QR Code Generator

This package is a node.js implementation of a QR code generator. It is designed for use in a server-side node.js environment, but it can be tested and used client side with the qrcode binary that comes with the npm install.

Original source code is from https://github.com/soldair/node-qrcode.

There is sample code in the /examples folder of the source code, and further information on usage is found in the README.md file, the contents of which are enclosed here for convenience.

node-qrcode

QR code/2d barcode generator.

Highlights

- Works on server and client (and react native with svg)
- CLI utility

- Save QR code as image
- Support for Numeric, Alphanumeric, Kanji and Byte mode
- Support for mixed modes
- Support for chinese, cyrillic, greek and japanese characters
- Support for multibyte characters (like emojis 😁)
- Auto generates optimized segments for best data compression and smallest QR Code size
- App agnostic readability, QR Codes by definition are app agnostic

Installation

Inside your project folder do:

```
npm install --save qrcode
```

or, install it globally to use `qrcode` from the command line to save qrcode images or generate ones you can view in your terminal.

```
npm install -g qrcode
```

Usage

CLI

```
Usage: qrcode [options] <input string>




QR Code options:

  -v, --qversion  QR Code symbol version (1 - 40)                      [number]

  -e, --error     Error correction level           [choices: "L", "M", "Q", "H"]

  -m, --mask      Mask pattern (0 - 7)                                 [number]



Renderer options:
```

```
  -t, --type        Output type                  [choices: "png", "svg", "utf8"]

  -w, --width       Image width (px)                                    [number]

  -s, --scale       Scale factor                                        [number]

  -q, --qzone       Quiet zone size                                     [number]

  -l, --lightcolor  Light RGBA hex color

  -d, --darkcolor   Dark RGBA hex color

  --small  Output smaller QR code to terminal                          [boolean]



Options:

  -o, --output  Output file

  -h, --help    Show help                                              [boolean]

  --version     Show version number                                    [boolean]



Examples:

  qrcode "some text"                   Draw in terminal window

  qrcode -o out.png "some text"        Save as png image

  qrcode -d F00 -o out.png "some text"  Use red as foreground color
```

If not specified, output type is guessed from file extension.
Recognized extensions are `png`, `svg` and `txt`.

Browser

`node-qrcode` can be used in browser through module bundlers like **Browserify** and **Webpack** or by including the precompiled bundle present in `build/` folder.

Module bundlers

```
<!-- index.html -->

<html>

  <body>

    <canvas id="canvas"></canvas>

    <script src="bundle.js"></script>

  </body>

</html>

// index.js -> bundle.js

var QRCode = require('qrcode')

var canvas = document.getElementById('canvas')



QRCode.toCanvas(canvas, 'sample text', function (error) {

  if (error) console.error(error)

  console.log('success!');

})
```

Precompiled bundle

```
<canvas id="canvas"></canvas>
```

```
<script src="/build/qrcode.js"></script>

<script>

  QRCode.toCanvas(document.getElementById('canvas'), 'sample
text', function (error) {

    if (error) console.error(error)

    console.log('success!');

  })

</script>
```

If you install through `npm`, precompiled files will be available in `node_modules/qrcode/build/` folder.

The precompiled bundle have support for **Internet Explorer 10+, Safari 5.1+, and all evergreen browsers**.

NodeJS

Require the module `qrcode`

```
var QRCode = require('qrcode')



QRCode.toDataURL('I am a pony!', function (err, url) {

  console.log(url)

})
```

render a qrcode for the terminal

```
var QRCode = require('qrcode')



QRCode.toString('I am a pony!',{type:'terminal'}, function (err,
url) {
```

```
  console.log(url)

})
```

ES6/ES7

Promises and Async/Await can be used in place of callback function.

```
import QRCode from 'qrcode'



// With promises

QRCode.toDataURL('I am a pony!')

  .then(url => {

    console.log(url)

  })

  .catch(err => {

    console.error(err)

  })



// With async/await

const generateQR = async text => {

  try {

    console.log(await QRCode.toDataURL(text))

  } catch (err) {

    console.error(err)
```

```
  }

}
```

Error correction level

Error correction capability allows to successfully scan a QR Code even if the symbol is dirty or damaged. Four levels are available to choose according to the operating environment.

Higher levels offer a better error resistance but reduce the symbol's capacity. If the chances that the QR Code symbol may be corrupted are low (for example if it is showed through a monitor) is possible to safely use a low error level such as `Low` or `Medium`.

Possible levels are shown below:

Level	Error resistance
L (Low)	**~7%**
M (Medium)	**~15%**
Q (Quartile)	**~25%**
H (High)	**~30%**

The percentage indicates the maximum amount of damaged surface after which the symbol becomes unreadable.

Error level can be set through `options.errorCorrectionLevel` property.
If not specified, the default value is `M`.

```
QRCode.toDataURL('some text', { errorCorrectionLevel: 'H' },
function (err, url) {

  console.log(url)

})
```

QR Code capacity

Capacity depends on symbol version and error correction level. Also encoding modes may influence the amount of storable data.

The QR Code versions range from version **1** to version **40**.
Each version has a different number of modules (black and white dots), which define the symbol's size. For version 1 they are `21x21`, for version 2 `25x25` e so on. Higher is the version, more are the storable data, and of course bigger will be the QR Code symbol.

The table below shows the maximum number of storable characters in each encoding mode and for each error correction level.

Mode	L	M	Q	H
Numeric	7089	5596	3993	3057
Alphanumeric	4296	3391	2420	1852
Byte	2953	2331	1663	1273
Kanji	1817	1435	1024	784

Note: Maximum characters number can be different when using **Mixed modes**.

QR Code version can be set through `options.version` property.
If no version is specified, the more suitable value will be used. Unless a specific version is required, this option is not needed.

```
QRCode.toDataURL('some text', { version: 2 }, function (err, url) {

  console.log(url)

})
```

Encoding modes

Modes can be used to encode a string in a more efficient way.
A mode may be more suitable than others depending on the string content. A list of supported modes are shown in the table below:

Mode	Characters	Compression
Numeric	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	3 characters are represented by 10 bits
Alphanumeric	0–9, A–Z (upper-case only), space, $, %, *, +, -, ., /, :	2 characters are represented by 11 bits
Kanji	Characters from the Shift JIS system based on JIS X 0208	2 kanji are represented by 13 bits
Byte	Characters from the ISO/IEC 8859-1 character set	Each characters are represented by 8 bits

Choose the right mode may be tricky if the input text is unknown.
In these cases **Byte** mode is the best choice since all characters can be encoded with it. (See **Multibyte characters**)
However, if the QR Code reader supports mixed modes, using **Auto mode** may produce better results.

Mixed modes

Mixed modes are also possible. A QR code can be generated from a series of segments having different encoding modes to optimize the data compression.
However, switching from a mode to another has a cost which may lead to a worst result if it's not taken into account. See **Manual mode** for an example of how to specify segments with different encoding modes.

Auto mode

By **default**, automatic mode selection is used.
The input string is automatically splitted in various segments optimized to produce the shortest possible bitstream using mixed modes.
This is the preferred way to generate the QR Code.

For example, the string **ABCDE12345678?A1A** will be splitted in 3 segments with the following modes:

Segment	Mode
ABCDE	Alphanumeric
12345678	Numeric
?A1A	Byte

Any other combinations of segments and modes will result in a longer bitstream.
If you need to keep the QR Code size small, this mode will produce the best results.

Manual mode

If auto mode doesn't work for you or you have specific needs, is also possible to manually specify each segment with the relative mode. In this way no segment optimizations will be applied under the hood.
Segments list can be passed as an array of object:

```
var QRCode = require('qrcode')



var segs = [

  { data: 'ABCDEFG', mode: 'alphanumeric' },

  { data: '0123456', mode: 'numeric' }

]



QRCode.toDataURL(segs, function (err, url) {

  console.log(url)
```

```
  })
```

Kanji mode

With kanji mode is possible to encode characters from the Shift JIS system in an optimized way.
Unfortunately, there isn't a way to calculate a Shifted JIS values from, for example, a character encoded in UTF-8, for this reason a conversion table from the input characters to the SJIS values is needed.
This table is not included by default in the bundle to keep the size as small as possible.

If your application requires kanji support, you will need to pass a function that will take care of converting the input characters to appropriate values.

An helper method is provided by the lib through an optional file that you can include as shown in the example below.

Note: Support for Kanji mode is only needed if you want to benefit of the data compression, otherwise is still possible to encode kanji using Byte mode (See **Multibyte characters**).

```
  var QRCode = require('qrcode')

  var toSJIS = require('qrcode/helper/to-sjis')



  QRCode.toDataURL(kanjiString, { toSJISFunc: toSJIS }, function
(err, url) {

    console.log(url)

  })
```

With precompiled bundle:

```
<canvas id="canvas"></canvas>



<script src="/build/qrcode.min.js"></script>
```

```
<script src="/build/qrcode.tosjis.min.js"></script>

<script>

  QRCode.toCanvas(document.getElementById('canvas'),

    'sample text', { toSJISFunc: QRCode.toSJIS }, function
(error) {

    if (error) console.error(error)

    console.log('success!')

  })

</script>
```

Binary data

QR Codes can hold arbitrary byte-based binary data. If you attempt to create a binary QR Code by first converting the data to a JavaScript string, it will fail to encode propery because string encoding adds additional bytes. Instead, you must pass a `Uint8ClampedArray` or compatible array, or a Node **Buffer**, as follows:

```
// Regular array example

// WARNING: Element values will be clamped to 0-255 even if your
data contains higher values.

const QRCode = require('qrcode')

QRCode.toFile(

  'foo.png',

  [{ data: [253,254,255], mode: 'byte' }],

  ...options...,

  ...callback...
```

```
)

// Uint8ClampedArray example

const QRCode = require('qrcode')



QRCode.toFile(

  'foo.png',

  [{ data: new Uint8ClampedArray([253,254,255]), mode: 'byte'
}],

  ...options...,

  ...callback...

)

// Node Buffer example

// WARNING: Element values will be clamped to 0-255 even if your
data contains higher values.

const QRCode = require('qrcode')



QRCode.toFile(

  'foo.png',

  [{ data: Buffer.from([253,254,255]), mode: 'byte' }],

  ...options...,

  ...callback...

)
```

TypeScript users: if you are using **@types/qrcode**, you will need to add a `// @ts-ignore` above the data segment because it expects `data: string`.

Multibyte characters

Support for multibyte characters isn't present in the initial QR Code standard, but is possible to encode UTF-8 characters in Byte mode.

QR Codes provide a way to specify a different type of character set through ECI (Extended Channel Interpretation), but it's not fully implemented in this lib yet.

Most QR Code readers, however, are able to recognize multibyte characters even without ECI.

Note that a single Kanji/Kana or Emoji can take up to 4 bytes.

API

Browser:

- create()
- toCanvas()
- toDataURL()
- toString()

Server:

- create()
- toCanvas()
- toDataURL()
- toString()
- toFile()
- toFileStream()

Browser API

`create(text, [options])`

Creates QR Code symbol and returns a qrcode object.

`text`

Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`
See **QR Code options**.

`returns`
Type: `Object`

```
// QRCode object

{

  modules,                // Bitmatrix class with modules data

  version,                // Calculated QR Code version

  errorCorrectionLevel,   // Error Correction Level

  maskPattern,            // Calculated Mask pattern

  segments                // Generated segments

}
```

`toCanvas(canvasElement, text, [options], [cb(error)])`

`toCanvas(text, [options], [cb(error, canvas)])`

Draws qr code symbol to canvas.
If `canvasElement` is omitted a new canvas is returned.

`canvasElement`
Type: `DOMElement`

Canvas where to draw QR Code.

`text`
Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`

See **Options**.

`cb`

Type: `Function`

Callback function called on finish.

Example

```
QRCode.toCanvas('text', { errorCorrectionLevel: 'H' }, function
(err, canvas) {

  if (err) throw err



  var container = document.getElementById('container')

  container.appendChild(canvas)

})
```

`toDataURL(text, [options], [cb(error, url)])`

`toDataURL(canvasElement, text, [options], [cb(error, url)])`

Returns a Data URI containing a representation of the QR Code image.
If provided, `canvasElement` will be used as canvas to generate the data URI.

`canvasElement`

Type: `DOMElement`

Canvas where to draw QR Code.

`text`

Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`

- **`type`**
  Type: `String`
  Default: `image/png`

  Data URI format.
  Possible values are: `image/png`, `image/jpeg`, `image/webp`.

- **`rendererOpts.quality`**
  Type: `Number`
  Default: `0.92`

  A Number between `0` and `1` indicating image quality if the requested type is `image/jpeg` or `image/webp`.

See **Options** for other settings.

`cb`
Type: `Function`

Callback function called on finish.

Example

```
var opts = {

  errorCorrectionLevel: 'H',

  type: 'image/jpeg',

  quality: 0.3,

  margin: 1,

  color: {

    dark:"#010599FF",

    light:"#FFBF60FF"

  }

}
```

```
QRCode.toDataURL('text', opts, function (err, url) {

  if (err) throw err



  var img = document.getElementById('image')

  img.src = url

})
```

`toString(text, [options], [cb(error, string)])`

Returns a string representation of the QR Code.

`text`

Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`

- **`type`**
  Type: `String`
  Default: `utf8`

  Output format.
  Possible values are: `terminal`,`utf8`, and `svg`.

See **Options** for other settings.

`cb`

Type: `Function`

Callback function called on finish.

Example

```
QRCode.toString('http://www.google.com', function (err, string)
{

  if (err) throw err
```

```
  console.log(string)

})
```

Server API

`create(text, [options])`

See **create**.

`toCanvas(canvas, text, [options], [cb(error)])`

Draws qr code symbol to **node canvas**.

`text`
Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`
See **Options**.

`cb`
Type: `Function`

Callback function called on finish.

`toDataURL(text, [options], [cb(error, url)])`

Returns a Data URI containing a representation of the QR Code image.
Only works with `image/png` type for now.

`text`
Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`

See **Options** for other settings.

`cb`
Type: `Function`

Callback function called on finish.

`toString(text, [options], [cb(error, string)])`

Returns a string representation of the QR Code.
If choosen output format is `svg` it will returns a string containing xml code.

`text`
Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`

- **`type`**
  Type: `String`
  Default: `utf8`

  Output format.
  Possible values are: `utf8`, `svg`, `terminal`.

See **Options** for other settings.

`cb`
Type: `Function`

Callback function called on finish.

Example

```
QRCode.toString('http://www.google.com', function (err, string)
{

  if (err) throw err

  console.log(string)

})
```

`toFile(path, text, [options], [cb(error)])`

Saves QR Code to image file.
If `options.type` is not specified, the format will be guessed from file extension.
Recognized extensions are `png`, `svg`, `txt`.

`path`

Type: `String`

Path where to save the file.

`text`

Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`

- **`type`**
  Type: `String`
  Default: `png`

  Output format.
  Possible values are: `png`, `svg`, `utf8`.

- **`rendererOpts.deflateLevel`** (png only)
  Type: `Number`
  Default: `9`

  Compression level for deflate.

- **`rendererOpts.deflateStrategy`** (png only)
  Type: `Number`
  Default: `3`

  Compression strategy for deflate.

See **Options** for other settings.

`cb`

Type: `Function`

Callback function called on finish.

Example

```
QRCode.toFile('path/to/filename.png', 'Some text', {
  color: {
    dark: '#00F',  // Blue dots
    light: '#0000' // Transparent background
  }
}, function (err) {
  if (err) throw err
  console.log('done')
})
```

`toFileStream(stream, text, [options])`

Writes QR Code image to stream. Only works with `png` format for now.

`stream`
Type: `stream.Writable`

Node stream.

`text`
Type: `String|Array`

Text to encode or a list of objects describing segments.

`options`
See **Options**.

Options

QR Code options

`version`
Type: `Number`

QR Code version. If not specified the more suitable value will be calculated.

`errorCorrectionLevel`
Type: `String`
Default: `M`

Error correction level.
Possible values are `low, medium, quartile, high` or `L, M, Q, H`.

`maskPattern`
Type: `Number`

Mask pattern used to mask the symbol.
Possible values are `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`.
If not specified the more suitable value will be calculated.

`toSJISFunc`
Type: `Function`

Helper function used internally to convert a kanji to its Shift JIS value.
Provide this function if you need support for Kanji mode.

Renderers options

`margin`
Type: `Number`
Default: `4`

Define how much wide the quiet zone should be.

`scale`
Type: `Number`
Default: `4`

Scale factor. A value of `1` means 1px per modules (black dots).

`small`
Type: `Boolean`
Default: `false`

Relevant only for terminal renderer. Outputs smaller QR code.

`width`

Type: `Number`

Forces a specific width for the output image.
If width is too small to contain the qr symbol, this option will be ignored.
Takes precedence over `scale`.

`color.dark`

Type: `String`
Default: `#000000ff`

Color of dark module. Value must be in hex format (RGBA).
Note: dark color should always be darker than `color.light`.

`color.light`

Type: `String`
Default: `#ffffffff`

Color of light module. Value must be in hex format (RGBA).

GS1 QR Codes

There was a real good discussion here about them. but in short any qrcode generator will make gs1 compatible qrcodes, but what defines a gs1 qrcode is a header with metadata that describes your gs1 information.

https://github.com/soldair/node-qrcode/issues/45

Credits

This lib is based on "QRCode for JavaScript" which Kazuhiko Arase thankfully MIT licensed.

License

MIT

The word "QR Code" is registered trademark of:
DENSO WAVE INCORPORATED