

100% OFFLINE

· WEB-WORKER SANDBOX

· ZERO DEPENDENCIES

Regex Tester & Evaluator

User Guide & Field Handbook

Write a pattern and sample text, watch matches highlight live, break out capture groups and offsets, verify a whole list of inputs — and catch catastrophic patterns before they bite.

```
/^(?!30) (?!([0-9])\1{9}) \d{10}$/
```

What the program does

A single-file, fully offline instrument for reading and validating regular expressions.

1

Live matching

Type a pattern and sample text; every match highlights the instant you stop typing.

2

Groups & offsets

Each capture group is broken out with its value and exact start–end character offsets.

3

Verify a whole set

Test-case mode checks the pattern against a list of inputs with a running pass/fail tally.

4

Safe by design

A 250 ms watchdog stops catastrophic patterns before they can freeze the browser tab.

Engine: **JavaScript (ECMAScript)** — everything runs in your browser, nothing is ever uploaded.

What you need to run it

One file. No install, no dependencies, no build step, no internet.

A**index.html**

The entire application — HTML, CSS and JavaScript in one file. Double-click to open it.

B**A modern browser**

Chrome, Edge, Firefox or Safari. All evaluation happens client-side in a Web Worker.

C**A pattern to test**

Optional. Bring a regex from your rules file, or start from the built-in cheat sheet.

Private by default. Your patterns and test text are read locally and never sent over the network — safe for confidential validation rules and sample data.

The screen at a glance

Eight areas, top to bottom. Every panel reads the same live evaluation.

1

Pattern field

The raw regular expression, framed by `/.../` and its active flags.

2

Flag toggles

`g i m s u v y` — each shown with its plain-language name.

3

Status bar

Match count, execution time in milliseconds, and current state.

4

Mode switch

Flip between Text & highlight and Test cases.

5

Test text & results

Sample text with live highlights; matches and groups listed alongside.

6

Replace preview

The result of a replacement, updated as you type it.

7

Pattern library

Save, export and import named patterns as JSON.

8

Cheat sheet

A click-to-insert reference plus engine-flavour notes.

Writing a pattern

The field is raw regex — the single most important habit is one backslash, not two.

1 It's raw regex

Type the expression only — no quotes, no slashes. The `/.../` around the box is just a frame.

2 Single backslashes

Write `\d`, `\1`, `\b` — never `\\d`. Doubling is only for JSON and code strings.

3 Errors surface live

An invalid pattern shows a plain reason; results clear until you fix it.

4 Engine label

A visible tag reads Engine: JavaScript (ECMAScript) — the flavour you are testing.

In the field, type:

`^\d{10}$` correct — matches ten digits

`^\\d{10}$` wrong — needs a literal `"\"`

Why it matters: a doubled backslash is still **valid** regex, so no error appears — it just quietly means “a backslash”, and your rule stops matching.

Flags — what each letter does

Toggles that change how the whole pattern is applied. Active ones show after the `/.../`.

g**global** Find every match, not just the first one.**m****multiline** `^` and `$` match at the start/end of each line.**u****unicode** Enables `\p{...}` escapes and correct emoji handling.**y****sticky** Match must start exactly at the current position.**i****ignore case** Letters match regardless of upper- or lowercase.**s****dotAll** The dot `.` also matches a newline character.**v****unicodeSets** Richer character classes — mutually exclusive with `u`.

Tip: for whole-string validation with `^ ... $` you usually need no flags at all — and `i` or `m` can quietly make a check looser than intended.

Reading matches & capture groups

The results panel is where a pattern proves what it actually captured.

1

Per-match detail

Index number, start–end offsets, and the matched value for every hit.

2

Nested groups

Each numbered or named group shows its value and its own offsets.

3

Participation

“did not participate” (undefined) is shown apart from “matched empty string”.

4

Click to locate

Click a match to scroll the text to it and pulse that highlight.

Example result for `/(?<y>\d{4})-(?<m>\d{2})/` on “2024-01”

#1	[0–7]	2024-01	\$1	<y>	2024	[0–4]	\$2	<m>	01	[5–7]
----	-------	---------	-----	-----	------	-------	-----	-----	----	-------

Highlighting in the text view

Colour carries meaning — the view is an instrument for reading text closely.

1 Two shades

Adjacent matches alternate between two shades so they stay distinct instead of merging into one block.

2 Group underlines

Spans captured by a group inside a match are underlined in a second colour, nested within the match.

3 Offsets everywhere

Character offsets are the recurring motif — in the results, the status bar, and each match value.

4 Aligned overlay

A transparent editor sits over a highlight mirror, kept pixel-aligned so text and colour never drift.

One colour is reserved for the error / timeout state and appears nowhere else — so a problem always reads as a problem.

Replace preview

Type a replacement and see the result live — the full token set is supported.

`$1-$99`

numbered group

`$<name>`

named group

`$&`

the whole match

`$``

text before match

`$'`

text after match

`$$`

a literal `$`

Worked example

pattern `(\w+)\@(\w+)` replacement `$2 at $1 [$&]`

input `me@host` result **`host at me [me@host]`**

Test-case mode

Often the real question is “does this catch all my known inputs?” — this answers it.

1

One row per input

Each string is tested independently.

2

PASS / FAIL badges

Colour-coded per row, at a glance.

3

Running tally

A live “2 / 8 match” count.

4

Add, remove, reorder

Build the list as you go.

Example — `/^(?!30)(?!([0-9])\1{9})\d{10}$/`

PASS	1234567890	ten digits, not 30..., not all-same
PASS	3123456789	starts 31 — only 30 is blocked
FAIL	3012345678	starts with 30
FAIL	1111111111	all identical digits

The catastrophic-backtracking watchdog

Live evaluation plus an unbounded engine is a hung tab waiting to happen. This prevents it.

1

Runs in a worker

All matching happens off the main thread, so the interface can stay responsive.

2

250 ms limit

A watchdog times every run. Cross the limit and the run is abandoned.

3

Terminate & respawn

The stuck worker is killed and instantly replaced — no reload needed.

4

Warns clearly

You get a plain message naming the likely cause: ReDoS backtracking.

Treat it as a feature. A pattern that trips the watchdog — like `(a+)+$` against a long string — is one that would hang a production system on hostile input. Surfacing that is genuinely useful.

Pattern library

Keep the patterns you rely on — and move them between machines as a file.

1

Save named patterns

Store the pattern, its flags and the test text together under a name you choose.

2

Export to JSON

Write your whole library to a file — the reliable, portable way to keep it.

3

Import to reload

Read a library file back in and merge it with what you already have saved.

4

Convenience storage

Saves also sit in localStorage, but treat that as a convenience, not a guarantee.

Why export/import is primary: localStorage behaves inconsistently under the file:// protocol across browsers — a JSON file is the dependable record.

Cheat sheet & flavour differences

A built-in reference — and an honest list of what ECMAScript regex cannot do.

1

Click to insert

Every token drops straight into the pattern field at your cursor.

2

Grouped reference

Anchors, character classes, quantifiers, groups, look-around and escapes.

3

Flavour gaps

No atomic groups, possessive quantifiers, recursion, or `\A \z \Z` anchors.

4

Not PCRE

A pattern that passes here can still fail in Python, Java, .NET or Go.

For the compliance case: if a rule is destined for another engine, the flavour notes tell you exactly which constructs won't carry over — test in the target engine too.

Testing regex that lives in JSON rules

The backslash count depends on where the pattern sits. Get this right and mismatches vanish.

In the JSON file (on disk)

```
"\\d{10}"
```

double

JSON treats `\` as an escape, so a real backslash is written `\\`.

After the program parses it

```
"\d{10}"
```

single

`JSON.parse()` turns every `\\` back into a single `\`.

In the tester's pattern field

```
\d{10}
```

single

The field is raw regex — type exactly what `parse()` would produce.

Rule of thumb: copying a rule from JSON into the field, drop one backslash from each pair. Don't change your JSON — it's correct as it is.

Troubleshooting & tips

The most common snags — and the one-line fix for each.

Everything fails

Doubled backslashes. Use single: `\d`, not `\\d`.

Valid but wrong

`\\1` is a literal backslash, not the backreference `\1`.

No matches

Check flags and anchors; an empty pattern shows nothing.

Timed out

Catastrophic backtracking — simplify the pattern.

Too loose

`i` or `m` can widen a check; clear them for validation.

Nothing saved

Export to JSON; `localStorage` isn't reliable on `file://`.

QUICK REFERENCE

You're ready to test

Open • Double-click index.html in any modern browser.

Pattern • Type raw regex — single backslashes (\d, not \\d).

Flags • Toggle g i m s u v y; clear them for whole-string validation.

Test • Use Test cases to check many inputs at once — PASS / FAIL tally.

Guardrail • A 250 ms watchdog stops catastrophic patterns cold.

Save • Export patterns to JSON; import to reload them anywhere.