Picomatch

Blazing fast and accurate glob matcher written in JavaScript.

No dependencies and full support for standard and extended Bash glob features, including braces, extglobs, POSIX brackets, and regular expressions.

Why picomatch?

Lightweight - No dependencies

Minimal - Tiny API surface. Main export is a function that takes a glob pattern and returns a matcher function.

Fast - Loads in about 2ms (that's several times faster than a single frame of a HD movie at 60fps)

Performant - Use the returned matcher function to speed up repeat matching (like when watching files)

Accurate matching - Using wildcards (* and ?), globstars (**) for nested directories, advanced globbing with extglobs, braces, and POSIX brackets, and support for escaping special characters with \ or quotes.

Well tested - Thousands of unit tests

See the library comparison to other libraries.

Table of Contents

Click to expand

Install

Install with npm:

```
npm install --save picomatch
```

Usage

The main export is a function that takes a glob pattern and an options object and returns a function for matching strings.

```
const pm = require('picomatch');
const isMatch = pm('*.js');
```



```
npm install && npm test
```

Building docs

(This project's readme.md is generated by verb, please don't edit the readme directly. Any changes to the readme must be made in the .verb.md readme template.)

To generate the readme, run the following command:

```
npm install -g verbose/verb#dev verb-generate-readme && verb
```

Author

Jon Schlinkert

- GitHub Profile
- Twitter Profile
- LinkedIn Profile

License

extglobs are combined with globstars, braces, slashes, and so on: `!(**/{a,b,*/c})`.

Thus, given that there is no canonical glob specification to use as a single source of truth when differences of opinion arise regarding behavior, sometimes we have to implement our best judgement and rely on feedback from users to make improvements.

Performance

Although this library performs well in benchmarks, and in most cases it's faster than other popular libraries we benchmarked against, we will always choose accuracy over performance. It's not helpful to anyone if our library is faster at returning the wrong answer.

About

Contributing

Pull requests and stars are always welcome. For bugs and feature requests, please create an issue.

Please read the contributing guide for advice on opening issues, pull requests, and coding standards.

Running Tests

Running and reviewing unit tests is a great way to get familiarized with a library and its API. You can install dependencies and run tests with the following command:


```
console.log(isMatch('abcd')); //=> false
console.log(isMatch('a.js')); //=> true
console.log(isMatch('a.md')); //=> false
console.log(isMatch('a/b.js')); //=>
        false
```

API

picomatch

Creates a matcher function from one or more glob patterns. The returned function takes a string to match as its first argument, and returns true if the string is a match. The returned matcher function also takes a boolean as the second argument that, when true, returns an object with additional information.

Params

- `globs` **{String|Array}**: One or more glob patterns.
- `options` **{Object=}**
- `returns` **{Function=}**: Returns a matcher function.

Example

```
const picomatch = require('picomatch');
// picomatch(glob[, options]);

const isMatch = picomatch('*.!(*a)');
```

```
console.log(isMatch('a.a')); //=> false
console.log(isMatch('a.b')); //=> true
```

Example without node.js

For environments without `node.js`, `picomatch/posix` provides you a dependency-free matcher, without automatic OS detection.

```
const picomatch = require('picomatch/
        posix');
// the same API, defaulting to posix
        paths
const isMatch = picomatch('a/*');
console.log(isMatch('a\\b')); //=> false
console.log(isMatch('a/b')); //=> true

// you can still configure the matcher
        function to accept windows paths
const isMatch = picomatch('a/*', {
        options: windows });
console.log(isMatch('a\\b')); //=> true
console.log(isMatch('a/b')); //=> true
```

.test

Test `input` with the given `regex`. This is used by the main `picomatch()` function to test the input string.

Params

- `input` **{String}**: String to test.
- `regex` **{RegExp}**


```
  minimatch x 14,299 ops/sec ±0.26% (96
runs sampled)

# .makeRe - medium ranges ({1..100000}
*.txt)
  picomatch x 395,020 ops/sec ±0.87% (89
runs sampled)
  minimatch x 2 ops/sec ±4.59% (10 runs
sampled)

# .makeRe - long ranges ({1..10000000}
*.txt)
  picomatch x 400,036 ops/sec ±0.83% (90
runs sampled)
  minimatch (FATAL ERROR: Ineffective
mark-compacts near heap limit Allocation
failed - JavaScript heap out of memory)
```

Philosophies

The goal of this library is to be blazing fast, without compromising on accuracy.

Accuracy

The number one of goal of this library is accuracy. However, it's not unusual for different glob implementations to have different rules for matching behavior, even with simple wildcard matching. It gets increasingly more complicated when combinations of different features are combined, like when

```
(97 runs sampled)
  minimatch x 632,772 ops/sec ±0.14% (98
runs sampled)

# .makeRe star; dot=true (*)
  picomatch x 3,500,079 ops/sec ±0.26%
(99 runs sampled)
  minimatch x 564,916 ops/sec ±0.23% (96
runs sampled)

# .makeRe globstar (**)
  picomatch x 3,261,000 ops/sec ±0.27%
(98 runs sampled)
  minimatch x 1,664,766 ops/sec ±0.20%
(100 runs sampled)

# .makeRe globstars (**/**/**)
  picomatch x 3,284,469 ops/sec ±0.18%
(97 runs sampled)
  minimatch x 1,435,880 ops/sec ±0.34%
(95 runs sampled)

# .makeRe with leading star (*.txt)
  picomatch x 3,100,197 ops/sec ±0.35%
(99 runs sampled)
  minimatch x 428,347 ops/sec ±0.42% (94
runs sampled)

# .makeRe - basic braces ({a,b,c}*.txt)
  picomatch x 443,578 ops/sec ±1.33% (89
runs sampled)
  minimatch x 107,143 ops/sec ±0.35% (94
runs sampled)

# .makeRe - short ranges ({a..z}*.txt)
  picomatch x 415,484 ops/sec ±0.76% (96
runs sampled)
```


- `returns` **{Object}**: Returns an object with matching info.

Example

```
const picomatch = require('picomatch');
// picomatch.test(input, regex[,
        options]);

console.log(picomatch.test('foo/bar', /
        ^(?:([^/]*?)\/([^/]*?))$/));
// { isMatch: true, match: [ 'foo/',
        'foo', 'bar' ], output: 'foo/
        bar' }
```

.matchBase

Match the basename of a filepath.

Params

- `input` **{String}**: String to test.
- `glob` **{RegExp|String}**: Glob pattern or regex created by .makeRe.
- `returns` **{Boolean}**

Example

```
const picomatch = require('picomatch');
// picomatch.matchBase(input, glob[,
        options]);
console.log(picomatch.matchBase('foo/
        bar.js', '*.js'); // true
```

.isMatch

Returns true if **any** of the given glob `patterns` match the specified `string`.

Params

- **{String|Array}**: str The string to test.
- **{String|Array}**: patterns One or more glob patterns to use for matching.
- **{Object}**: See available options.
- `returns` **{Boolean}**: Returns true if any patterns match `str`

Example

```
const picomatch = require('picomatch');
// picomatch.isMatch(string, patterns[,
        options]);

console.log(picomatch.isMatch('a.a',
        ['b.*', '*.a'])); //=> true
console.log(picomatch.isMatch('a.a',
        'b.*')); //=> false
```

.parse

Parse a glob pattern to create the source string for a regular expression.

Params

- `pattern` **{String}**
- `options` **{Object}**

Feature	`minimatch`	`micromatch`	`picomatch`	`nanomatch`	e
Wildcard matching (*?+)					
Advancing globbing	✔	✔	✔	-	-
Brace *matching*	✔	✔	✔	-	-
Brace *expansion*	✔	✔	-	-	-
Extglobs	partial	✔	✔	-	✔
Posix brackets	-	✔	✔	-	-
Regular expression syntax	-	✔	✔	✔	✔
File system operations	-	-	-	-	-

Benchmarks

Performance comparison of picomatch and minimatch.

(Pay special attention to the last three benchmarks. Minimatch freezes on long ranges.)

```
# .makeRe star (*)
  picomatch x 4,449,159 ops/sec ±0.24%
```

Matching special characters as literals

If you wish to match the following special characters in a filepath, and you want to use these characters in your glob pattern, they must be escaped with backslashes or quotes:

Special Characters

Some characters that are used for matching in regular expressions are also regarded as valid file path characters on some platforms.

To match any of the following characters as literals: `$^*+?()[]

Examples:

```
console.log(pm.makeRe('foo/bar \\(1\
        \)'));
console.log(pm.makeRe('foo/bar \\(1\
        \)'));
```

Library Comparisons

The following table shows which features are supported by minimatch, micromatch, picomatch, nanomatch, extglob, braces, and expand-brackets.

Feature	minimatch	micromatch	picomatch	nanomatch	extgl
	✔	✔	✔	✔	-

- `returns` **{Object}**: Returns an object with useful properties and output to be used as a regex source string.

Example

```
const picomatch = require('picomatch');
const result =
        picomatch.parse(pattern[,
        options]);
```

.scan

Scan a glob pattern to separate the pattern into segments.

Params

- `input` **{String}**: Glob pattern to scan.
- `options` **{Object}**
- `returns` **{Object}**: Returns an object with

Example

```
const picomatch = require('picomatch');
// picomatch.scan(input[, options]);

const result = picomatch.scan('!./foo/
        *.js');
console.log(result);
{ prefix: '!./',
  input: '!./foo/*.js',
  start: 3,
  base: 'foo',
  glob: '*.js',
  isBrace: false,
  isBracket: false,
```

```
  isGlob: true,
  isExtglob: false,
  isGlobstar: false,
  negated: true }
```

.compileRe

Compile a regular expression from the `state` object returned by the parse() method.

Params

- `state` **{Object}**
- `options` **{Object}**
- `returnOutput` **{Boolean}**: Intended for implementors, this argument allows you to return the raw output from the parser.
- `returnState` **{Boolean}**: Adds the state to a `state` property on the returned regex. Useful for implementors and debugging.
- `returns` **{RegExp}**

.makeRe

Create a regular expression from a parsed glob pattern.

Params

- `state` **{String}**: The object returned from the `.parse` method.
- `options` **{Object}**

- `[:graph:]` - Graph characters, equivalent to `[\\x21-\\x7E]`.
- `[:lower:]` - Lowercase letters, equivalent to `[a-z]`.
- `[:print:]` - Print characters, equivalent to `[\\x20-\\x7E ]`.
- `[:punct:]` - Punctuation and symbols, equivalent to `[\\-!"#$%&\'()\\*+,./:;<=>?@[\\]^_{|}~]`.
- `[:space:]` - Extended space characters, equivalent to `[ \\t\\r\\n\\v\\f]`.
- `[:upper:]` - Uppercase letters, equivalent to `[A-Z]`.
- `[:word:]` - Word characters (letters, numbers and underscores), equivalent to `[A-Za-z0-9_]`.
- `[:xdigit:]` - Hexadecimal digits, equivalent to `[A-Fa-f0-9]`.

See the Bash Reference Manual for more information.

Braces

Picomatch does not do brace expansion. For brace expansion and advanced matching with braces, use micromatch instead. Picomatch has very basic support for braces.

```
// supports multiple extglobs
console.log(pm.isMatch('foo.bar', '!
        (foo).!(bar)')); // false

// supports nested extglobs
console.log(pm.isMatch('foo.bar', '!(!
        (foo)).!(!(bar))')); // true
```

POSIX brackets

POSIX classes are disabled by default. Enable this feature by setting the `posix` option to true.

Enable POSIX bracket support

```
console.log(pm.makeRe('[[:word:]]+', {
        posix: true }));
//=> /^(?:(?=.)[A-Za-z0-9_]+\/?)$/
```

Supported POSIX classes

The following named POSIX bracket expressions are supported:

- `[:alnum:]` - Alphanumeric characters, equ `[a-zA-Z0-9]`
- `[:alpha:]` - Alphabetical characters, equivalent to `[a-zA-Z]`.
- `[:ascii:]` - ASCII characters, equivalent to `[\\x00-\\x7F]`.
- `[:blank:]` - Space and tab characters, equivalent to `[ \t]`.
- `[:cntrl:]` - Control characters, equivalent to `[\\x00-\\x1F\\x7F]`.
- `[:digit:]` - Numerical digits, equivalent to `[0-9]`.

- `returnOutput` **{Boolean}**: Implementors may use this argument to return the compiled output, instead of a regular expression. This is not exposed on the options to prevent end-users from mutating the result.
- `returnState` **{Boolean}**: Implementors may use this argument to return the state from the parsed glob with the returned regular expression.
- `returns` **{RegExp}**: Returns a regex created from the given pattern.

Example

```
const picomatch = require('picomatch');
const state = picomatch.parse('*.js');
// picomatch.compileRe(state[,
        options]);

console.log(picomatch.compileRe(state));
//=> /^(?:(?!\.)(?=.)[^/]*?\.js)$/
```

.toRegex

Create a regular expression from the given regex source string.

Params

- `source` **{String}**: Regular expression source string.
- `options` **{Object}**
- `returns` **{RegExp}**

Example

```
const picomatch = require('picomatch');
// picomatch.toRegex(source[, options]);

const { output } =
        picomatch.parse('*.js');
console.log(picomatch.toRegex(output));
//=> /^(?:(?!\.)(?=.)[^/]*?\.js)$/
```

Options

Picomatch options

The following options may be used with the main `picomatch()` function or any of the methods on the picomatch API.

Option	Type	Default value	Description
`basename`	`boolean`	`false`	If set, then patterns without slashes will b matched against the basename of the path contains slashes. For example, `a?b` would match the path `/xyz 123/acb`, but not `/ xyz/acb/123`.
`bash`	`boolean`	`false`	Follow bash matchin rules more strictly -

Pattern	Description
`*(pattern)`	Match *zero or more* consecutive occurrences of `pattern`
`+(pattern)`	Match *one or more* consecutive occurrences of `pattern`
`?(pattern)`	Match *zero or **one*** consecutive occurrences of `pattern`
`!(pattern)`	Match *anything but* `pattern`

Examples

```
const pm = require('picomatch');

// *(pattern) matches ZERO or more of
        "pattern"
console.log(pm.isMatch('a',
        'a*(z)')); // true
console.log(pm.isMatch('az',
        'a*(z)')); // true
console.log(pm.isMatch('azzz',
        'a*(z)')); // true

// +(pattern) matches ONE or more of
        "pattern"
console.log(pm.isMatch('a', 'a+
        (z)')); // false
console.log(pm.isMatch('az', 'a+
        (z)')); // true
console.log(pm.isMatch('azzz', 'a+
        (z)')); // true
```

Character	Description
	match the characters `a`, `b` or `c`, and nothing else.

Matching behavior vs. Bash

Picomatch's matching features and expected results in unit tests are based on Bash's unit tests and the Bash 4.3 specification, with the following exceptions:

- Bash will match `foo/bar/baz` with `*`. Picomatch only matches nested directories with `**`.
- Bash greedily matches with negated extglobs. For example, Bash 4.3 says that `!(foo)*` should match `foo` and `foobar`, since the trailing `*` bracktracks to match the preceding pattern. This is very memory-inefficient, and IMHO, also incorrect. Picomatch would return `false` for both `foo` and `foobar`.

Advanced globbing

- extglobs
- POSIX brackets
- Braces

Extglobs

Pattern	Description
`@(pattern)`	Match *only one* consecutive occurrence of `pattern`

Option	Type	Default value	Description
			disallows back escape characte treats single sta globstars (`**`).
`capture`	`boolean`	`undefined`	Return regex m supporting met
`contains`	`boolean`	`undefined`	Allows glob to any part of the string(s).
`cwd`	`string`	`process.cwd()`	Current workin directory. Used `picomatch.`
`debug`	`boolean`	`undefined`	Debug regular expressions wh error is thrown
`dot`	`boolean`	`false`	Enable dotfile m By default, dot ignored unless explicitly defin pattern, or `options.do`
`expandRange`	`function`	`undefined`	Custom functio expanding rang brace patterns, `{a..z}`. The receives the ra as two argume must return a st used in the gen regex. It's reco that returned st wrapped in par
`failglob`	`boolean`	`false`	Throws an erro matches are fou

Option	Type	Default value	Description
			Based on the bash op of the same name.
`fastpaths`	`boolean`	`true`	To speed up processi full parsing is skippe for a handful commo glob patterns. Disabl this behavior by setti this option to `false`
`flags`	`string`	`undefined`	Regex flags to use in generated regex. If defined, the `nocase` option will be overridden.
format	`function`	`undefined`	Custom function for formatting the return string. This is useful removing leading slashes, converting Windows paths to Po paths, etc.
`ignore`	`array|string`	`undefined`	One or more glob patterns for excludin strings that should nc matched from the res
`keepQuotes`	`boolean`	`false`	Retain quotes in the generated regex, sinc quotes may also be u as an alternative to backslashes.
`literalBrackets`	`boolean`	`undefined`	When `true`, bracket the glob pattern will escaped so that only literal brackets will b matched.
`matchBase`	`boolean`	`false`	Alias for `basename`

Basic globbing

Character	Description
`*`	Matches any character zero or more times, excluding path separators. Does *not match* path separators or hidden files or directories (“dotfiles”), unless explicitly enabled by setting the `dot` option to `true`.
`**`	Matches any character zero or more times, including path separators. Note that `**` will only match path separators (`/`, and `\\` with the `windows` option) when they are the only characters in a path segment. Thus, `foo**/bar` is equivalent to `foo*/bar`, and `foo/a**b/bar` is equivalent to `foo/a*b/bar`, and *more than two* consecutive stars in a glob path segment are regarded as *a single star*. Thus, `foo/***/bar` is equivalent to `foo/*/bar`.
`?`	Matches any character excluding path separators one time. Does *not match* path separators or leading dots.
`[abc]`	Matches any characters inside the brackets. For example, `[abc]` would

```
isMatch('bar');
isMatch('baz');
```

options.onResult

```
const onResult = ({ glob, regex, input,
        output }) => {
  console.log({ glob, regex, input,
        output });
};

const isMatch = picomatch('*',
        { onResult, ignore: 'f*' });
isMatch('foo');
isMatch('bar');
isMatch('baz');
```

Globbing features

- Basic globbing (Wildcard matching)
- Advanced globbing (extglobs, posix brackets, brace matching)

Option	Type	Default value	Description
`maxLength`	`number`	`65536`	Limit the max the input string is thrown if the string is longer value.
`nobrace`	`boolean`	`false`	Disable brace so that `{a,b}` `{1..3}` would treated as litera characters.
`nobracket`	`boolean`	`undefined`	Disable matchi regex brackets.
`nocase`	`boolean`	`false`	Make matching insensitive. Equ the regex `i` flag that this option overridden by t `flags` option.
`nodupes`	`boolean`	`true`	Deprecated, us `nounique` ins option will be r in a future majo By default dup removed. Disab uniquification l this option to fa
`noext`	`boolean`	`false`	Alias for `noex`
`noextglob`	`boolean`	`false`	Disable suppor matching with (like `+(a|b)`
`noglobstar`	`boolean`	`false`	Disable suppor matching neste directories with (`**`)
`nonegate`	`boolean`	`false`	

Option	Type	Default value	Description
			Disable support for negating with leading
noquantifiers	boolean	false	Disable support for r quantifiers (like a{1,2}) and treat th as brace patterns to b expanded.
onIgnore	function	undefined	Function to be called ignored items.
onMatch	function	undefined	Function to be called matched items.
onResult	function	undefined	Function to be called all items, regardless whether or not they a matched or ignored.
posix	boolean	false	Support POSIX character classes (“po brackets”).
posixSlashes	boolean	undefined	Convert all slashes in paths to forward slas This does not conver slashes in the glob pattern itself
prepend	boolean	undefined	String to prepend to t generated regex used matching.
regex	boolean	false	Use regular expressio rules for + (instead o matching literal +), a for stars that follow closing parentheses c brackets (as in) * an]*).
strictBrackets	boolean	undefined	


```
// strip leading './' from strings
const format = str => str.replace(/^\.
        \//, '');
const isMatch = picomatch('foo/*.js',
        { format });
console.log(isMatch('./foo/bar.js')); //
        => true
```

options.onMatch

```
const onMatch = ({ glob, regex, input,
        output }) => {
  console.log({ glob, regex, input,
        output });
};

const isMatch = picomatch('*',
        { onMatch });
isMatch('foo');
isMatch('bar');
isMatch('baz');
```

options.onIgnore

```
const onIgnore = ({ glob, regex, input,
        output }) => {
  console.log({ glob, regex, input,
        output });
};

const isMatch = picomatch('*',
        { onIgnore, ignore: 'f*' });
isMatch('foo');
```

```
  expandRange(a, b) {
    return `(${fill(a, b, { toRegex:
        true })})`;
  }
});

console.log(regex);
//=> /^(?:foo\/((?:0[1-9]|1[0-9]|
        2[0-5]))\/bar)$/

console.log(regex.test('foo/00/
        bar'))  // false
console.log(regex.test('foo/01/
        bar'))  // true
console.log(regex.test('foo/10/
        bar')) // true
console.log(regex.test('foo/22/
        bar')) // true
console.log(regex.test('foo/25/
        bar')) // true
console.log(regex.test('foo/26/
        bar')) // false
```

options.format

Type: `function`

Default: `undefined`

Custom function for formatting strings before they're matched.

Example

Option	Type	Default value	Description
			Throw an error
			brackets, brace
			parens are imba
`strictSlashes`	`boolean`	`undefined`	When true, pic won't match tra slashes with sir
`unescape`	`boolean`	`undefined`	Remove backsl preceding escap characters in th pattern. By def backslashes are
`unixify`	`boolean`	`undefined`	Alias for `posixSlash` backwards com
`windows`	`boolean`	`false`	Also accept bac as the path sepa

Scan Options

In addition to the main picomatch options, the following options may also be used with the .scan method.

Option	Type	Default value	Description
`tokens`	`boolean`	`false`	When `true`, the returned object will include an array of tokens (objects), representing each path "segment" in the scanned glob pattern
`parts`	`boolean`	`false`	

Option	Type	Default value	Description
			When `true`, the returned object will include an array of strings representing each path "segment" in the scanned glob pattern. This is automatically enabled when `options.tokens` is true

Example

```
const picomatch = require('picomatch');
const result = picomatch.scan('!./foo/
        *.js', { tokens: true });
console.log(result);
// {
//   prefix: '!./',
//   input: '!./foo/*.js',
//   start: 3,
//   base: 'foo',
//   glob: '*.js',
//   isBrace: false,
//   isBracket: false,
//   isGlob: true,
//   isExtglob: false,
//   isGlobstar: false,
//   negated: true,
//   maxDepth: 2,
//   tokens: [
```



```
//     { value: '!./', depth: 0, isGlob:
         false, negated: true, isPrefix:
         true },
//     { value: 'foo', depth: 1, isGlob:
         false },
//     { value: '*.js', depth: 1,
         isGlob: true }
//   ],
//   slashes: [ 2, 6 ],
//   parts: [ 'foo', '*.js' ]
// }
```

Options Examples

options.expandRange

Type: `function`

Default: `undefined`

Custom function for expanding ranges in brace patterns. The fill-range library is ideal for this purpose, or you can use custom code to do whatever you need.

Example

The following example shows how to create a glob that matches a folder

```
const fill = require('fill-range');
const regex = pm.makeRe('foo/{01..25}/
        bar', {
```