
Quiescent

A lightweight ClojureScript abstraction over ReactJS, emphasizing its ability to (re)render immutable values efficiently.

An obligatory TodoMVC implementation is available.

See the documentation for instructions and examples of how to use Quiescent.

Rationale

ReactJS is an extremely interesting approach to UI rendering for HTML-based web applications. Its core value proposition is to make it possible to write one set of UI rendering code that works for both initial creation and any future updates. Updates are extremely efficient, because only the minimal set of necessary deltas are actually sent to the DOM.

In other words, React lets you write your UI code *once*, and still have a dynamic, data-driven application. Ideally, it eliminates any need to write explicit UI manipulation code. Instead, you are responsible only for supplying a specification of what the UI should look like, given certain data. The ReactJS implementation takes care of all the details of how to morph the existing DOM to correspond to changes in the input data.

Quiescent is intended to expose this feature and *only* this feature in a way that is convenient and idiomatic to ClojureScript, while remaining highly efficient.

It has the following design goals:

- **single concern:** Quiescent is designed to do one thing: render and re-render HTML from immutable values. It does not attempt to participate in state management, inform how you organize your application, or force the use of any specific technique for managing state updates. Feel free to use atoms, watchers, core.async, compilation-based models, message-passing, etc. The only requirement is that values passed to Quiescent components are immutable value types.
- **avoid OO idioms:** ReactJS is itself highly object-oriented, with stateful objects that may implement a variety of behaviors. Quiescent provides a purely functional interface, allowing users to construct a ReactJS component tree using only the basic tools of functional programming: function definition and composition.
- **top-down rendering:** All renders and updates are initiated by instructing Quiescent to render a particular value to a particular location in the DOM. Individual tree components do not maintain their own state and do not re-render themselves unless explicitly directed to do so.

This is intended to emulate classical ‘rendering’ semantics for 2D or 3D bitmapped applications, where the entirety of a scene is re-drawn from scratch for every frame. Of course, React/Quiescent does not actually do this, for performance reasons, but the fact that it does not is an implementation detail; the conceptual model is the same.

- **leverage immutability:** By assuming that any value provided to a rendered component is immutable, Quiescent can prevent ReactJS from even calculating if it needs to render sub-trees that have not changed from the last time the application was updated. Since equality checks are highly efficient in ClojureScript, large application structures can be re-rendered frequently with almost no performance hit apart from that necessary to re-render leaf nodes that actually did change.
- **compatibility:** Although you will hopefully be able to write the vast majority of your application using Quiescent’s model, you can, if necessary, always fall back and use a raw ReactJS component (or, for that matter, a ReactJS component constructed using another ClojureScript interface). This is possible at any level of the rendering component tree.

These goals differ slightly from other ClojureScript interfaces to React, as described below.

Comparison with Om

Om is another ClojureScript interface to ReactJS, highly capable and well-designed. It provides categorically more features than Quiescent, at the cost of taking more control and specifying more rigidly the way application state is modeled, updated and re-rendered.

The most important conceptual distinctions are:

- To create an Om component you must implement a protocol; due to its relative lack of capability, Quiescent only requires components to provide a single render function.
- Om controls the primary application state atom and how it is updated. In Quiescent this is entirely the responsibility of the consumer.
- Om explicitly allows components to maintain local state, while Quiescent forbids this. In my opinion the benefits of requiring components to account for local state do not justify the pervasive cost to implementation.

Specifically, addressing the two reasons David Nolen gives for allowing application state:

1. *“Local state pollutes application data.”* An alternative view is that all data is application data, whether it is transient or persistent, important or insignificant, wherever it is actually located. There is little harm in including it in the top-level data structure, and if your

application does demand a hierarchy of data (persistent vs. transient, etc) for different purposes, it is better to model that relationship explicitly rather than leaving transient data tucked invisibly away in component state.

2. “*Local state is always up-to-date.*” This is not relevant in Quiescent’s render model because Quiescent does not manage state.

- Om components are always aware of their location in the primary application state, via a *cursor* (a hybrid of functional zippers and lenses). Quiescent components are not. This means that Om components can dispatch updates to “themselves”, whereas a DOM event handler function attached to a Quiescent component can only effect change by reaching back and doing something to the top-level application state (e.g., by sending a `core.async` message, swapping the top-level atom, etc).

This does, somewhat, negate the concept of component modularity; Quiescent’s contention is that the benefit of top-down, value-based rendering exceeds that of truly modular components.

Ultimately, though, Om is an excellent, well-thought-out library, and if your needs or design goals more closely align with its capabilities than with Quiescent, you should absolutely use it.

Comparison with Reagent

Reagent is another ClojureScript wrapper for ReactJS. It is, perhaps, *easier* than either Om or Quiescent and certainly the most readable of the three.

The key differences between Reagent and Quiescent (or Om, for that matter) are:

- Reagent defaults to a Hiccup-like syntax for component definitions.
- Reagent handles updates via a special version of an atom (a *reactive atom* or *ratom*). Whenever a component references a *ratom*, watches are established such that the component will re-render when the value of the *ratom* changes. As such, Reagent components are not driven by top-down data or a singular application state, but by whatever *ratoms* are referenced in their definition.
- Reagent, like Om, maintains full control of the render/re-render cycle.

Although I do not have as much first-hand experience with Reagent, it seems to be a very convenient approach, and if it meets your needs you should definitely give it a try.

Implementation

This section presumes familiarity with how ReactJS works.

In short, basic Quiescent components implement only two of ReactJS’s component lifecycle events:

-
1. *shouldComponentUpdate* is always predicated exclusively on whether the immutable value passed to the component constructor function has changed. If so, the component re-renders. If not, it doesn't.
 2. The implementation of *render* is provided as a function supplied by the user. The output of this render function is presumed to be a single ReactJS component, as it is in vanilla ReactJS. The function, however, is passed the immutable value that was used to construct the function. It is also passed any additional arguments that were provided to the component constructor.

See Also

- The ReactJS website for information on ReactJS and how it works.

CHANGE LOG

0.3.2

- Upgrade to React 15.1.0 (#58)
- Ignore *mask* from *cljs.core* to avoid compiler warning (#57)

0.3.1

- Add newly supported SVG tags which React supports (#43)

0.3.0

- Upgrade to React 0.14 (#56)
- Use symbol name for *defcomponent* as default React component name (#46)
- Make it easier to idiomatically use React components defined elsewhere (for example, React Bootstrap). (Inspired by #40)

0.2.0

Warning: This release contains breaking changes.

- **breaking change:** The primary namespace has been renamed from *quiescent* to *quiescent.core* to avoid a single-segment namespace, which could cause a variety of problems.

-
- **breaking change:** Quiescent now includes a transitive dependency to ReactJS using ClojureScript's new, expanded foreign library support (<https://github.com/clojure/clojurescript/wiki/Foreign-Dependencies>). You do not need to, and should not, include ReactJS either in your ClojureScript preamble nor in your HTML.
 - Support for new versions of ReactJS.
 - The “wrapper” functionality is deprecated (see explanation below).
 - Instead, you can now supply explicit lifecycle hooks when defining components.
 - Keys are now supported on custom components.
 - Added support for naming custom components, which will be helpful for debugging (including the debugging browser extensions provided by ReactJS).
 - Added support for ReactJS' animation extensions (<http://facebook.github.io/react/docs/animation.html>)
 - Added an examples directory
 - Added support for uncontrolled inputs (fixes #32 and #36).

motivation for wrapper deprecation As it turns out, there is no way in the current model that wrappers can provide the desired functionality in all cases.

A wrapper component can only modify its *own* lifecycle methods, not truly those of its child. But neither can it access the “shouldComponentUpdate” of the child - it must have a “shouldComponentUpdate” that constantly returns true. Therefore, an “onRender” wrapper would fire *even if* the wrapped component explicitly did not render due to an unchanged value (or otherwise overriding “shouldComponentUpdate”).

0.1.2

- Issue #20 - Wrapper components now copy the “key” property from their wrappee.
- Issue #18 - Define a map of generated DOM functions. This useful for programmatically generated UIs.
- Issue #16 - Allow specification of multiple lifecycle handler functions using the same wrapper component. See documentation on [quiescent/wrapper](#).
- Project no longer has ReactJS as a leiningen dependency; clients are responsible for obtaining the most recent version of ReactJS and incorporating that in their own projects.

License

Copyright © 2014-2015 Luke VanderHart

Distributed under the Eclipse Public License (see LICENSE file).