useQueryState hook for Next.js - Like React.useState, but stored in the URL query string.

next-usequerystate.vercel.app/

MIT license

243 stars **23** forks Activity

franky47 chore: Add links to sources ··· ✓ 9 hours ago 389

View code

README.md

useQueryState for Next.js

npm v1.8.0 license MIT Continuous Integration passing dependencies 0 of 1 outdated

useQueryState hook for Next.js - Like React.useState, but stored in the URL query string

Features

- 🔀 ***new:*** Supports both the `app` and `pages` routers
- 🧘‍♀️ Simple: the URL is the source of truth
- 🕰 Replace history or append to use the Back button to navigate state updates
- ⚡️ Built-in parsers for common state types (integer, float, boolean, Date, and more)
- ♊️ Related querystrings with `useQueryStates`
- 📡 Shallow mode by default for URL query updates, opt-in to notify server components

Installation

```
$ pnpm add next-usequerystate
$ yarn add next-usequerystate
$ npm install next-usequerystate
```

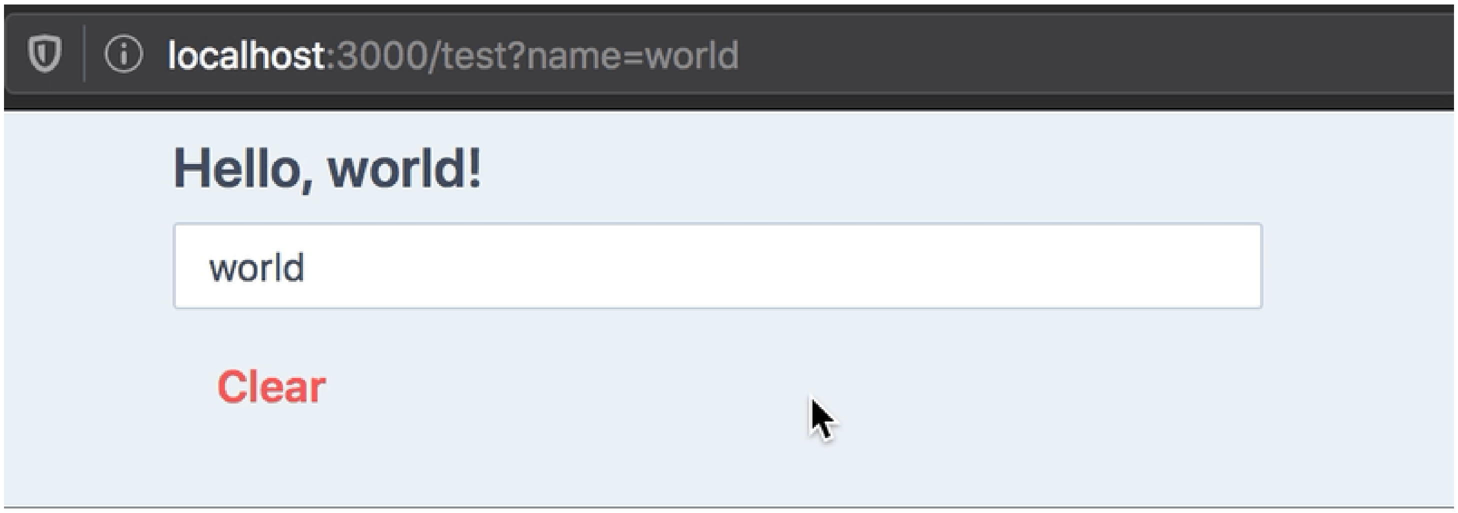
> Note: version 1.8.0 requires Next.js 13.4+.

Usage

```
'use client' // app router: only works in client components

import { useQueryState } from 'next-usequerystate'

export default () => {
  const [name, setName] = useQueryState('name')
  return (
    <>
      <h1>Hello, {name || 'anonymous visitor'}!</h1>
      <input value={name || ''} onChange={e => setName(e.target.value)} />
      <button onClick={() => setName(null)}>Clear</button>
    </>
  )
}
```



Documentation

`useQueryState` takes one required argument: the key to use in the query string.

Like `React.useState`, it returns an array with the value present in the query string as a string (or `null` if none was found), and a state updater function.

Example outputs for our hello world example:

URL	name value	Notes
`/`	`null`	No `name` key in URL
`/?name=`	`''`	Empty string
`/?name=foo`	`'foo'`	
`/?name=2`	`'2'`	Always returns a string by default, see Parsing below

Parsing

If your state type is not a string, you must pass a parsing function in the second argument object.

We provide parsers for common and more advanced object types:

```
import {
  parseAsString,
  parseAsInteger,
  parseAsFloat,
  parseAsBoolean,
  parseAsTimestamp,
  parseAsIsoDateTime,
  parseAsArrayOf,
  parseAsJson,
  parseAsStringEnum
} from 'next-usequerystate'

useQueryState('tag') // defaults to string
useQueryState('count', parseAsInteger)
useQueryState('brightness', parseAsFloat)
useQueryState('darkMode', parseAsBoolean)
useQueryState('after', parseAsTimestamp) // state is a Date
useQueryState('date', parseAsIsoDateTime) // state is a Date
useQueryState('array', parseAsArrayOf(parseAsInteger)) // state is number[]
useQueryState('json', parseAsJson<Point>()) // state is a Point

// Enums (string-based only)
enum Direction {
  up = 'UP',
  down = 'DOWN',
  left = 'LEFT',
  right = 'RIGHT'
}

const [direction, setDirection] = useQueryState(
  'direction',
  parseAsStringEnum<Direction>(Object.values(Direction)) // pass a list of allowed values
    .withDefault(Direction.up)
)
```

You may pass a custom set of `parse` and `serialize` functions:

```
import { useQueryState } from 'next-usequerystate'

export default () => {
  const [hex, setHex] = useQueryState('hex', {
    // TypeScript will automatically infer it's a number
    // based on what `parse` returns.
    parse: (query: string) => parseInt(query, 16),
    serialize: value => value.toString(16)
  })
}
```

Using parsers in Server Components

If you wish to parse the searchParams in server components, you'll need to import the parsers from `next-usequerystate/parsers`, which doesn't include the `"use client"` directive.

You can then use the `parseServerSide` method:

```
import { parseAsInteger } from 'next-usequerystate/parsers'

type PageProps = {
  searchParams: {
    counter?: string | string[]
  }
}

const counterParser = parseAsInteger.withDefault(1)

export default function ServerPage({ searchParams }: PageProps) {
  const counter = counterParser.parseServerSide(searchParams.counter)
  console.log('Server side counter: %d', counter)
  return (
    ...
  )
}
```

See the server-side parsing demo for a live example showing how to reuse parser configurations between client and server code.

> Note: parsers **don't validate** your data. If you expect positive integers or JSON-encoded objects of a particular shape, you'll need to feed the result of the parser to a schema validation library, like Zod.

Default value

When the query string is not present in the URL, the default behaviour is to return `null` as state.

It can make state updating and UI rendering tedious. Take this example of a simple counter stored in the URL:

```
import { useQueryState, parseAsInteger } from 'next-usequerystate'

export default () => {
  const [count, setCount] = useQueryState('count', parseAsInteger)
  return (
    <>
      <pre>count: {count}</pre>
      <button onClick={() => setCount(0)}>Reset</button>
      {/* handling null values in setCount is annoying: */}
      <button onClick={() => setCount(c => c ?? 0 + 1)}>+</button>
      <button onClick={() => setCount(c => c ?? 0 - 1)}>-</button>
      <button onClick={() => setCount(null)}>Clear</button>
    </>
    )
  }
```

You can specify a default value to be returned in this case:

```
const [count, setCount] = useQueryState('count', parseAsInteger.withDefault(0))

const increment = () => setCount(c => c + 1) // c will never be null
const decrement = () => setCount(c => c - 1) // c will never be null
const clearCount = () => setCount(null) // Remove query from the URL
```

Note: the default value is internal to React, it will **not** be written to the URL.

Setting the state to `null` will remove the key in the query string and set the state to the default value.

Options

History

By default, state updates are done by replacing the current history entry with the updated query when state changes.

You can see this as a sort of `git squash`, where all state-changing operations are merged into a single history value.

You can also opt-in to push a new history item for each state change, per key, which will let you use the Back button to navigate state updates:

```
// Default: replace current history with new state
useQueryState('foo', { history: 'replace' })

// Append state changes to history:
useQueryState('foo', { history: 'push' })
```

Any other value for the `history` option will fallback to the default.

You can also override the history mode when calling the state updater function:

```
const [query, setQuery] = useQueryState('q', { history: 'push' })

// This overrides the hook declaration setting:
setQuery(null, { history: 'replace' })
```

Shallow

By default, query state updates are done in a *client-first* manner: there are no network calls to the server.

This is equivalent to the `shallow` option of the Next.js router set to `true`.

> Note: the app router doesn't yet have this capabily natively, but `next-usequerystate` does, by bypassing the router on shallow updates.

To opt-in to query updates notifying the server (to re-run `getServerSideProps` in the pages router and re-render Server Components on the app router), you can set `shallow` to `false` :

```
const [state, setState] = useQueryState('foo', { shallow: false })

// You can also pass the option on calls to setState:
setState('bar', { shallow: false })
```

Scroll

The Next.js router scrolls to the top of the page on navigation updates, which may not be desirable when updating the query string with local state.

Query state updates won't scroll to the top of the page by default, but you can opt-in to this behaviour (which was the default up to 1.8.0):

```
const [state, setState] = useQueryState('foo', { scroll: true })

// You can also pass the option on calls to setState:
setState('bar', { scroll: true })
```

Configuring parsers, default value & options

You can use a builder pattern to facilitate specifying all of those things:

```
useQueryState(
  'counter',
  parseAsInteger
    .withOptions({
      history: 'push',
      shallow: false
    })
    .withDefault(0)
)
```

Note: `withDefault` must always come **after** `withOptions` to ensure proper type safety (providing a non-nullable state type).

You can get this pattern for your custom parsers too, and compose them with others:

```
import { createParser, parseAsHex } from 'next-usequerystate/parsers'

// Wrapping your parser/serializer in `createParser`
// gives it access to the builder pattern & server-side
// parsing capabilities:
const hexColorSchema = createParser({
  parse(query) {
    if (query.length !== 6) {
      return null // always return null for invalid inputs
    }
    return {
      // When composing other parsers, they may return null too.
      r: parseAsHex.parse(query.slice(0, 2)) ?? 0x00,
      g: parseAsHex.parse(query.slice(2, 4)) ?? 0x00,
      b: parseAsHex.parse(query.slice(4)) ?? 0x00
    }
  },
  serialize({ r, g, b }) {
    return (
      parseAsHex.serialize(r) +
      parseAsHex.serialize(g) +
      parseAsHex.serialize(b)
    )
  }
})
  // Eg: set common options directly
  .withOptions({ history: 'push' })

// Or on usage:
useQueryState(
  'tribute',
  hexColorSchema.withDefault({
    r: 0x66,
    g: 0x33,
    b: 0x99
  })
)
```

Note: see this example running in the hex-colors demo.

Multiple Queries (batching)

You can call as many state update function as needed in a single event loop tick, and they will be applied to the URL asynchronously:

```
  const MultipleQueriesDemo = () => {
    const [lat, setLat] = useQueryState('lat', parseAsFloat)
    const [lng, setLng] = useQueryState('lng', parseAsFloat)
    const randomCoordinates = React.useCallback(() => {
      setLat(Math.random() * 180 - 90)
      setLng(Math.random() * 360 - 180)
    }, [])
  }
```

If you wish to know when the URL has been updated, and what it contains, you can await the Promise returned by the state updater function, which gives you the updated URLSearchParameters object:

```
const randomCoordinates = React.useCallback(() => {
  setLat(42)
  return setLng(12)
}, [])

randomCoordinates().then((search: URLSearchParams) => {
  search.get('lat') // 42
  search.get('lng') // 12, has been queued and batch-updated
})
```

▶ *Implementation details (Promise caching)*

`useQueryStates`

For query keys that should always move together, you can use `useQueryStates` with an object containing each key's type:

```
import { useQueryStates, parseAsFloat } from 'next-usequerystate'

const [coordinates, setCoordinates] = useQueryStates(
  {
    lat: parseAsFloat.withDefault(45.18),
    lng: parseAsFloat.withDefault(5.72)
  },
  {
    history: 'push'
  }
)

const { lat, lng } = coordinates

// Set all (or a subset of) the keys in one go:
const search = await setCoordinates({
  lat: Math.random() * 180 - 90,
  lng: Math.random() * 360 - 180
})
```

Testing

Currently, the best way to test the behaviour of your components using `useQueryState(s)` is end-to-end testing, with tools like Playwright or Cypress.

Running components that use the Next.js router in isolation requires mocking it, which is being worked on for the app router.

If you wish to spy on changes in the URL query string, you can use `subscribeToQueryUpdates` :

```
import { subscribeToQueryUpdates } from 'next-usequerystate'

React.useEffect(
  () =>
    subscribeToQueryUpdates(({ search, source }) => {
      console.log(search.toString()) // URLSearchParams
      console.log(source) // 'internal' | 'external'
    }),
  // This returns an unsubscribe function
  []
)
```

See issue #259 for more testing-related discussions.

Caveats

Because the Next.js **pages router** is not available in an SSR context, this hook will always return `null` (or the default value if supplied) on SSR/SSG.

This limitation doesn't apply to the app router.

Lossy serialization

If your serializer loses precision or doesn't accurately represent the underlying state value, you will lose this precision when reloading the page or restoring state from the URL (eg: on navigation).

Example:

```
const geoCoordParser = {
  parse: parseFloat,
  serialize: v => v.toFixed(4) // Loses precision
}

const [lat, setLat] = useQueryState('lat', geoCoordParser)
```

Here, setting a latitude of 1.23456789 will render a URL query string of `lat=1.2345` , while the internal `lat` state will be correctly set to 1.23456789.

Upon reloading the page, the state will be incorrectly set to 1.2345.

License

MIT

- Made with ❤️ by François Best

Using this package at work ? Sponsor me to help with support and maintenance.

Releases 43

v1.8.0 Latest
16 hours ago

+ 42 releases

Sponsor this project

franky47 François Best

liberapay.com/**francoisbest**

https://paypal.me/francoisbest?locale.x=fr_FR

Learn more about GitHub Sponsors

Packages

No packages published

Contributors 9

Languages

● **TypeScript** 92.6% ● **JavaScript** 7.3% ● **Shell** 0.1%