Middleware · Redux

Middleware

You've seen middleware in action in the Async Actions example. If you've used server-side libraries like Express and Koa, you were also probably already familiar with the concept of *middleware*. In these frameworks, middleware is some code you can put between the framework receiving a request, and the framework generating a response. For example, Express or Koa middleware may add CORS headers, logging, compression, and more. The best feature of middleware is that it's composable in a chain. You can use multiple independent third-party middleware in a single project.

Redux middleware solves different problems than Express or Koa middleware, but in a conceptually similar way. **It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.** People use Redux middleware for logging, crash reporting, talking to an asynchronous API, routing, and more.

This article is divided into an in-depth intro to help you grok the concept, and a few practical examples to show the power of middleware at the very end. You may find it helpful to switch back and forth between them, as you flip between feeling bored and inspired.

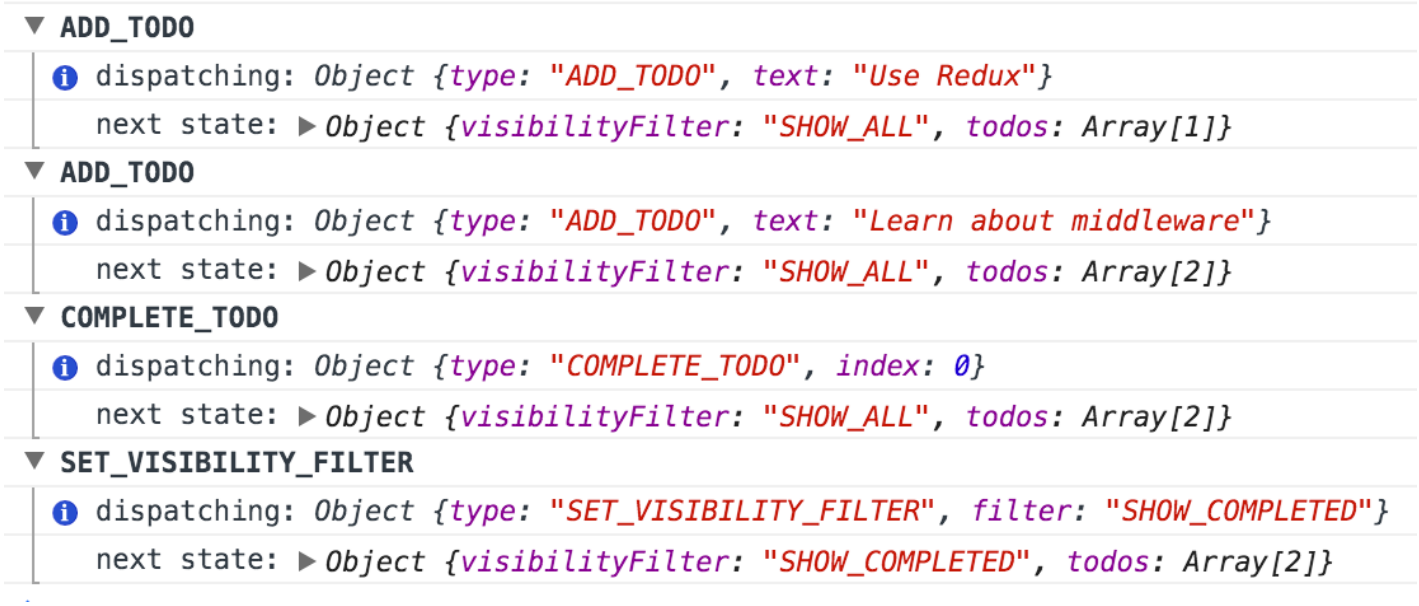
Understanding Middleware

While middleware can be used for a variety of things, including asynchronous API calls, it's really important that you understand where it comes from. We'll guide you through the thought process leading to middleware, by using logging and crash reporting as examples.

Problem: Logging

One of the benefits of Redux is that it makes state changes predictable and transparent. Every time an action is dispatched, the new state is computed and saved. The state cannot change by itself, it can only change as a consequence of a specific action.

Wouldn't it be nice if we logged every action that happens in the app, together with the state computed after it? When something goes wrong, we can look back at our log, and figure out which action corrupted the state.



How do we approach this with Redux?

Attempt #1: Logging Manually

The most naïve solution is just to log the action and the next state yourself every time you call `store.dispatch(action)`. It's not really a solution, but just a first step towards understanding the problem.

> **Note**
>
> If you're using react-redux or similar bindings, you likely won't have direct access to the store instance in your components. For the next few paragraphs, just assume you pass the store down explicitly.

Say, you call this when creating a todo:

```
store.dispatch(addTodo('Use Redux'))
```

To log the action and state, you can change it to something like this:

```
let action = addTodo('Use Redux')

console.log('dispatching', action)
store.dispatch(action)
console.log('next state', store.getState())
```

This produces the desired effect, but you wouldn't want to do it every time.

Attempt #2: Wrapping Dispatch

You can extract logging into a function:

```
function dispatchAndLog(store, action) {
  console.log('dispatching', action)
  store.dispatch(action)
  console.log('next state', store.getState())
}
```

You can then use it everywhere instead of `store.dispatch()`:

```
dispatchAndLog(store, addTodo('Use Redux'))
```

We could end this here, but it's not very convenient to import a special function every time.

Attempt #3: Monkeypatching Dispatch

What if we just replace the `dispatch` function on the store instance? The Redux store is just a plain object with a few methods, and we're writing JavaScript, so we can just monkeypatch the `dispatch` implementation:

```
let next = store.dispatch
store.dispatch = function dispatchAndLog(action) {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}
```

This is already closer to what we want! No matter where we dispatch an action, it is guaranteed to be logged. Monkeypatching never feels right, but we can live with this for now.

Problem: Crash Reporting

What if we want to apply **more than one** such transformation to `dispatch`?

A different useful transformation that comes to my mind is reporting JavaScript errors in production. The global `window.onerror` event is not reliable because it doesn't provide stack information in some older browsers, which is crucial to understand why an error is happening.

Wouldn't it be useful if, any time an error is thrown as a result of dispatching an action, we would send it to a crash reporting service like Sentry with the stack trace, the action that caused the error, and the current state? This way it's much easier to reproduce the error in development.

However, it is important that we keep logging and crash reporting separate. Ideally we want them to be different modules, potentially in different packages. Otherwise we can't have an ecosystem of such utilities. (Hint: we're slowly getting to what middleware is!)

If logging and crash reporting are separate utilities, they might look like this:

```
function patchStoreToAddLogging(store) {
  let next = store.dispatch
  store.dispatch = function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}

function patchStoreToAddCrashReporting(store) {
  let next = store.dispatch
  store.dispatch = function dispatchAndReportErrors(action) {
    try {
      return next(action)
    } catch (err) {
      console.error('Caught an exception!', err)
      Raven.captureException(err, {
        extra: {
          action,
          state: store.getState()
        }
      })
      throw err
    }
  }
}
```

If these functions are published as separate modules, we can later use them to patch our store:

```
patchStoreToAddLogging(store)
patchStoreToAddCrashReporting(store)
```

Still, this isn't nice.

Attempt #4: Hiding Monkeypatching

Monkeypatching is a hack. “Replace any method you like”, what kind of API is that? Let's figure out the essence of it instead. Previously, our functions replaced `store.dispatch`. What if they *returned* the new `dispatch` function instead?

```
function logger(store) {
  let next = store.dispatch

  // Previously:
  // store.dispatch = function dispatchAndLog(action) {

  return function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}
```

We could provide a helper inside Redux that would apply the actual monkeypatching as an implementation detail:

```
function applyMiddlewareByMonkeypatching(store, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()

  // Transform dispatch function with each middleware.
  middlewares.forEach(middleware =>
    store.dispatch = middleware(store)
  )
}
```

We could use it to apply multiple middleware like this:

```
applyMiddlewareByMonkeypatching(store, [ logger, crashReporter ])
```

However, it is still monkeypatching.

The fact that we hide it inside the library doesn't alter this fact.

Attempt #5: Removing Monkeypatching

Why do we even overwrite `dispatch`? Of course, to be able to call it later, but there's also another reason: so that every middleware can access (and call) the previously wrapped `store.dispatch`:

```
function logger(store) {
  // Must point to the function returned by the previous middleware:
  let next = store.dispatch

  return function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}
```

It is essential to chaining middleware!

If `applyMiddlewareByMonkeypatching` doesn't assign `store.dispatch` immediately after processing the first middleware, `store.dispatch` will keep pointing to the original `dispatch` function. Then the second middleware will also be bound to the original `dispatch` function.

But there's also a different way to enable chaining. The middleware could accept the `next()` dispatch function as a parameter instead of reading it from the `store` instance.

```
function logger(store) {
  return function wrapDispatchToAddLogging(next) {
    return function dispatchAndLog(action) {
      console.log('dispatching', action)
      let result = next(action)
      console.log('next state', store.getState())
      return result
    }
  }
}
```

It's a "we need to go deeper" kind of moment, so it might take a while for this to make sense. The function cascade feels intimidating. ES6 arrow functions make this currying easier on eyes:

```
const logger = store => next => action => {
```

```
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}
```

This is exactly what Redux middleware looks like.

Now middleware takes the `next()` dispatch function, and returns a dispatch function, which in turn serves as `next()` to the middleware to the left, and so on. It's still useful to have access to some store methods like `getState()`, so `store` stays available as the top-level argument.

Attempt #6: Naïvely Applying the Middleware

Instead of `applyMiddlewareByMonkeypatching()`, we could write `applyMiddleware()` that first obtains the final, fully wrapped `dispatch()` function, and returns a copy of the store using it:

```
// Warning: Naïve implementation!
// That's *not* Redux API.

function applyMiddleware(store, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()

  let dispatch = store.dispatch
  middlewares.forEach(middleware =>
    dispatch = middleware(store)(dispatch)
```

```
  )

  return Object.assign({}, store, { dispatch })
}
```

The implementation of `applyMiddleware()` that ships with Redux is similar, but **different in three important aspects**:

- It only exposes a subset of the store API to the middleware: `dispatch(action)` and `getState()`.
- It does a bit of trickery to make sure that if you call `store.dispatch(action)` from your middleware instead of `next(action)`, the action will actually travel the whole middleware chain again, including the current middleware. This is useful for asynchronous middleware, as we have seen previously.
- To ensure that you may only apply middleware once, it operates on `createStore()` rather than on `store` itself. Instead of `(store, middlewares) => store`, its signature is `(...middlewares) => (createStore) => createStore`.

Because it is cumbersome to apply functions to `createStore()` before using it, `createStore()` accepts an optional last argument to specify such functions.

The Final Approach

Given this middleware we just wrote:

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
```

```
    })
    throw err
  }
}
```

Here's how to apply it to a Redux store:

```
import { createStore, combineReducers, applyMiddleware } from 'redux'

let todoApp = combineReducers(reducers)
let store = createStore(
  todoApp,
  // applyMiddleware() tells createStore() how to handle middleware
  applyMiddleware(logger, crashReporter)
)
```

That's it! Now any actions dispatched to the store instance will flow through `logger` and `crashReporter`:

```
// Will flow through both logger and crashReporter middleware!
store.dispatch(addTodo('Use Redux'))
```

Seven Examples

If your head boiled from reading the above section, imagine what it was like to write it. This section is meant to be a relaxation for you and me, and will help get your gears turning.

Each function below is a valid Redux middleware. They are not equally useful, but at least they are equally fun.

```
/**
 * Logs all actions and states after they are dispatched.
 */
const logger = store => next => action => {
  console.group(action.type)
  console.info('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  console.groupEnd(action.type)
  return result
```

```
}

/**
 * Sends crash reports as state is updated and listeners are notified.
 */
const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}

/**
 * Schedules actions with { meta: { delay: N } } to be delayed by N milliseconds.
 * Makes `dispatch` return a function to cancel the timeout in this case.
 */
const timeoutScheduler = store => next => action => {
  if (!action.meta || !action.meta.delay) {
    return next(action)
  }

  let timeoutId = setTimeout(
    () => next(action),
    action.meta.delay
  )

  return function cancel() {
    clearTimeout(timeoutId)
  }
}

/**
 * Schedules actions with { meta: { raf: true } } to be dispatched inside a rAF
```

```
loop
 * frame.  Makes `dispatch` return a function to remove the action from the queue
in
 * this case.
 */
const rafScheduler = store => next => {
  let queuedActions = []
  let frame = null

  function loop() {
    frame = null
    try {
      if (queuedActions.length) {
        next(queuedActions.shift())
      }
    } finally {
      maybeRaf()
    }
  }

  function maybeRaf() {
    if (queuedActions.length && !frame) {
      frame = requestAnimationFrame(loop)
    }
  }

  return action => {
    if (!action.meta || !action.meta.raf) {
      return next(action)
    }

    queuedActions.push(action)
    maybeRaf()

    return function cancel() {
      queuedActions = queuedActions.filter(a => a !== action)
    }
  }
}

/**
```

```
 * Lets you dispatch promises in addition to actions.
 * If the promise is resolved, its result will be dispatched as an action.
 * The promise is returned from `dispatch` so the caller may handle rejection.
 */
const vanillaPromise = store => next => action => {
  if (typeof action.then !== 'function') {
    return next(action)
  }

  return Promise.resolve(action).then(store.dispatch)
}

/**
 * Lets you dispatch special actions with a { promise } field.
 *
 * This middleware will turn them into a single action at the beginning,
 * and a single success (or failure) action when the `promise` resolves.
 *
 * For convenience, `dispatch` will return the promise so the caller can wait.
 */
const readyStatePromise = store => next => action => {
  if (!action.promise) {
    return next(action)
  }

  function makeAction(ready, data) {
    let newAction = Object.assign({}, action, { ready }, data)
    delete newAction.promise
    return newAction
  }

  next(makeAction(false))
  return action.promise.then(
    result => next(makeAction(true, { result })),
    error => next(makeAction(true, { error }))
  )
}

/**
 * Lets you dispatch a function instead of an action.
 * This function will receive `dispatch` and `getState` as arguments.
```

```
 *
 * Useful for early exits (conditions over `getState()`), as well
 * as for async control flow (it can `dispatch()` something else).
 *
 * `dispatch` will return the return value of the dispatched function.
 */
const thunk = store => next => action =>
  typeof action === 'function' ?
    action(store.dispatch, store.getState) :
    next(action)


// You can use all of them! (It doesn't mean you should.)
let todoApp = combineReducers(reducers)
let store = createStore(
  todoApp,
  applyMiddleware(
    rafScheduler,
    timeoutScheduler,
    thunk,
    vanillaPromise,
    readyStatePromise,
    logger,
    crashReporter
  )
)
```