speakeasyjs / **speakeasy**

Two-factor authentication for Node.js. One-time passcode generator (HOTP/TOTP) with support for Google Authenticator.

#javascript #two-factor #two-factor-authentication #node #nodejs #node-js #mfa #hotp #totp #multi-factor

212 commits | **3** branches | **7** releases | **15** contributors | MIT

Branch: master | New pull request | Find file | Clone or download

jakelee8 committed on Sep 14, 2017 Merge pull request #87 from railsstudent/update-totp-verify-documenta... ... Latest commit `cb58351` on Sep 14, 2017

test	Merge conflict	2 years ago
.gitignore	Revive as Passcode	3 years ago
.npmignore	Add .npmignore	3 years ago
.travis.yml	[proj] Added Coveralls and updated Travis script	3 years ago
CHANGELOG.md	[maint] Update CHANGELOG.md. We're releasing today!	3 years ago
LICENSE	[maint] Updated LICENSE	3 years ago
README.md	Merge pull request #87 from railsstudent/update-totp-verify-documenta...	a year ago
index.js	Merge pull request #74 from railsstudent/otpauthURL-default-values	2 years ago
jsdoc.json	Minor updates to README and jsdoc	3 years ago
package.json	[pkg] Added bugs field to package.json	3 years ago

README.md

Speakeasy

two-factor authentication for node.js

Jump to — Install · Demo · Two-Factor Usage · General Usage · Documentation · Contributing · License

Speakeasy is a one-time passcode generator, ideal for use in two-factor authentication, that supports Google Authenticator and other two-factor devices.

It is well-tested and includes robust support for custom token lengths, authentication windows, hash algorithms like SHA256 and SHA512, and other features, and includes helpers like a secret key generator.

Speakeasy implements one-time passcode generators as standardized by the Initiative for Open Authentication (OATH). The HMAC-Based One-Time Password (HOTP) algorithm defined by RFC 4226 and the Time-Based One-time Password (TOTP) algorithm defined in RFC 6238 are supported. This project incorporates code from passcode, originally a fork of Speakeasy, and notp.

Install

```
npm install --save speakeasy
```

Demo

This demo uses the `generateSecret` method of Speakeasy to generate a secret key, displays a Google Authenticator–compatible QR code which you can scan into your phone's two-factor app, and shows the token, which you can verify with your phone. Includes sample code. https://sedemo-mktb.rhcloud.com/

View demo

Two-Factor Usage

Let's say you have a user that wants to enable two-factor authentication, and you intend to do two-factor authentication using an app like Google Authenticator, Duo Security, Authy, etc. This is a three-step process:

1. Generate a secret
2. Show a QR code for the user to scan in
3. Authenticate the token for the first time

Generating a key

Use Speakeasy's key generator to get a key.

```
var secret = speakeasy.generateSecret();
// Returns an object with secret.ascii, secret.hex, and secret.base32.
// Also returns secret.otpauth_url, which we'll use later.
```

This will generate a secret key of length 32, which will be the secret key for the user.

Now, we want to make sure that this secret works by validating the token that the user gets from it for the first time. In other words, we don't want to set this as the user's secret key just yet – we first want to verify their token for the first time. We need to persist the secret so that we can use it for token validation later.

So, store one of the encodings for the secret, preferably `secret.base32`, somewhere temporary, since we'll use that in the future to authenticate the user's first token.

```
// Example for storing the secret key somewhere (varies by implementation):
user.two_factor_temp_secret = secret.base32;
```

Displaying a QR code

Next, we'll want to display a QR code to the user so they can scan in the secret into their app. Google Authenticator and similar apps take in a QR code that holds a URL with the protocol `otpauth://`, which you get automatically from `secret.otpauth_url`.

Use a QR code module to generate a QR code that stores the data in `secret.otpauth_url`, and then display the QR code to the user. This is one simple way to do it, which generates a PNG data URL which you can put into an `<img>` tag on a webpage:

```
// Use the qrcode package
// npm install --save qrcode
var QRCode = require('qrcode');

// Get the data URL of the authenticator URL
QRCode.toDataURL(secret.otpauth_url, function(err, data_url) {
  console.log(data_url);

  // Display this data URL to the user in an <img> tag
  // Example:
  write('<img src="' + data_url + '">');
});
```

Ask the user to scan this QR code into their authenticator app.

Verifying the token

Finally, we want to make sure that the token on the server side and the token on the client side match. The best practice is to do a token check before fully enabling two-factor authenticaton for the user. This code applies to the first and subsequent token checks.

After the user scans the QR code, ask the user to enter in the token that they see in their app. Then, verify it against the secret.

```
// Let's say the user says that the token they have is 132890
var userToken = '132890';

// Let's say we stored the user's temporary secret in a user object like above:
// (This is specific to your implementation)
var base32secret = user.two_factor_temp_secret;


// Use verify() to check the token against the secret
var verified = speakeasy.totp.verify({ secret: base32secret,
                                       encoding: 'base32',
                                       token: userToken });
```

`verified` will be true if the token is successfully verified, false if not.

If successfully verified, you can now save the secret to the user's account and use the same process above whenever you need to use two-factor to authenticate the user, like during login.

```
// Example for saving user's token (varies by implementation):
user.two_factor_secret = user.two_factor_temp_secret;
user.two_factor_enabled = true
```

Now you're done implementing two-factor authentication!

General Usage

```
var speakeasy = require("speakeasy");
```

Generating a key

```
// Generate a secret key.
var secret = speakeasy.generateSecret({length: 20});
// Access using secret.ascii, secret.hex, or secret.base32.
```

Getting a time-based token for the current time

```
// Generate a time-based token based on the base-32 key.
// HOTP (counter-based tokens) can also be used if `totp` is replaced by
// `hotp` (i.e. speakeasy.hotp()) and a `counter` is given in the options.
var token = speakeasy.totp({
  secret: secret.base32,
  encoding: 'base32'
});

// Returns token for the secret at the current time
// Compare this to user input
```

Verifying a token

```
// Verify a given token
var tokenValidates = speakeasy.totp.verify({
  secret: secret.base32,
  encoding: 'base32',
  token: '123456',
  window: 6
```

```
});
// Returns true if the token matches
```

Verifying a token and calculating a delta

A TOTP is incremented every `step` time-step seconds. By default, the time-step is 30 seconds. You may change the time-step using the `step` option, with units in seconds.

```
// Verify a given token is within 3 time-steps (+/- 2 minutes) from the server
// time-step.
var tokenDelta = speakeasy.totp.verifyDelta({
  secret: secret.base32,
  encoding: 'base32',
  token: '123456',
  window: 2,
  step: 60
});
// Returns {delta: 0} where the delta is the time step difference
// between the given token and the current time
```

Getting a time-based token for a custom time

```
var token = speakeasy.totp({
  secret: secret.base32,
  encoding: 'base32',
  time: 1453667708 // specified in seconds
});

// Verify a time-based token for a custom time
var tokenValidates = speakeasy.totp.verify({
  secret: secret.base32,
  encoding: 'base32',
  token: token,
  time: 1453667708
});
```

Calculating a counter-based token

```
// Get a counter-based token
var token = speakeasy.hotp({
  secret: secret.base32,
  encoding: 'base32',
  counter: 123
});

// Verify a counter-based token
var tokenValidates = speakeasy.hotp.verify({
  secret: secret.base32,
  encoding: 'base32',
  token: '123456',
  counter: 123
});
```

Using other encodings

The default encoding (when `encoding` is not specified) is `ascii`.

```
// Specifying an ASCII token for TOTP
// (encoding is 'ascii' by default)
var token = speakeasy.totp({
  secret: secret.ascii
});



// Specifying a hex token for TOTP
var token = speakeasy.totp({
  secret: secret.hex,
  encoding: 'hex'
});
```

Using other hash algorithms

The default hash algorithm is SHA1.

```
// Specifying SHA256
var token = speakeasy.totp({
  secret: secret.ascii,
  algorithm: 'sha256'
});


// Specifying SHA512
var token = speakeasy.totp({
  secret: secret.ascii,
  algorithm: 'sha512'
});
```

Getting an otpauth:// URL and QR code for non-SHA1 hash algorithms

```
// Generate a secret, if needed
var secret = speakeasy.generateSecret();
// By default, generateSecret() returns an otpauth_url for SHA1

// Use otpauthURL() to get a custom authentication URL for SHA512
var url = speakeasy.otpauthURL({ secret: secret.ascii, label: 'Name of Secret', algorithm: 'sha512' });

// Pass URL into a QR code generator
```

Specifying a window for verifying HOTP and TOTP

Verify a HOTP token with counter value 42 and a window of 10. HOTP has a one-sided window, so this will check counter values from 42 to 52, inclusive, and return a `{ delta: n }` where `n` is the difference between the given counter value and the counter position at which the token was found, or `undefined` if it was not found within the window. See the `hotp.verifyDelta(options)` documentation for more info.

```
var token = speakeasy.hotp.verifyDelta({
  secret: secret.ascii,
  counter: 42,
  token: '123456',
  window: 10
});
```

How this works:

```
// Set ASCII secret
var secret = 'rNONHRni6BAk7y2TiKrv';

// Get HOTP counter token at counter = 42
var counter42 = speakeasy.hotp({ secret: secret, counter: 42 });
// => '566646'

// Get HOTP counter token at counter = 45
var counter45 = speakeasy.hotp({ secret: secret, counter: 45 });
// => '323238'

// Verify the secret at counter 42 with the actual value and a window of 10
// This will check all counter values from 42 to 52, inclusive
speakeasy.hotp.verifyDelta({ secret: secret, counter: 42, token: counter42, window: 10 });
// => { delta: 0 } because the given token at counter 42 is 0 steps away from the given counter 42

// Verify the secret at counter 45, but give a counter of 42 and a window of 10
// This will check all counter values from 42 to 52, inclusive
speakeasy.hotp.verifyDelta({ secret: secret, counter: 42, token: counter45, window: 10 });
// => { delta: 3 } because the given token at counter 45 is 0 steps away from given counter 42

// Not in window: specify a window of 1, which only tests counters 42 and 43, not 45
speakeasy.hotp.verifyDelta({ secret: secret, counter: 42, token: counter45, window: 1 });
// => undefined
```

```
// Shortcut to use verify() to simply return whether it is verified as within the window
speakeasy.hotp.verify({ secret: secret, counter: 42, token: counter45, window: 10 });
// => true

// Not in window: specify a window of 1, which only tests counters 42 and 43, not 45
speakeasy.hotp.verify({ secret: secret, counter: 42, token: counter45, window: 1 });
// => false
```

Verify a TOTP token at the current time with a window of 2. Since the default time step is 30 seconds, and TOTP has a two-sided window, this will check tokens between [current time minus two tokens before] and [current time plus two tokens after]. In other words, with a time step of 30 seconds, it will check the token at the current time, plus the tokens at the current time minus 30 seconds, minus 60 seconds, plus 30 seconds, and plus 60 seconds – basically, it will check tokens between a minute ago and a minute from now. It will return a `{ delta: n }` where `n` is the difference between the current time step and the counter position at which the token was found, or `undefined` if it was not found within the window. See the `totp.verifyDelta(options)` documentation for more info.

```
var verified = speakeasy.totp.verifyDelta({
  secret: secret.ascii,
  token: '123456',
  window: 2
});
```

The mechanics of TOTP windows are the same as for HOTP, as shown above, just with two-sided windows, meaning that the `delta` value can be negative if the token is found before the given time or counter.

```
var secret = 'rNONHRni6BAk7y2TiKrv';

// By way of example, we will force TOTP to return tokens at time 1453853945 and
// at time 1453854005 (60 seconds ahead, or 2 steps ahead)
var token1 = speakeasy.totp({ secret: secret, time: 1453853945 }); // 625175
var token3 = speakeasy.totp({ secret: secret, time: 1453854005 }); // 222636
var token2 = speakeasy.totp({ secret: secret, time: 1453854065 }); // 013052

// We can check the time at token 3, 1453853975, with token 1, but use a window of 2
// With a time step of 30 seconds, this will check all tokens from 60 seconds
// before the time to 60 seconds after the time
speakeasy.totp.verifyDelta({ secret: secret, token: token1, window: 2, time: 1453854005 });
// => { delta: -2 }

// token is valid because because token is 60 seconds before time
speakeasy.totp.verify({ secret: secret, token: token1, window: 2, time: 1453854005 });
// => true

// token is valid because because token is 0 seconds before time
speakeasy.totp.verify({ secret: secret, token: token3, window: 2, time: 1453854005 });
// => true

// token is valid because because token is 60 seconds after time
speakeasy.totp.verify({ secret: secret, token: token2, window: 2, time: 1453854005 });
// => true

// This signifies that the given token, token1, is -2 steps away from
// the given time, which means that it is the token for the value at
// (-2 * time step) = (-2 * 30 seconds) = 60 seconds ago.
```

As shown previously, you can also change `verifyDelta()` to `verify()` to simply return a boolean if the given token is within the given window.

Documentation

Full API documentation (in JSDoc format) is available below and at http://speakeasyjs.github.io/speakeasy/

View documentation

Functions

digest(options) ⇒ *Buffer*
Digest the one-time passcode options.

hotp(options) ⇒ *String*
Generate a counter-based one-time token.

hotp.verifyDelta(options) ⇒ *Object*
Verify a counter-based one-time token against the secret and return the delta.

hotp.verify(options) ⇒ *Boolean*
Verify a counter-based one-time token against the secret and return true if it verifies.

totp(options) ⇒ *String*
Generate a time-based one-time token.

totp.verifyDelta(options) ⇒ *Object*
Verify a time-based one-time token against the secret and return the delta.

totp.verify(options) ⇒ *Boolean*
Verify a time-based one-time token against the secret and return true if it verifies.

generateSecret(options) ⇒ *Object* | *GeneratedSecret*
Generates a random secret with the set A-Z a-z 0-9 and symbols, of any length (default 32).

generateSecretASCII([length], [symbols]) ⇒ *String*
Generates a key of a certain length (default 32) from A-Z, a-z, 0-9, and symbols (if requested).

otpauthURL(options) ⇒ *String*
Generate an URL for use with the Google Authenticator app.

Typedefs

GeneratedSecret : *Object*

digest(options) ⇒ Buffer

Digest the one-time passcode options.

Kind: function

Returns: `Buffer` - The one-time passcode as a buffer.

Param	Type	Default	Description
options	`Object`		
options.secret	`String`		Shared secret key
options.counter	`Integer`		Counter value
[options.encoding]	`String`	`"ascii"`	Key encoding (ascii, hex, base32, base64).
[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).
[options.key]	`String`		(DEPRECATED. Use `secret` instead.) Shared secret key

hotp(options) ⇒ String

Generate a counter-based one-time token. Specify the key and counter, and receive the one-time password for that counter position as a string. You can also specify a token length, as well as the encoding (ASCII, hexadecimal, or base32) and the hashing algorithm to use (SHA1, SHA256, SHA512).

Kind: function

Returns: `String` - The one-time passcode.

Param	Type	Default	Description
options	`Object`		

Param	Type	Default	Description
options.secret	`String`		Shared secret key
options.counter	`Integer`		Counter value
[options.digest]	`Buffer`		Digest, automatically generated by default
[options.digits]	`Integer`	`6`	The number of digits for the one-time passcode.
[options.encoding]	`String`	`"ascii"`	Key encoding (ascii, hex, base32, base64).
[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).
[options.key]	`String`		(DEPRECATED. Use `secret` instead.) Shared secret key
[options.length]	`Integer`	`6`	(DEPRECATED. Use `digits` instead.) The number of digits for the one-time passcode.

hotp.verifyDelta(options) ⇒ `Object`

Verify a counter-based one-time token against the secret and return the delta. By default, it verifies the token at the given counter value, with no leeway (no look-ahead or look-behind). A token validated at the current counter value will have a delta of 0.

You can specify a window to add more leeway to the verification process. Setting the window param will check for the token at the given counter value as well as `window` tokens ahead (one-sided window). See param for more info.

`verifyDelta()` will return the delta between the counter value of the token and the given counter value. For example, if given a counter 5 and a window 10, `verifyDelta()` will look at tokens from 5 to 15, inclusive. If it finds it at counter position 7, it will return `{ delta: 2 }`.

Kind: function

Returns: `Object` - On success, returns an object with the counter difference between the client and the server as the `delta` property (i.e. `{ delta: 0 }`).

Param	Type	Default	Description
options	`Object`		
options.secret	`String`		Shared secret key
options.token	`String`		Passcode to validate
options.counter	`Integer`		Counter value. This should be stored by the application and must be incremented for each request.
[options.digits]	`Integer`	`6`	The number of digits for the one-time passcode.
[options.window]	`Integer`	`0`	The allowable margin for the counter. The function will check "W" codes in the future against the provided passcode, e.g. if W = 10, and C = 5, this function will check the passcode against all One Time Passcodes between 5 and 15, inclusive.
[options.encoding]	`String`	`"ascii"`	Key encoding (ascii, hex, base32, base64).

[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).

hotp.verify(options) ⇒ `Boolean`

Verify a counter-based one-time token against the secret and return true if it verifies. Helper function for `hotp.verifyDelta()`` that returns a boolean instead of an object. For more on how to use a window with this, see hotp.verifyDelta.

Kind: function

Returns: `Boolean` - Returns true if the token matches within the given window, false otherwise.

Param	Type	Default	Description
options	`Object`		
options.secret	`String`		Shared secret key
options.token	`String`		Passcode to validate
options.counter	`Integer`		Counter value. This should be stored by the application and must be incremented for each request.
[options.digits]	`Integer`	`6`	The number of digits for the one-time passcode.
[options.window]	`Integer`	`0`	The allowable margin for the counter. The function will check "W" codes in the future against the provided passcode, e.g. if W = 10, and C = 5, this function will check the passcode against all One Time Passcodes between 5 and 15, inclusive.
[options.encoding]	`String`	`"ascii"`	Key encoding (ascii, hex, base32, base64).
[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).

totp(options) ⇒ `String`

Generate a time-based one-time token. Specify the key, and receive the one-time password for that time as a string. By default, it uses the current time and a time step of 30 seconds, so there is a new token every 30 seconds. You may override the time step and epoch for custom timing. You can also specify a token length, as well as the encoding (ASCII, hexadecimal, or base32) and the hashing algorithm to use (SHA1, SHA256, SHA512).

Under the hood, TOTP calculates the counter value by finding how many time steps have passed since the epoch, and calls HOTP with that counter value.

Kind: function

Returns: `String` - The one-time passcode.

Param	Type	Default	Description
options	`Object`		
options.secret	`String`		Shared secret key
[options.time]	`Integer`		Time in seconds with which to calculate counter value. Defaults to `Date.now()`.
[options.step]	`Integer`	`30`	Time step in seconds
[options.epoch]	`Integer`	`0`	Initial time since the UNIX epoch from which to calculate the counter value. Defaults to 0 (no offset).
[options.counter]	`Integer`		Counter value, calculated by default.
[options.digits]	`Integer`	`6`	The number of digits for the one-time passcode.
[options.encoding]	`String`	`"ascii"`	Key encoding (ascii, hex, base32, base64).
[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).
[options.key]	`String`		(DEPRECATED. Use `secret` instead.) Shared secret key
[options.initial_time]	`Integer`	`0`	(DEPRECATED. Use `epoch` instead.) Initial time since the UNIX epoch from which to calculate the counter value. Defaults to 0 (no offset).
[options.length]	`Integer`	`6`	(DEPRECATED. Use `digits` instead.) The number of digits for the one-time passcode.

totp.verifyDelta(options) ⇒ `Object`

Verify a time-based one-time token against the secret and return the delta. By default, it verifies the token at the current time window, with no leeway (no look-ahead or look-behind). A token validated at the current time window will have a delta of 0.

You can specify a window to add more leeway to the verification process. Setting the window param will check for the token at the given counter value as well as `window` tokens ahead and `window` tokens behind (two-sided window). See param for more info.

`verifyDelta()` will return the delta between the counter value of the token and the given counter value. For example, if given a time at counter 1000 and a window of 5, `verifyDelta()` will look at tokens from 995 to 1005, inclusive. In other words, if the time-step is 30 seconds, it will look at tokens from 2.5 minutes ago to 2.5 minutes in the future, inclusive. If it finds it at counter position 1002, it will return `{ delta: 2 }`. If it finds it at counter position 997, it will return `{ delta: -3 }`.

Kind: function

Returns: `Object` - On success, returns an object with the time step difference between the client and the server as the `delta` property (e.g. `{ delta: 0 }`).

Param	Type	Default	Description
options	`Object`		
options.secret	`String`		Shared secret key
options.token	`String`		Passcode to validate
[options.time]	`Integer`		Time in seconds with which to calculate counter value. Defaults to `Date.now()`.
[options.step]	`Integer`	`30`	Time step in seconds
[options.epoch]	`Integer`	`0`	Initial time since the UNIX epoch from which to calculate the counter value. Defaults to 0 (no offset).
[options.counter]	`Integer`		Counter value, calculated by default.
[options.digits]	`Integer`	`6`	The number of digits for the one-time passcode.
[options.window]	`Integer`	`0`	The allowable margin for the counter. The function will check "W" codes in the future and the past against the provided passcode, e.g. if W = 5, and C = 1000, this function will check the passcode against all One Time Passcodes between 995 and 1005, inclusive.
[options.encoding]	`String`	`"ascii"`	Key encoding (ascii, hex, base32, base64).
[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).

totp.verify(options) ⇒ `Boolean`

Verify a time-based one-time token against the secret and return true if it verifies. Helper function for verifyDelta() that returns a boolean instead of an object. For more on how to use a window with this, see totp.verifyDelta.

Kind: function

Returns: `Boolean` - Returns true if the token matches within the given window, false otherwise.

Param	Type	Default	Description
options	`Object`		
options.secret	`String`		Shared secret key
options.token	`String`		Passcode to validate
[options.time]	`Integer`		Time in seconds with which to calculate counter value. Defaults to `Date.now()`.
[options.step]	`Integer`	`30`	Time step in seconds
[options.epoch]	`Integer`	`0`	Initial time since the UNIX epoch from which to calculate the counter value. Defaults to 0 (no offset).
[options.counter]	`Integer`		Counter value, calculated by default.

Param	Type	Default	Description
[options.digits]	`Integer`	`6`	The number of digits for the one-time passcode.
[options.window]	`Integer`	`0`	The allowable margin for the counter. The function will check "W" codes in the future and the past against the provided passcode, e.g. if W = 5, and C = 1000, this function will check the passcode against all One Time Passcodes between 995 and 1005, inclusive.
[options.encoding]	`String`	`"ascii"`	Key encoding (ascii, hex, base32, base64).
[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).

generateSecret(options) ⇒ `Object` | `GeneratedSecret`

Generates a random secret with the set A-Z a-z 0-9 and symbols, of any length (default 32). Returns the secret key in ASCII, hexadecimal, and base32 format, along with the URL used for the QR code for Google Authenticator (an otpauth URL). Use a QR code library to generate a QR code based on the Google Authenticator URL to obtain a QR code you can scan into the app.

Kind: function

Returns: A `GeneratedSecret` object

Param	Type	Default	Description
options	`Object`		
[options.length]	`Integer`	`32`	Length of the secret
[options.symbols]	`Boolean`	`false`	Whether to include symbols
[options.otpauth_url]	`Boolean`	`true`	Whether to output a Google Authenticator-compatible otpauth:// URL (only returns otpauth:// URL, no QR code)
[options.name]	`String`		The name to use with Google Authenticator.
[options.qr_codes]	`Boolean`	`false`	(DEPRECATED. Do not use to prevent leaking of secret to a third party. Use your own QR code implementation.) Output QR code URLs for the token.
[options.google_auth_qr]	`Boolean`	`false`	(DEPRECATED. Do not use to prevent leaking of secret to a third party. Use your own QR code implementation.) Output a Google Authenticator otpauth:// QR code URL.
[options.issuer]	`String`		The provider or service with which the secret key is associated.

generateSecretASCII([length], [symbols]) ⇒ `String`

Generates a key of a certain length (default 32) from A-Z, a-z, 0-9, and symbols (if requested).

Kind: function

Returns: `String` - The generated key.

Param	Type	Default	Description
[length]	`Integer`	`32`	The length of the key.
[symbols]	`Boolean`	`false`	Whether to include symbols in the key.

otpauthURL(options) ⇒ `String`

Generate a Google Authenticator-compatible otpauth:// URL for passing the secret to a mobile device to install the secret.

Authenticator considers TOTP codes valid for 30 seconds. Additionally, the app presents 6 digits codes to the user. According to the documentation, the period and number of digits are currently ignored by the app.

To generate a suitable QR Code, pass the generated URL to a QR Code generator, such as the `qr-image` module.

Kind: function

Throws: Error if secret or label is missing, or if hotp is used and a counter is missing, if the type is not one of `hotp` or `totp`, if the number of digits is non-numeric, or an invalid period is used. Warns if the number of digits is not either 6 or 8 (though 6 is the only one supported by Google Authenticator), and if the hashihng algorithm is not one of the supported SHA1, SHA256, or SHA512.

Returns: `String` - A URL suitable for use with the Google Authenticator.

See: https://github.com/google/google-authenticator/wiki/Key-Uri-Format

Param	Type	Default	Description
options	`Object`		
options.secret	`String`		Shared secret key
options.label	`String`		Used to identify the account with which the secret key is associated, e.g. the user's email address.
[options.type]	`String`	`"totp"`	Either "hotp" or "totp".
[options.counter]	`Integer`		The initial counter value, required for HOTP.
[options.issuer]	`String`		The provider or service with which the secret key is associated.
[options.algorithm]	`String`	`"sha1"`	Hash algorithm (sha1, sha256, sha512).
[options.digits]	`Integer`	`6`	The number of digits for the one-time passcode. Currently ignored by Google Authenticator.
[options.period]	`Integer`	`30`	The length of time for which a TOTP code will be valid, in seconds. Currently ignored by Google Authenticator.
[options.encoding]	`String`	`ascii`	Key encoding (ascii, hex, base32, base64). If the key is not encoded in Base-32, it will be reencoded.

GeneratedSecret : `Object`

Kind: global typedef

Properties

Name	Type	Description
ascii	`String`	ASCII representation of the secret
hex	`String`	Hex representation of the secret
base32	`String`	Base32 representation of the secret
qr_code_ascii	`String`	URL for the QR code for the ASCII secret.
qr_code_hex	`String`	URL for the QR code for the hex secret.
qr_code_base32	`String`	URL for the QR code for the base32 secret.
google_auth_qr	`String`	URL for the Google Authenticator otpauth URL's QR code.
otpauth_url	`String`	Google Authenticator-compatible otpauth URL.

Contributing

We're very happy to have your contributions in Speakeasy.

Contributing code — First, make sure you've added tests if adding new functionality. Then, run `npm test` to run all the tests to make sure they pass. Next, make a pull request to this repo. Thanks!

Filing an issue — Submit issues to the GitHub Issues page.

Maintainers —

- Mark Bao (markbao)
- Michael Phan-Ba (mikepb)

License

This project incorporates code from passcode, which was originally a fork of speakeasy, and notp, both of which are licensed under MIT. Please see the LICENSE file for the full combined license.

Icons created by Gregor Črešnar, iconoci, and Danny Sturgess from the Noun Project.