext - Source text to be encrypted.
 * @param   {string} password - The password to use to generate a key for encryption.
 * @param   {number} nBits - Number of bits to be used in the key; 128 / 192 / 256.
 * @returns {string} Encrypted text, base-64 encoded.
 *
 * @example
 *   const encr = AesCtr.encrypt('big secret', 'pāşšŵōřđ', 256); // 'lwGl66VVwVObKIr6of8HVqJr'
 */
static encrypt(plaintext, password, nBits) {
    if (![ 128, 192, 256 ].includes(nBits)) throw new Error('Key size is not 128 / 192 / 256');
    plaintext = AesCtr.utf8Encode(String(plaintext));
    password = AesCtr.utf8Encode(String(password));

    // use AES itself to encrypt password to get cipher key (using plain password as source for key
    // expansion) to give us well encrypted key (in real use hashed password could be used for key)
    const nBytes = nBits/8; // no bytes in key (16/24/32)
    const pwBytes = new Array(nBytes);
    for (let i=0; i<nBytes; i++) { // use 1st 16/24/32 chars of password for key
        pwBytes[i] = i<password.length ?  password.charCodeAt(i) : 0;
    }
    let key = Aes.cipher(pwBytes, Aes.keyExpansion(pwBytes)); // gives us 16-byte key
    key = key.concat(key.slice(0, nBytes-16)); // expand key to 16/24/32 bytes long

    // initialise 1st 8 bytes of counter block with nonce (NIST SP 800-38A §B.2): [0-1] = millisec,
    // [2-3] = random, [4-7] = seconds, together giving full sub-millisec uniqueness up to Feb 2106
    const timestamp = (new Date()).getTime(); // milliseconds since 1-Jan-1970
    const nonceMs = timestamp%1000;
    const nonceSec = Math.floor(timestamp/1000);
    const nonceRnd = Math.floor(Math.random()*0xffff);
    // for debugging: const [ nonceMs, nonceSec, nonceRnd ] = [ 0, 0, 0 ];
    const counterBlock = [ // 16-byte array; blocksize is fixed at 16 for AES
        nonceMs  & 0xff, nonceMs >>>8 & 0xff,
        nonceRnd & 0xff, nonceRnd>>>8 & 0xff,
        nonceSec & 0xff, nonceSec>>>8 & 0xff, nonceSec>>>16 & 0xff, nonceSec>>>24 & 0xff,
        0, 0, 0, 0, 0, 0, 0, 0,
    ];

    // and convert nonce to a string to go on the front of the ciphertext
    const nonceStr = counterBlock.slice(0, 8).map(i => String.fromCharCode(i)).join('');

    // convert (utf-8) plaintext to byte array
    const plaintextBytes = plaintext.split('').map(ch => ch.charCodeAt(0));

    // ------------ perform encryption ------------
    const ciphertextBytes = AesCtr.nistEncryption(plaintextBytes, key, counterBlock);

    // convert byte array to (utf-8) ciphertext string
    const ciphertextUtf8 = ciphertextBytes.map(i => String.fromCharCode(i)).join('');

    // base-64 encode ciphertext
    const ciphertextB64 =  AesCtr.base64Encode(nonceStr+ciphertextUtf8);

    return ciphertextB64;
}

/**
 * NIST SP 800-38A sets out recommendations for block cipher modes of operation in terms of byte
 * operations. This implements the §6.5 Counter Mode (CTR).
 *
 *     Oⱼ = CIPHₖ(Tⱼ)      for j = 1, 2 … n
 *     Cⱼ = Pⱼ ⊕ Oⱼ        for j = 1, 2 … n-1
 *     C*ₙ = P* ⊕ MSBᵤ(Oₙ) final (partial?) block
 *   where CIPHₖ is the forward cipher function, O output blocks, P plaintext blocks, C
 *   ciphertext blocks
 *
 * @param   {number[]} plaintext - Plaintext to be encrypted, as byte array.
 * @param   {number[]} key - Key to be used to encrypt plaintext.
 * @param   {number[]} counterBlock - Initial 16-byte CTR counter block (with nonce & 0 counter).
 * @returns {number[]} Ciphertext as byte array.
```

```
     *
     * @private
     */
    static nistEncryption(plaintext, key, counterBlock) {
        const blockSize = 16; // block size fixed at 16 bytes / 128 bits (Nb=4) for AES

        // generate key schedule - an expansion of the key into distinct Key Rounds for each round
        const keySchedule = Aes.keyExpansion(key);

        const blockCount = Math.ceil(plaintext.length/blockSize);
        const ciphertext = new Array(plaintext.length);

        for (let b=0; b<blockCount; b++) {
            // ---- encrypt counter block; Oⱼ = CIPHₖ(Tⱼ) ----
            const cipherCntr = Aes.cipher(counterBlock, keySchedule);

            // block size is reduced on final block
            const blockLength = b<blockCount-1 ? blockSize : (plaintext.length-1)%blockSize + 1;

            // ---- xor plaintext with ciphered counter byte-by-byte; Cⱼ = Pⱼ ⊕ Oⱼ ----
            for (let i=0; i<blockLength; i++) {
                ciphertext[b*blockSize + i] = cipherCntr[i] ^ plaintext[b*blockSize + i];
            }

            // increment counter block (counter in 2nd 8 bytes of counter block, big-endian)
            counterBlock[blockSize-1]++;
            // and propagate carry digits
            for (let i=blockSize-1; i>=8; i--) {
                counterBlock[i-1] += counterBlock[i] >> 8;
                counterBlock[i] &= 0xff;
            }

            // if within web worker, announce progress every 1000 blocks (roughly every 50ms)
            if (typeof workerGlobalScope != 'undefined' && self instanceof workerGlobalScope) {
                if (b%1000 == 0) self.postMessage({ progress: b/blockCount });
            }
        }

        return ciphertext;
    }


    /**
     * Decrypt a text encrypted by AES in counter mode of operation.
     *
     * @param   {string} ciphertext - Cipher text to be decrypted.
     * @param   {string} password - Password to use to generate a key for decryption.
     * @param   {number} nBits - Number of bits to be used in the key; 128 / 192 / 256.
     * @returns {string} Decrypted text
     *
     * @example
     *   const decr = AesCtr.decrypt('lwGl66VVwVObKIr6of8HVqJr', 'pāṣšŵōřđ', 256); // 'big secret'
     */
    static decrypt(ciphertext, password, nBits) {
        if (![ 128, 192, 256 ].includes(nBits)) throw new Error('Key size is not 128 / 192 / 256');
        ciphertext = AesCtr.base64Decode(String(ciphertext));
        password = AesCtr.utf8Encode(String(password));

        // use AES to encrypt password (mirroring encrypt routine)
        const nBytes = nBits/8; // no bytes in key
        const pwBytes = new Array(nBytes);
        for (let i=0; i<nBytes; i++) { // use 1st nBytes chars of password for key
            pwBytes[i] = i<password.length ?  password.charCodeAt(i) : 0;
        }
        let key = Aes.cipher(pwBytes, Aes.keyExpansion(pwBytes));
        key = key.concat(key.slice(0, nBytes-16)); // expand key to 16/24/32 bytes long

        // recover nonce from 1st 8 bytes of ciphertext into 1st 8 bytes of counter block
        const counterBlock = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];
        for (let i=0; i<8; i++) counterBlock[i] = ciphertext.charCodeAt(i);

        // convert ciphertext to byte array (skipping past initial 8 bytes)
        const ciphertextBytes = new Array(ciphertext.length-8);
        for (let i=8; i<ciphertext.length; i++) ciphertextBytes[i-8] = ciphertext.charCodeAt(i);

        // ------------ perform decryption ------------
```

```
        const plaintextBytes = AesCtr.nistDecryption(ciphertextBytes, key, counterBlock);

        // convert byte array to (utf-8) plaintext string
        const plaintextUtf8 = plaintextBytes.map(i => String.fromCharCode(i)).join('');

        // decode from UTF8 back to Unicode multi-byte chars
        const plaintext = AesCtr.utf8Decode(plaintextUtf8);

        return plaintext;
    }

    /**
     * NIST SP 800-38A sets out recommendations for block cipher modes of operation in terms of byte
     * operations. This implements the §6.5 Counter Mode (CTR).
     *
     *     Oⱼ = CIPHₖ(Tⱼ)      for j = 1, 2 … n
     *     Pⱼ = Cⱼ ⊕ Oⱼ        for j = 1, 2 … n-1
     *     P*ₙ = C* ⊕ MSBᵤ(Oₙ) final (partial?) block
     *   where CIPHₖ is the forward cipher function, O output blocks, C ciphertext blocks, P
     *   plaintext blocks
     *
     * @param   {number[]} ciphertext - Ciphertext to be decrypted, as byte array.
     * @param   {number[]} key - Key to be used to decrypt ciphertext.
     * @param   {number[]} counterBlock - Initial 16-byte CTR counter block (with nonce & 0 counter).
     * @returns {number[]} Plaintext as byte array.
     *
     * @private
     */
    static nistDecryption(ciphertext, key, counterBlock) {
        const blockSize = 16; // block size fixed at 16 bytes / 128 bits (Nb=4) for AES

        // generate key schedule - an expansion of the key into distinct Key Rounds for each round
        const keySchedule = Aes.keyExpansion(key);

        const blockCount = Math.ceil(ciphertext.length/blockSize);
        const plaintext = new Array(ciphertext.length);

        for (let b=0; b<blockCount; b++) {
            // ---- decrypt counter block; Oⱼ = CIPHₖ(Tⱼ) ----
            const cipherCntr = Aes.cipher(counterBlock, keySchedule);

            // block size is reduced on final block
            const blockLength = b<blockCount-1 ? blockSize : (ciphertext.length-1)%blockSize + 1;

            // ---- xor ciphertext with ciphered counter byte-by-byte; Pⱼ = Cⱼ ⊕ Oⱼ ----
            for (let i=0; i<blockLength; i++) {
                plaintext[b*blockSize + i] = cipherCntr[i] ^ ciphertext[b*blockSize + i];
            }

            // increment counter block (counter in 2nd 8 bytes of counter block, big-endian)
            counterBlock[blockSize-1]++;
            // and propagate carry digits
            for (let i=blockSize-1; i>=8; i--) {
                counterBlock[i-1] += counterBlock[i] >> 8;
                counterBlock[i] &= 0xff;
            }

            // if within web worker, announce progress every 1000 blocks (roughly every 50ms)
            if (typeof WorkerGlobalScope != 'undefined' && self instanceof WorkerGlobalScope) {
                if (b%1000 == 0) self.postMessage({ progress: b/blockCount });
            }
        }

        return plaintext;
    }


    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */


    /**
     * Encodes multi-byte string to utf8.
     *
     * Note utf8Encode is an identity function with 7-bit ascii strings, but not with 8-bit strings;
     * utf8Encode('x') = 'x', but utf8Encode('ça') = 'Ã§a', and utf8Encode('Ã§a') = 'ÃƒÂ§a'.
```

```
     */
    static utf8Encode(str) {
        try {
            return new TextEncoder().encode(str, 'utf-8').reduce((prev, curr) => prev + String.fromCharCode(curr), '');
        } catch (e) { // no TextEncoder available?
            return unescape(encodeURIComponent(str)); // monsur.hossa.in/2012/07/20/utf-8-in-javascript.html
        }
    }

    /**
     * Decodes utf8 string to multi-byte.
     */
    static utf8Decode(str) {
        try {
            return new TextEncoder().decode(str, 'utf-8').reduce((prev, curr) => prev + String.fromCharCode(curr), '');
        } catch (e) { // no TextEncoder available?
            return decodeURIComponent(escape(str)); // monsur.hossa.in/2012/07/20/utf-8-in-javascript.html
        }
    }

    /*
     * Encodes string as base-64.
     *
     * - developer.mozilla.org/en-US/docs/Web/API/window.btoa, nodejs.org/api/buffer.html
     * - note: btoa & Buffer/binary work on single-byte Unicode (C0/C1), so ok for utf8 strings, not for general Unicode...
     * - note: if btoa()/atob() are not available (eg IE9-), try github.com/davidchambers/Base64.js
     */
    static base64Encode(str) {
        if (typeof btoa != 'undefined') return btoa(str); // browser
        if (typeof Buffer != 'undefined') return new Buffer(str, 'binary').toString('base64'); // Node.js
        throw new Error('No Base64 Encode');
    }

    /*
     * Decodes base-64 encoded string.
     */
    static base64Decode(str) {
        if (typeof atob != 'undefined') return atob(str); // browser
        if (typeof Buffer != 'undefined') return new Buffer(str, 'base64').toString('binary'); // Node.js
        throw new Error('No Base64 Decode');
    }

}


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

export default AesCtr;
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/*  Web worker to encrypt/decrypt files using AES counter-mode        (c) Chris Veness 2016-2018  */
/*                                                                                    MIT Licence  */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

import AesCtr from './js/crypto/aes-ctr.js';

/**
 * Web worker to encrypt/decrypt files using AES counter-mode.
 *
 * @param {string} msg.data.op - 'encrypt' or 'decrypt'.
 * @param {File}   msg.data.file - File to be encrypted or decrypted.
 * @param {string} msg.data.password - Password to use to encrypt/decrypt file.
 * @param {number} msg.data.bits - Number of bits to use for key.
 * @returns {ciphertext|plaintext} - Blob containing encrypted ciphertext / decrypted plaintext.
 *
 * @example
 *   var worker = new Worker('aes-ctr-file-webworker.js');
 *   var file = this.files[0];
 *   worker.postMessage({ op:'encrypt', file:file, password:'L0ck it up ŝaf3', bits:256 });
 *   worker.onmessage = function(msg) {
 *     if (msg.data.progress != 'complete') {
 *       $('progress').val(msg.data.progress * 100); // update progress bar
 *     }
 *     if (msg.data.progress == 'complete') {
```

```
 *       saveAs(msg.data.ciphertext, file.name+'.encrypted'); // save encrypted file
 *     }
 *   }
 *
 * Note saveAs() cannot run in web worker, so encrypted/decrypted file has to be passed back to UI
 * thread to be saved.
 *
 * TODO: error handling on failed decryption
 */
onmessage = function(msg) {
    switch (msg.data.op) {
        case 'encrypt':
            var reader = new FileReaderSync();
            var plaintext = reader.readAsText(msg.data.file, 'utf-8');
            var ciphertext = AesCtr.encrypt(plaintext, msg.data.password, msg.data.bits);
            // return encrypted file as Blob; UI thread can then use saveAs()
            var blob = new Blob([ciphertext], { type: 'text/plain' });
            self.postMessage({ progress: 'complete', ciphertext: blob });
            break;
        case 'decrypt':
            var reader = new FileReaderSync();
            var ciphertext = reader.readAsText(msg.data.file, 'iso-8859-1');
            var plaintext = AesCtr.decrypt(ciphertext, msg.data.password, msg.data.bits);
            // return decrypted file as Blob; UI thread can then use saveAs()
            var blob = new Blob([plaintext], { type: 'application/octet-stream' });
            self.postMessage({ progress: 'complete', plaintext: blob });
            break;
    }
};
```