

---

**This library is no longer maintained, feel free to fork and update as per the license**

## **transducers-js**

A high performance Transducers implementation for JavaScript.

Transducers are composable algorithmic transformations. They are independent from the context of their input and output sources and specify only the essence of the transformation in terms of an individual element. Because transducers are decoupled from input or output sources, they can be used in many different processes - collections, streams, channels, observables, etc. Transducers compose directly, without awareness of input or creation of intermediate aggregates.

For further details about Transducers see the following resources: \* “Transducers are coming” announce blog post \* Rich Hickey’s Transducers StrangeLoop presentation \* API Docs

transducers-js is brought to you by Cognitect Labs.

## **Releases and Dependency Information**

- Latest release: 0.4.180

### **JavaScript**

You can include either the release (2K gzipped) or development build of transducers-js on your webpage. We also provide Require.js compatible release and dev builds.

### **Node.js**

transducers-js is released to npm. Add transducers-js to your **package.json** dependencies:

```
1 {...
2   "dependencies": {
3     "transducers-js": "0.4.180"
4   }
5   ...}
```

---

## Bower

You can also include transducers-js in your `bower.json` dependencies:

```
1 {...
2   "dependencies": {
3     "transducers-js": "0.4.180"
4   }
5   ...}
```

## Usage

### Requiring

To import the library under Node.js you can just use `require`:

```
1 var t = require("transducers-js");
```

The browser release of the library simply exports a top level `transducers` object:

```
1 var t = transducers;
```

### Basic Usage

With <=ES5:

```
1 var map    = t.map,
2     filter = t.filter,
3     comp   = t.comp,
4     into   = t.into;
5
6 var inc = function(n) { return n + 1; };
7 var isEven = function(n) { return n % 2 == 0; };
8 var xf = comp(map(inc), filter(isEven));
9
10 console.log(into([], xf, [0,1,2,3,4])); // [2,4]
```

With ES6:

```
1 let {map, filter, comp, into} = t;
2
3 let inc = (n) => n + 1;
4 let isEven = (n) => n % 2 == 0;
5 let xf = comp(map(inc), filter(isEven));
6
7 console.log(into([], xf, [0,1,2,3,4])); // [2,4]
```

---

## Documentation

Documentation can be found [here](#)

## Integration

transducers-js can also easily be used in combination with *existing* reduce implementations, whether native or the shims provided by Underscore and Lodash. Doing so with native and Underscore can deliver significant performance benefits. Transducers may be easily converted from their object representation into the necessary two-arity function via [toFn](#).

```
1 var arr    = [0,1,2,3,4,5,6,7,8,9,10],
2   apush    = function(arr, x) { arr.push(x); return arr; },
3   xf       = comp(map(inc), filter(isEven)),
4   toFn     = t.toFn;
5
6 arr.reduce(toFn(xf, apush), []); // native
7 _(arr).reduce(toFn(xf, apush), []); // underscore or lodash
```

## Immutable-js

transducers-js can work with custom collection types and still deliver the same performance benefits, for example with Immutable-js:

```
1 var Immutable = require("immutable"),
2   t           = require("transducers-js"),
3   comp        = t.comp,
4   map         = t.map,
5   filter      = t.filter,
6   transduce   = t.transduce;
7
8 var inc = function(n) { return n + 1; };
9 var isEven = function(n) { return n % 2 == 0; };
10 var sum = function(a,b) { return a+b; };
11
12 var largeVector = Immutable.List();
13
14 for(var i = 0; i < 10000000; i++) {
15   largeVector = largeVector.push(i);
16 }
17
18 // built in Immutable-js functionality
19 largeVector.map(inc).filter(isEven).reduce(sum);
20
21 // faster with transducers
```

---

```
22 var xf = comp(map(inc), filter(isEven));
23 transduce(xf, sum, 0, largeVector);
```

## ES6 Collections

ES6 collections return iterators and therefore can be reduced/transduced. For example with transit-js collections which satisfy many of the proposed Map/Set methods:

```
1 var transit = require("transit-js"),
2   t         = require("transducers-js"),
3   m         = transit.map(["foo", "bar", "baz", "woz"]),
4   vUC       = function(kv) { return [kv[0], kv[1].toUpperCase()]; },
5   xf        = t.map(vUC);
6   madd      = function(m, kv) { m.set(kv[0], kv[1]); return m; };
7
8 transduce(xf, madd, transit.map(), m.entries()); // Map ["foo", "BAR",
    "baz", "WOZ"]
```

## The Transducer Protocol

It is a goal that all JavaScript transducer implementations interoperate regardless of the surface level API. Towards this end the following outlines the protocol all transducers must follow.

### Transducer composition

Transducers are simply a function of one arity. The only argument is another transducer *transformer* (labeled *xf* in the code base). Note the distinction between the *transducer* which is a function of one argument and the *transformer* an object whose methods we'll describe in the following section.

For example the following simplified definition of *map*:

```
1 var map = function(f) {
2   return function(xf) {
3     return Map(f, xf);
4   };
5 };
```

Since transducers are simply functions of one argument they can be composed easily via function composition to create transformer pipelines. Note that transducers return transformers when invoked.

---

## Transformer protocol

Transformers are objects. They must implement 3 methods, `@@transducer/init`, `@@transducer/result` and `@@transducer/step`. If a transformer is intended to be composed with other transformers they should either close over the next transformer or store it in a field.

For example the `Map` transformer could look something like the following:

```
1 var Map = function(f, xf) {
2   return {
3     "@@transducer/init": function() {
4       return xf["@@transducer/init"]();
5     },
6     "@@transducer/result": function(result) {
7       return xf["@@transducer/result"](result);
8     },
9     "@@transducer/step": function(result, input) {
10      return xf["@@transducer/step"](result, f(input));
11    }
12  };
13 };
```

Note how we take care to call the next transformer in the pipeline. We could have of course created `Map` as a proper JavaScript type with prototype methods - this is in fact how it is done in `transducers-js`.

## Reduced

Detecting the reduced state is critical to short circuiting a reduction/transduction. A reduced value is denoted by any JavaScript object that has the property `@@transducer/reduced` set to **true**. The reduced value should be stored in the `@@transducer/value` property of this object.

## Iteration

Anything which implements `@@iterator` which returns an ES6 compliant iterator is reducible/transducible. An ES6 iterator may also just be given directly to `reduce` or `transduce`.

## Building

Fetch the dependencies:

```
1 bin/deps
```

---

To build for Node.js

```
1 bin/build_release_node
```

To build for the browser

```
1 bin/build_release_browser
```

## Running the tests

Make sure you've first fetched the dependencies, then:

```
1 bin/test
```

## Contributing

This library is open source, developed internally by Cognitect. Issues can be filed using GitHub Issues.

This project is provided without support or guarantee of continued development. Because transducers-js may be incorporated into products or client projects, we prefer to do development internally and do not accept pull requests or patches.

## Copyright and License

Copyright © 2014-2015 Cognitect

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

```
1 http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.