[bcryptjs](https://www.npmjs.com/package/bcryptjs)

Top of Form

Bottom of Form

Optimized bcrypt in JavaScript with zero dependencies. Compatible to the C++ **[bcrypt](https://npmjs.org/package/bcrypt)** binding on node.js and also working in the browser.

**Security considerations**

Besides incorporating a salt to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power. ([**see**](http://en.wikipedia.org/wiki/Bcrypt))

While bcrypt.js is compatible to the C++ bcrypt binding, it is written in pure JavaScript and thus slower ([**about 30%**](https://github.com/dcodeIO/bcrypt.js/wiki/Benchmark)), effectively reducing the number of iterations that can be processed in an equal time span.

The maximum input length is 72 bytes (note that UTF8 encoded characters use up to 4 bytes) and the length of generated hashes is 60 characters.

**Usage**

The library is compatible with CommonJS and AMD loaders and is exposed globally as dcodeIO.bcrypt if neither is available.

**node.js**

On node.js, the inbuilt [**crypto module**](http://nodejs.org/api/crypto.html)'s randomBytes interface is used to obtain secure random numbers.

npm install bcryptjs

var bcrypt **=** require('bcryptjs');

**...**

**Browser**

In the browser, bcrypt.js relies on [**Web Crypto API**](http://www.w3.org/TR/WebCryptoAPI)'s getRandomValues interface to obtain secure random numbers. If no cryptographically secure source of randomness is available, you may specify one through **[bcrypt.setRandomFallback](https://github.com/dcodeIO/bcrypt.js" \l "setrandomfallbackrandom)**.

var bcrypt **=** dcodeIO.bcrypt;

**...**

or

require.config({

    paths**:** { "bcrypt"**:** "/path/to/bcrypt.js" }

});

require(["bcrypt"], function(bcrypt) {

**...**

});

**Usage - Sync**

To hash a password:

var bcrypt **=** require('bcryptjs');

var salt **=** bcrypt.genSaltSync(10);

var hash **=** bcrypt.hashSync("B4c0/\/", salt);

*// Store hash in your password DB.*

To check a password:

*// Load hash from your password DB.*

bcrypt.compareSync("B4c0/\/", hash); *// true*

bcrypt.compareSync("not\_bacon", hash); *// false*

Auto-gen a salt and hash:

var hash **=** bcrypt.hashSync('bacon', 8);

**Usage - Async**

To hash a password:

var bcrypt **=** require('bcryptjs');

bcrypt.genSalt(10, function(err, salt) {

    bcrypt.hash("B4c0/\/", salt, function(err, hash) {

*// Store hash in your password DB.*

    });

});

To check a password:

*// Load hash from your password DB.*

bcrypt.compare("B4c0/\/", hash, function(err, res) {

*// res === true*

});

bcrypt.compare("not\_bacon", hash, function(err, res) {

*// res === false*

});

*// As of bcryptjs 2.4.0, compare returns a promise if callback is omitted:*

bcrypt.compare("B4c0/\/", hash).then((res) => {

*// res === true*

});

Auto-gen a salt and hash:

bcrypt.hash('bacon', 8, function(err, hash) {

});

**Note:** Under the hood, asynchronisation splits a crypto operation into small chunks. After the completion of a chunk, the execution of the next chunk is placed on the back of [**JS event loop queue**](https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop), thus efficiently sharing the computational resources with the other operations in the queue.

**API**

**setRandomFallback(random)**

Sets the pseudo random number generator to use as a fallback if neither node's crypto module nor the Web Crypto API is available. Please note: It is highly important that the PRNG used is cryptographically secure and that it is seeded properly!

| **parameter** | **type** | **description** |
| --- | --- | --- |
| random | *function(number):!Array.<number>* | Function taking the number of bytes to generate as its sole argument, returning the corresponding array of cryptographically secure random byte values. |
| **@see** |  | <http://nodejs.org/api/crypto.html> |
| **@see** |  | <http://www.w3.org/TR/WebCryptoAPI/> |

**Hint:** You might use [**isaac.js**](https://github.com/rubycon/isaac.js) as a CSPRNG but you still have to make sure to seed it properly.

**genSaltSync(rounds=, seed\_length=)**

Synchronously generates a salt.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| rounds | *number* | Number of rounds to use, defaults to 10 if omitted |
| seed\_length | *number* | Not supported. |
| **@returns** | *string* | Resulting salt |
| **@throws** | *Error* | If a random fallback is required but not set |

**genSalt(rounds=, seed\_length=, callback)**

Asynchronously generates a salt.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| rounds | *number | function(Error, string=)* | Number of rounds to use, defaults to 10 if omitted |
| seed\_length | *number | function(Error, string=)* | Not supported. |
| callback | *function(Error, string=)* | Callback receiving the error, if any, and the resulting salt |
| **@returns** | *Promise* | If callback has been omitted |
| **@throws** | *Error* | If callback is present but not a function |

**hashSync(s, salt=)**

Synchronously generates a hash for the given string.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| s | *string* | String to hash |
| salt | *number | string* | Salt length to generate or salt to use, default to 10 |
| **@returns** | *string* | Resulting hash |

**hash(s, salt, callback, progressCallback=)**

Asynchronously generates a hash for the given string.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| s | *string* | String to hash |
| salt | *number | string* | Salt length to generate or salt to use |
| callback | *function(Error, string=)* | Callback receiving the error, if any, and the resulting hash |
| progressCallback | *function(number)* | Callback successively called with the percentage of rounds completed (0.0 - 1.0), maximally once per MAX\_EXECUTION\_TIME = 100 ms. |
| **@returns** | *Promise* | If callback has been omitted |
| **@throws** | *Error* | If callback is present but not a function |

**compareSync(s, hash)**

Synchronously tests a string against a hash.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| s | *string* | String to compare |
| hash | *string* | Hash to test against |
| **@returns** | *boolean* | true if matching, otherwise false |
| **@throws** | *Error* | If an argument is illegal |

**compare(s, hash, callback, progressCallback=)**

Asynchronously compares the given data against the given hash.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| s | *string* | Data to compare |
| hash | *string* | Data to be compared to |
| callback | *function(Error, boolean)* | Callback receiving the error, if any, otherwise the result |
| progressCallback | *function(number)* | Callback successively called with the percentage of rounds completed (0.0 - 1.0), maximally once per MAX\_EXECUTION\_TIME = 100 ms. |
| **@returns** | *Promise* | If callback has been omitted |
| **@throws** | *Error* | If callback is present but not a function |

**getRounds(hash)**

Gets the number of rounds used to encrypt the specified hash.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| hash | *string* | Hash to extract the used number of rounds from |
| **@returns** | *number* | Number of rounds used |
| **@throws** | *Error* | If hash is not a string |

**getSalt(hash)**

Gets the salt portion from a hash. Does not validate the hash.

| **parameter** | **type** | **description** |
| --- | --- | --- |
| hash | *string* | Hash to extract the salt from |
| **@returns** | *string* | Extracted salt part |
| **@throws** | *Error* | If hash is not a string or otherwise invalid |

**Command line**

Usage: bcrypt <input> [salt]

If the input has spaces inside, simply surround it with quotes.