morphdom

Download count all time

Lightweight module for morphing an existing DOM node tree to match a target DOM node tree. It's fast and works with the real DOM—no virtual DOM needed!

This module was created to solve the problem of updating the DOM in response to a UI component or page being rerendered. One way to update the DOM is to simply toss away the existing DOM tree and replace it with a new DOM tree (e.g., `myContainer.innerHTML = newHTML`). While replacing an existing DOM tree with an entirely new DOM tree will actually be very fast, it comes with a cost. The cost is that all of the internal state associated with the existing DOM nodes (scroll positions, input caret positions, CSS transition states, etc.) will be lost. Instead of replacing the existing DOM tree with a new DOM tree we want to *transform* the existing DOM tree to match the new DOM tree while minimizing the number of changes to the existing DOM tree. This is exactly what the `morphdom` module does! Give it an existing DOM node tree and a target DOM node tree and it will efficiently transform the existing DOM node tree to

exactly match the target DOM node tree with the minimum amount of changes.

`morphdom` does not rely on any virtual DOM abstractions. Because `morphdom` is using the *real* DOM, the DOM that the web browser is maintaining will always be the source of truth. Even if you have code that manually manipulates the DOM things will still work as expected. In addition, `morphdom` can be used with any templating language that produces an HTML string.

The transformation is done in a single pass of both the original DOM tree and the target DOM tree and is designed to minimize changes to the DOM while still ensuring that the morphed DOM exactly matches the target DOM. In addition, the algorithm used by this module will automatically match up elements that have corresponding IDs and that are found in both the original and target DOM tree.

Support for diffing the real DOM with a virtual DOM was introduced in `v2.1.0`. Virtual DOM nodes are expected to implement the minimal subset of the real DOM API required by `morphdom` and virtual DOM nodes are automatically upgraded real DOM nodes if they need to be moved into the real DOM. For more details, please see: docs/virtual-dom.md.

License

MIT

Usage

First install the module into your project:

```
npm install morphdom --save
```

NOTE: Published npm packages: `- dist/morphdom-umd.js - dist/morphdom-esm.js`

The code below shows how to morph one `element` to `another` element.

```
var morphdom = require('morphdom');

var el1 = document.createElement('div');
el1.className = 'foo';

var el2 = document.createElement('div');
el2.className = 'bar';

morphdom(el1, el2);

expect(el1.className).to.equal('bar');
```

You can also pass in an HTML string for the second argument:

```
var morphdom = require('morphdom');

var el1 = document.createElement('div');
el1.className = 'foo';
el1.innerHTML = 'Hello John';
```

```
morphdom(el1, 'Hello Frank');

expect(el1.className).to.equal('bar');
expect(el1.innerHTML).to.equal('Hello
        Frank');
```

NOTE: This module will modify both the original and target DOM node tree during the transformation. It is assumed that the target DOM node tree will be discarded after the original DOM node tree is morphed.

Contribute

Pull Requests welcome. Please submit Github issues for any feature enhancements, bugs or documentation problems. Please make sure tests pass:

```
npm test
```

Maintainers

- Patrick Steele-Idem (Twitter: @psteeleidem)
- Scott Newcomer (Twitter: @puekey)

Examples

See: ./examples/

Browser Support

- IE9+ and any modern browser
- Proper namespace support added in `v1.4.0`

morphdom nanomorph virtual-dom attr-value-empty-string **0.01ms** 0.02ms **0.01ms** change-tagname 0.01ms **0.00ms** 0.01ms change-tagname-ids 0.02ms **0.00ms** 0.02ms data-table **0.60ms** 1.38ms 0.80ms data-table2 2.72ms 12.42ms **0.84ms** equal 0.50ms 1.33ms **0.02ms** id-change-tag-name **0.03ms** 0.04ms 0.04ms ids-nested 0.08ms 0.02ms **0.01ms** ids-nested-2 0.03ms **0.02ms** 0.05ms ids-nested-3 0.03ms **0.01ms** 0.02ms ids-nested-4 0.04ms 0.04ms **0.03ms** ids-nested-5 **0.04ms** 0.08ms 0.06ms ids-nested-6 **0.03ms** 0.03ms 0.03ms ids-nested-7 **0.02ms** 0.03ms 0.02ms ids-prepend 0.03ms 0.03ms **0.02ms** input-element 0.02ms 0.04ms **0.00ms** input-element-disabled **0.01ms** 0.02ms **0.01ms** input-element-enabled **0.01ms** 0.02ms **0.01ms** large 2.88ms 12.11ms **0.35ms** lengthen **0.03ms** 0.15ms 0.04ms one **0.01ms** 0.03ms 0.01ms reverse 0.03ms 0.05ms **0.02ms** reverse-ids 0.05ms 0.07ms **0.02ms** select-element 0.07ms 0.14ms **0.03ms** shorten 0.03ms 0.07ms **0.02ms** simple **0.02ms** 0.05ms **0.02ms** simple-ids 0.05ms 0.09ms **0.04ms** simple-text-el **0.03ms** 0.04ms **0.03ms** svg 0.03ms 0.05ms **0.01ms** svg-append **0.06ms** 0.06ms 0.07ms svg-append-new **0.02ms 0.02ms** 0.18ms svg-no-default-namespace 0.05ms 0.09ms **0.04ms** svg-xlink 0.01ms 0.05ms **0.00ms** tag-to-text **0.00ms 0.00ms** 0.01ms text-to-tag **0.00ms 0.00ms** 0.01ms text-to-text **0.00ms** 0.01ms 0.00ms textarea **0.01ms** 0.02ms 0.01ms todomvc 0.55ms 3.05ms **0.32ms** todomvc2 **0.05ms** 0.07ms 0.09ms two **0.01ms** 0.02ms **0.01ms**

NOTE: Chrome 72.0.3626.121

Benchmarks

Below are the results on running benchmarks on various DOM transformations for both `morphdom`, `nanomorph` and virtual-dom. This benchmark uses a high performance timer (i.e., `window.performance.now()`) if available. For each test the benchmark runner will run `100` iterations. After all of the iterations are completed for one test the average time per iteration is calculated by dividing the total time by the number of iterations.

To run the benchmarks:

```
npm run benchmark
```

And then open the generated `test-page.html` in the browser to view the results:

```
file:///HOME/path-to-morphdom/test/
mocha-headless/generated/test-page.html
```

The table below shows some sample benchmark results when running the benchmarks on a MacBook Pro (2.3 GHz Intel Core i5, 8 GB 2133 MHz LPDDR3). Remember, as noted above, the larger the diff needed to evaluate, the more vdom will perform better. The average time per iteration for each test is shown in the table below:

- Total time for morphdom: 820.02ms
- Total time for virtual-dom: 333.81ms (winner)
- Total time for nanomorph: 3,177.85ms

API

morphdom(fromNode, toNode, options) : Node

The `morphdom(fromNode, toNode, options)` function supports the following arguments:

- *fromNode* (`Node`)- The node to morph
- *toNode* (`Node|String`) - The node that the `fromNode` should be morphed to (or an HTML string)
- *options* (`Object`) - See below for supported options

The returned value will typically be the `fromNode`. However, in situations where the `fromNode` is not compatible with the `toNode` (either different node type or different tag name) then a different DOM node will be returned.

Supported options (all optional):

- **getNodeKey** (`Function(node)`) - Called to get the `Node`'s unique identifier. This is used by `morphdom` to rearrange elements rather than creating and destroying an element that already exists. This defaults to using the `Node`'s `id` property. (Note that form fields must not have a `name` corresponding to forms' DOM properties, e.g. `id`.)
- **addChild** (`Function(parentNode, childNode)`) - Called when adding a new child to a parent. By default,

`parentNode.appendChild(childNode)` is invoked. Use this callback to customize how a new child is added.

- **onBeforeNodeAdded** (`Function(node)`) - Called before a `Node` in the `to` tree is added to the `from` tree. If this function returns `false` then the node will not be added. Should return the node to be added.
- **onNodeAdded** (`Function(node)`) - Called after a `Node` in the `to` tree has been added to the `from` tree.
- **onBeforeElUpdated** (`Function(fromEl, toEl)`) - Called before a `HTMLElement` in the `from` tree is updated. If this function returns `false` then the element will not be updated. if this function returns an instance of `HTMLElement`, it will be used as the new fromEl tree to proceed with morphing for that branch, otherwise the current fromEl tree is used.
- **onElUpdated** (`Function(el)`) - Called after a `HTMLElement` in the `from` tree has been updated.
- **onBeforeNodeDiscarded** (`Function(node)`) - Called before a `Node` in the `from` tree is discarded. If this function returns `false` then the node will not be discarded.
- **onNodeDiscarded** (`Function(node)`) - Called after a `Node` in the `from` tree has been discarded.
- **onBeforeElChildrenUpdated** (`Function(fromEl, toEl)`) - Called before the children of a `HTMLElement` in the `from` tree are updated. If this function returns `false` then the child nodes will not be updated.
- **childrenOnly** (`Boolean`) - If `true` then only the children of the `fromNode` and `toNode` nodes will be morphed (the containing element will be skipped). Defaults to `false`.

- **Integrated Haskell Platform** (`all versions`) - A complete platform for developing server-rendered web applications in Haskell.
- **morphdom-swap for htmx**) (`all versions`) - an extension that uses `morphdom` as the swapping mechanism for htmx.
- **simply.js**(`all versions`) - Simple web-component library for simple web-apps.

NOTE: If you are using a `morphdom` *in your project please send a PR to add your project here*

- **Catberry.js** (`v6.0.0+`) - Catberry is a framework with Flux architecture, isomorphic web-components and progressive rendering.
- **Composer.js** (`v1.2.1`) - Composer is a set of stackable libraries for building complex single-page apps. It uses morphdom in its rendering engine for efficient and non-destructive updates to the DOM.
- **yo-yo.js** (`v1.2.2`) - A tiny library for building modular UI components using DOM diffing and ES6 tagged template literals. yo-yo powers a tiny, isomorphic framework called choo (`v3.3.0`), which is designed to be fun.
- **vomit.js** (`v0.9.19`) - A library that uses the power of ES6 template literals to quickly create DOM elements that you can update and compose with Objects, Arrays, other DOM elements, Functions, Promises and even Streams. All with the ease of a function call.
- **CableReady**(`v4.0+`) - Server Rendered SPAs. CableReady provides a standard interface for invoking common client-side DOM operations from the server via ActionCable.
- **Integrated Haskell Platform**(`all versions`) - A complete platform for developing server-rendered web applications in Haskell.
- **Ema**(`all versions`) - A change-aware static site generator library for Haskell. morphdom is used to provide hot reload in the live server.
- **CableReady** (`v4.0+`) - Server Rendered SPAs. CableReady provides a standard interface for invoking common client-side DOM operations from the server via ActionCable.

- **skipFromChildren** (`Function(fromEl)`) - called when indexing a the `fromEl` tree. False by default. Return `true` to skip indexing the from tree, which will keep current items in place after patch rather than removing them when not found in the `toEl`.

```
var morphdom = require('morphdom');
var morphedNode = morphdom(fromNode,
        toNode, {
  getNodeKey: function(node) {
    return node.id;
  },
  addChild: function(parentNode,
        childNode) {
    parentNode.appendChild(childNode);
  },
  onBeforeNodeAdded: function(node) {
    return node;
  },
  onNodeAdded: function(node) {

  },
  onBeforeElUpdated: function(fromEl,
        toEl) {
    return true;
  },
  onElUpdated: function(el) {

  },
  onBeforeNodeDiscarded: function(node)
        {
    return true;
  },
```

```
  onNodeDiscarded: function(node) {

  },
  onBeforeElChildrenUpdated:
        function(fromEl, toEl) {
    return true;
  },
  childrenOnly: false,
  skipFromChildren: function(fromEl,
        toEl) {
    return false;
  }
});
```


Widgets) and React we found that Marko was able to render pages ten times faster than React with much lower CPU usage (see: Marko vs React: Performance Benchmark)

A good strategy to optimize for performance is to render a template to an HTML string on the server, but to compile the template such that it renders to a DOM/virtual DOM in the browser. This approach offers the best performance for both the server and the browser. In the near future, support for rendering to a virtual DOM will be added to the Marko templating engine.

What projects are using `morphdom`?

`morphdom` is being used in the following projects:

- **Phoenix Live View** (`v0.0.1+`) - Rich, real-time user experiences with server-rendered HTML
- **TS LiveView** (`v0.1.0+`) - Build SSR realtime SPA with Typescript
- **Omi.js** (`v1.0.1+`) - Open and modern framework for building user interfaces.
- **Marko Widgets** (`v5.0.0-beta+`) - Marko Widgets is a high performance and lightweight UI components framework that uses the Marko templating engine for rendering UI components. You can see how Marko Widgets compares to React in performance by taking a look at the following benchmark: Marko vs React: Performance Benchmark

- The virtual DOM representations are not standardized and will vary by implementation
- The virtual DOM can only efficiently be used with code and templating languages that produce a virtual DOM tree

The premise for using a virtual DOM is that the DOM is "slow". While there is slightly more overhead in creating actual DOM nodes instead of lightweight virtual DOM nodes, in practice there isnt much difference. In addition, as web browsers get faster the DOM data structure will also likely continue to get faster so there benefits to avoiding the abstraction layer.

Moreover, we have found that diffing small changes may be faster with actual DOM. As the diffing become larger, the cost of diffs slow down due to IO and virtual dom benefits begin to show.

See the Benchmarks below for a comparison of `morphdom` with virtual-dom.

Which is better: rendering to an HTML string or rendering virtual DOM nodes?

There are many high performance templating engines that stream out HTML strings with no intermediate virtual DOM nodes being produced. On the server, rendering directly to an HTML string will *always* be faster than rendering virtual DOM nodes (that then get serialized to an HTML string). In a benchmark where we compared server-side rendering for Marko (with Marko

FAQ

Can I make morphdom blaze through the DOM tree even faster? Yes.

```
morphdom(fromNode, toNode, {
    onBeforeElUpdated: function(fromEl,
        toEl) {
        // spec - https://
        dom.spec.whatwg.org/#concept-
        node-equals
        if (fromEl.isEqualNode(toEl)) {
            return false
        }

        return true
    }
})
```

This avoids traversing through the entire subtree when you know they are equal. While we haven't added this to the core lib yet due to very minor concerns, this is an easy way to make DOM diffing speeds on par with virtual DOM.

Isn't the DOM slow?

UPDATE: As of `v2.1.0`, `morphdom` supports both diffing a real DOM tree with another real DOM tree and diffing a real DOM tree with a *virtual* DOM tree. See: docs/virtual-dom.md for more details.

No, the DOM *data structure* is not slow. The DOM is a key part of any web browser so it must be fast. Walking a DOM tree and reading the attributes on DOM nodes is *not* slow. However, if you attempt to read a computed property on a DOM node that requires a relayout of the page then *that* will be slow. However, `morphdom` only cares about the following properties and methods of a DOM node:

- `node.firstChild`
- `node.nextSibling`
- `node.nodeType`
- `node.nodeName`
- `node.nodeValue`
- `node.attributes`
- `node.value`
- `node.selected`
- `node.disabled`
- `actualize(document)` (non-standard, used to upgrade a virtual DOM node to a real DOM node)
- `hasAttributeNS(namespaceURI, name)`
- `isSameNode(anotherNode)`

What about the virtual DOM?

Libraries such as a React and virtual-dom solve a similar problem using a *Virtual DOM*. That is, at any given time there will be the *real* DOM (that the browser rendered) and a lightweight and persistent virtual DOM tree that is a mirror of the real DOM tree. Whenever the view needs to update, a new *virtual* DOM tree is rendered. The new virtual DOM tree is then compared with the old virtual DOM tree using a diffing algorithm. Based on the differences that are found, the *real* DOM is then "patched" to match the new virtual DOM tree and the new virtual DOM tree is persisted for future diffing.

Both `morphdom` and virtual DOM based solutions update the *real* DOM with the minimum number of changes. The only difference is in how the differences are determined. `morphdom` compares real DOM nodes while `virtual-dom` and others only compare virtual DOM nodes.

There are some drawbacks to using a virtual DOM-based approach even though the *Virtual DOM* has improved considerably over the last few years:

- The real DOM is not the source of truth (the persistent virtual DOM tree is the source of truth)
- The real DOM *cannot* be modified behind the scenes (e.g., no jQuery) because the diff is done against the virtual DOM tree
- A copy of the real DOM must be maintained in memory at all times (albeit a lightweight copy of the real DOM)
- The virtual DOM is an abstraction layer that introduces code overhead