
run-tests
no status
npm@latest
v10.1.0
license
MIT

URLPattern polyfills

URLPattern is a new web API for matching URLs. Its intended to both provide a convenient API for web developers and to be usable in other web APIs that need to match URLs; e.g. service workers. The explainer discusses the motivating use cases.

This is a polyfill for the URLPattern API so that the feature is available in browsers that don't support it natively. This polyfill passes the same web platform test suite.

How to load the polyfill

The polyfill works in browsers (ESM module) and in Node.js either via import (ESM module) or via require (CJS module).

The polyfill will only be loaded if the URLPattern doesn't already exist on the global object, and in that case it will add it to the global object.

Reporting a security issue

If you have information about a security issue or vulnerability with an Intel-maintained open source project on https://github.com/intel, please send an e-mail to secure@intel.com. Encrypt sensitive information using our PGP public key. For issues related to Intel products, please visit https://security-center.intel.com.

Learn more

- Explainer
- Design Document

loading as ESM module

```
// Conditional ESM module loading
        (Node.js and browser)
// @ts-ignore: Property 'UrlPattern'
        does not exist
if (!globalThis.URLPattern) {
  await import("urlpattern-polyfill");
}
/**
 * The above is the recommended way to
        load the ESM module, as it only
 * loads it on demand, thus when not
        natively supported by the
        runtime or
 * already polyfilled.
 */
import "urlpattern-polyfill";

/**
 * In case you want to replace an
        existing implementation with the
        polyfill:
 */
import {URLPattern} from "urlpattern-
        polyfill";
globalThis.URLPattern = URLPattern
```

Note:

The line with `// @ts-ignore: Property 'UrlPattern' does not exist` *is needed in some environments because before you load the polyfill it might not be available, and the feature-check in the if statement gives an TypeScript error. The whole idea is that it loads when its not there.*

loading as CommonJs module

```
// Conditional CJS module loading
        (Node.js)
if (!globalThis.URLPattern) {
   require("urlpattern-polyfill");
}
/**
 * The above is the recommended way to
        load the CommonJs module, as it
        only
 * loads it on demand, thus when not
        natively supported by the
        runtime or
 * already polyfilled.
 */
require("urlpattern-polyfill");

/**
```


Breaking changes

- V9.0.0 drops support for NodeJS 14 and lower. NodeJS 15 or higher is required. This is due to using private class fields, so we can have better optimalizations. There is *No* change in functionality, but we were able to reduce the size of the polyfill by ~2.5KB (~13%), thanks to a pr #118 from @jimmywarting.

Canonicalization

URLs have a canonical form that is based on ASCII, meaning that internationalized domain names (hostnames) also have a canonical ASCII based representation, and that other components such as `hash`, `search` and `pathname` are encoded using percent encoding.

Currently `URLPattern` does not perform any encoding or normalization of the patterns. So a developer would need to URL encode unicode characters before passing the pattern into the constructor. Similarly, the constructor does not do things like flattening pathnames such as /foo/../bar to /bar. Currently the pattern must be written to target canonical URL output manually.

It does, however, perform these operations for `test()` and `exec()` input.

Encoding components can easily be done manually, but do not encoding the pattern syntax:

```
encodeURIComponent("?q=æøå")
// "%3Fq%3D%C3%A6%C3%B8%C3%A5"

new URL("https://
        ølerlækkernårdetermit.dk").hostname
// "xn--lerlkkernrdetermit-dubo78a.dk"
```



```
 * In case you want to replace an
        existing implementation with the
        polyfill:
 */
const {URLPattern} =
        require("urlpattern-polyfill");;
globalThis.URLPattern = URLPattern
```

Note:

No matter how you load the polyfill, when there is no implementation in your environment, it will always add it to the global object.

Basic example

```
let p = new URLPattern({ pathname: '/
        foo/:name' });

let r = p.exec('https://example.com/foo/
        bar');
console.log(r.pathname.input); // "/foo/
        bar"
console.log(r.pathname.groups.name); //
        "bar"
```

```
let r2 = p.exec({ pathname: '/foo/
        baz' });
console.log(r2.pathname.groups.name); //
        "baz"
```

Example of matching same-origin JPG or PNG requests

```
// Match same-origin jpg or png URLs.
// Note: This uses a named group to make
        it easier to access
//       the result later.
const p = new URLPattern({
  pathname: '/*.:filetype(jpg|png)',
  baseURL: self.location
});

for (let url in url_list) {
  const r = p.exec(url);

  // skip non-matches
  if (!r) {
    continue;
  }

  if (r.pathname.groups['filetype'] ===
        'jpg') {
    // process jpg
  } else if
        (r.pathname.groups['filetype']
        === 'png') {
```


Currently we plan to have these known differences with path-to-regexp:

- No support for custom prefixes and suffixes.

Pattern syntax

The pattern syntax here is based on what is used in the popular path-to-regexp library.

- An understanding of a “divider” that separates segments of the string. For the pathname this is typically the `"/"` character.
- A regex group defined by an enclosed set of parentheses. Inside of the parentheses a general regex may be defined.
- A named group that matches characters until the next divider. The named group begins with a `":"` character and then a name. For example, `"/:foo/:bar"` has two named groups.
- A custom regex for a named group. In this case a set of parentheses with a regex immediately follows the named group; e.g. `"/:foo(.*)"` will override the default of matching to the next divider.
- A modifier may optionally follow a regex or named group. A modifier is a `"?"`, `"*"`, or `"+"` functions just as they do in regular expressions. When a group is optional or repeated and it’s preceded by a divider then the divider is also optional or repeated. For example, `"/foo/:bar?"` will match `"/foo"`, `"/foo/"`, or `"/foo/baz"`. Escaping the divider will make it required instead.
- A way to greedily match characters, even across dividers, by using `"(.*)"` (so-called unnamed groups).


```
    // process png
  }
}
```

The pattern in this case can be made simpler without the origin check by leaving off the baseURL.

```
// Match any URL ending with 'jpg' or
        'png'.
const p = new URLPattern({ pathname: '/
        *.:filetype(jpg|png)' });
```

Example of Short Form Support

We are planning to also support a “short form” for initializing URLPattern objects. This is supported by the polyfill but not yet by the Chromium implementation.

For example:

```
const p = new URLPattern("https://
        *.example.com/foo/*");
```

Or:

```
const p = new URLPattern("foo/*",
        self.location);
```

API reference

API overview with typeScript type annotations is found below.

Associated browser Web IDL can be found here.

```
type URLPatternInput = URLPatternInit |
        string;

class URLPattern {
  constructor(init?: URLPatternInput,
        baseURL?: string);

  test(input?: URLPatternInput,
        baseURL?: string): boolean;

  exec(input?: URLPatternInput,
        baseURL?: string):
        URLPatternResult | null;

  readonly protocol: string;
  readonly username: string;
  readonly password: string;
  readonly hostname: string;
  readonly port: string;
  readonly pathname: string;
  readonly search: string;
  readonly hash: string;
}

interface URLPatternInit {
  baseURL?: string;
  username?: string;
  password?: string;
  protocol?: string;
  hostname?: string;
  port?: string;
  pathname?: string;
  search?: string;
  hash?: string;
}

interface URLPatternResult {
  inputs: [URLPatternInput];
  protocol: URLPatternComponentResult;
  username: URLPatternComponentResult;
  password: URLPatternComponentResult;
  hostname: URLPatternComponentResult;
  port: URLPatternComponentResult;
  pathname: URLPatternComponentResult;
  search: URLPatternComponentResult;
  hash: URLPatternComponentResult;
}

interface URLPatternComponentResult {
  input: string;
  groups: {
      [key: string]: string | undefined;
  };
}
```