# Table of Contents


- Introduction 0
- Introduction to React 1
- Architecture 2
  - Introduction to Redux 2.1
- Hard and fast rules for apps 3
- State management: tools and utilities for keeping sane 4
  - Syncing data 4.1
  - Normalizer 4.2
  - Using Immutable.js Records 4.3
  - Transforming state per component 4.4
  - UI state 4.5
- Writing a Redux app 5
  - ES7, Harmony and Future JS 5.1
  - Scaffolding a new project 5.2
  - Data loading: building a JS SDK 5.3
  - Components 5.4
  - Sharing Components 5.5
  - Writing Middleware 5.6
  - Tests 5.7
  - Styles and Standards 5.8
- Process 6
  - Design 6.1
  - Handover 6.2
  - Development process 6.3
  - QA 6.4
  - Testing & Feedback 6.5
- Examples 7

# Redux without profanity

Welcome to a summary of how we write redux apps at Docker, Inc. aimed to be used as a guide for our projects and the Redux community. We hopped on the Redux train early at Docker. Along the way we've learned many lessons from the community, plus our own hefty trial and error.

It's a moving target which will be updated as new Redux patterns emerge internally and the community; things can always be improved and we'd love to keep doing things *The Right Way* (well, as right as we can make it).

## How this book is organized

This book has four main sections:

1. Introduction to React
2. Introduction to Redux
3. General rules for building redux apps
4. An example API & streaming app build from scratch

If you're new to React and Redux you might want to start from the beginning. With some experience you'll probably want to read the third section (general rules); this section is essentially best practices from all of our combined mistakes building past apps.

# Introduction to React

Facebook's React is "a library for user interfaces". Its single role is to manage and render your user interface. On its own it does nothing more... no communication with your API, no central state management, and no extra logic.

Facbeook's documentation and introduction to React is pretty thorough and worth a read. Because of that we'll jog through the key concepts to React and some tips to getting the most out of it.

That aside, let's talk about how React works and why it's taken the UI world by storm.

---

**Note**

This section assumes knowledge of HTML, CSS, and JavaScript (subsequently called JS), including ES6. If you don't know JS you should read http://eloquentjavascript.net/ before continuing. And you can brush up on ES6 with these resources: https://github.com/ericdouglas/ES6-Learning.

---

# How react works: components

The most important concept to react are **components**. Writing a component is like writing a group of HTML tags, or sometimes a single custom tag.

Here's a basic component defined using ES6 classes (this is the same as the React documentation):

```
import React from 'react';

/**
 * Rendering <HelloMessage text='Hello Sarah' /> results in this HTML:
 * <div>Hello Sarah</div>
 */

class HelloMessage extends Component {
  render() {
    return <div>{ this.props.text }</div>
  }
}
```

It's basic. We didn't discover a new particle. That said, the idea of a component is *so* powerful that it'll change the way you build UI — hopefully forever.

## Why components work

## Components are compositional

You can nest and combine components in any combination. You'll end up writing less code and you'll end up having less tests. It also means reusing components is *easy* – whether you want to reuse something across projects or in the same project.

## Components contain no state and no data logic

This makes writing components similar to writing basic HTML. They're easy to write, easy to test, and easy to come back to in a year's time and figure out just what the flying tosspot you were doing.

## Components have a basic lifecycle

There's no complex logic and no stream of events to follow. You'll be stoked to hear that you don't have to breakpoint and namespace *every single event* in your app and follow the chain 10 events deep to see why your views are updating.

## Components can be enhanced

You can enhance basic components by wrapping them with more complex parents. ES7 introduces decorators that allow us to do this with a single line. You essentially keep all of your hardcore logic within *higher-order components* — which only exist to add functionality to child components.

# Libraries and architecture

We use the following libraries and projects to build our UIs:

## Framework

- React: UI library
- Redux: state management and flux framework
- Normalizr: for data normalization
- Reselect: for passing data to components

Framework extensions:

- redux-form: form management within redux
- redux-ui: UI state management within redux

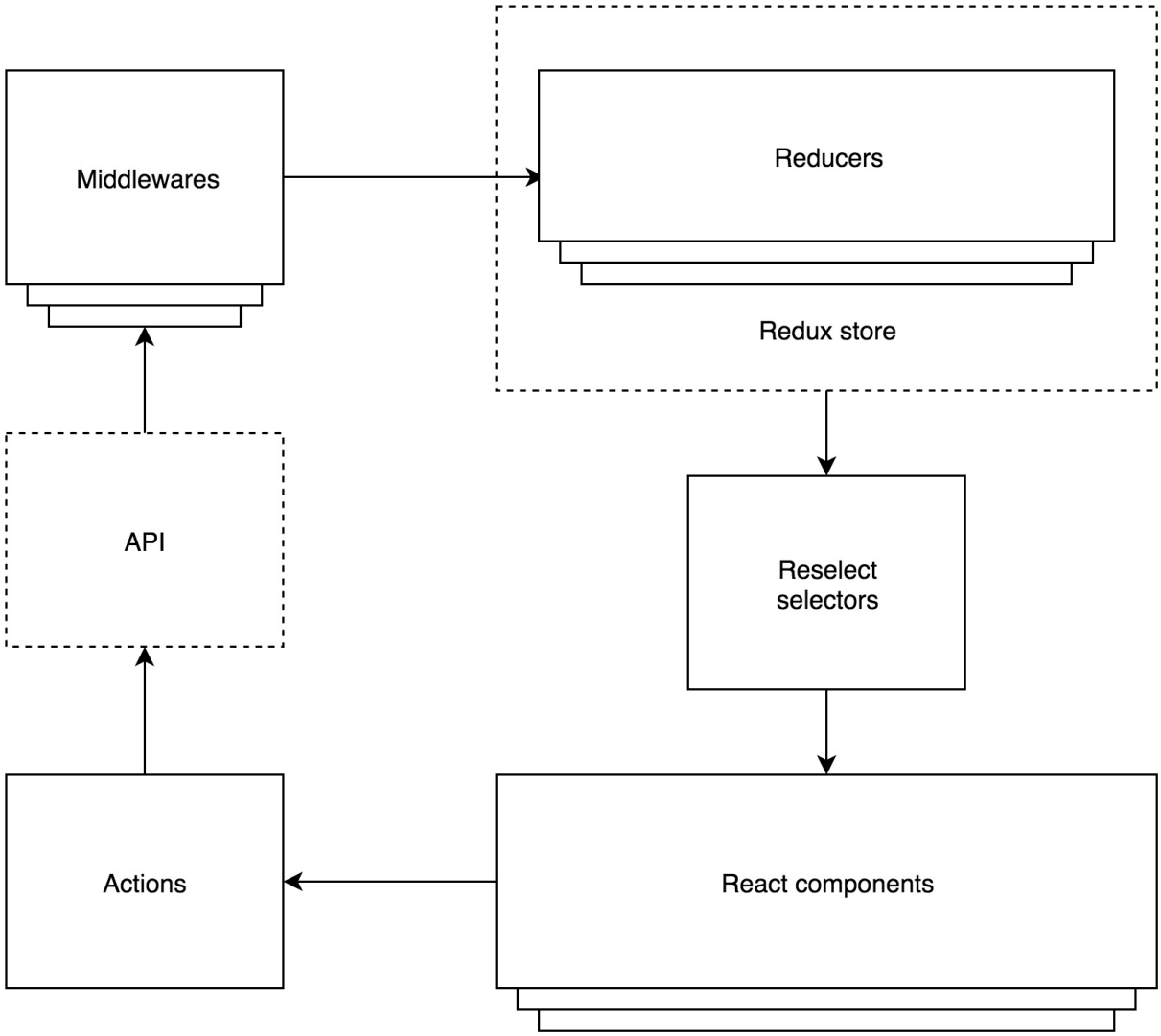
Tooling:

- webpack: module bundler
- postcss: css preprocess manager
- cssnext: css transpiling using future css syntax

This list isn't exhaustive though it comprises the most important tools in our stack. Each library solves a particular problem with frontend development; we'll walk through each of them in the coming chapter.

## Architecture

Redux follows the general flux architecture. All state is stored globally and is modified via *actions*. The flow of data and logic looks like this:




1. Starting at the bottom right we have components that can trigger *actions*.
2. *Actions* are flux' way of requesting that a state mutation happens. Each action may have a side effect like an API request
3. The action passes through *middleware*. Middleware handle action-specific tasks like listening to promise statuses. Middleware can also trigger new *actions*.
4. Once actions flow through the middleware they're passed into *reducers*
5. *Reducers* receive the *action* and any associated data. Reducers can choose to change global state within the store by returning new state.
6. Any state changes propagate to *reselect selectors*. The *selectors* transform data and pass it to *components*
7. *Components* receive the state they need as props, and render HTML to the DOM.

It's important to understand this flow, and it's even more important to know why we stick to it.

## Architecture benefits

This architecture has several benefits over classical frameworks such as Ember or Angular.

Implementing **redux** as 'flux' framework means that all data flows in a single direction. Having this means:

1. There are no side effects in components; components are predictable and understandable
2. Components can't directly modify state, so there's no need for data binding or event systems
3. It's hard to fall out of sync and mismanage state as all state is managed globally
4. Middleware can handle complex side effects such as listening to promise statuses

We've added **reselect** as an intermediary between the redux store and components; classically components get state directly from the store. **Reselect** queries and transforms data for each component. This means that each component is entirely reusable because they *only display data they're given*. We talk about this more in the 'transforming state per component' chapter.

## Future architecture plans

Fetching and normalizing data from each component using actions works, though it isn't declarative.

Ideally components would declare data they need without worrying about fetching data. We'd then have some background logic that takes care of API requests, caching, and data transformation on the component's behalf.

This is the idea behind Falcor and Relay. These mean that each component can be completely declarative, making things even easier to build and understand.

# Hard and fast rules for Redux apps

We've had more than our fair share of messy code and mishaps. While working through these we've learned some general rules to follow when you're building react apps. Follow these and your life will include less profanity than ours (while building your UI at least).

## State management

1. Use Falcor *or* normalizr + reselect to normalize state shape in reducers
2. Store each resource:
    i. using immutable.js
    ii. in its own reducer
    iii. store the entire normalizr result, not just `entities`
    iv. think about storing data within maps of parent resources (ie. for github, store repositories keyed by user/org names: `{ 'docker': data }` vs storing just normalized data)
3. Middleware that records state should have their own reducers
4. UI state should be managed globally via something like redux-ui
5. Use Immutable records where possible

## Writing code

1. Use ES7
2. Most actual code should be written as:
    i. Redux middleware
    ii. Reselect selectors (for querying and transforming state/resources)
    iii. Higher-order-components - usually as decorators
3. This means you should almost *never* write logic in a plain UI component

## UI components

1. Should always (try and) be dumb (ie. have no state)
    i. If components need state it should be global but ephemeral (see redux-ui)
2. Should specify their props in propTypes (eslint enforced)
3. Should have their own CSS files included via postcss + react-css-modules + style-loader
4. Should always load data declaratively if possible
5. Should **never** transform data itself; to transform data use reselect

## Higher order components

1. Should always use ES7 decorators for application
2. May maintain internal state depending on complexity and needs
3. By nature may contain business logic

## Selectors

1. Separate selectors into files ( `users` , `posts` , `router` etc)
2. Use a router selector for query state and params from the router: this will be used in other selectors and will become a dependency
3. Prefix selectors with `get` : `getCurrentUser` for the logged in user, `getRouteUser` for the user from the current router param etc.

## Misc: forms, other redux components

1. Use redux-form.

## Code structure

```
actions/
components/
  hoc/
  elements/
middleware/
reducers/
scenes/
selectors/
utils/
*.js
```

Example:

```
actions/
  index.js
  posts.js
  ui.js
components/
  hoc/
    modal/
      modal.js
      modal.css
  elements/
    button/
      button.js
      button.css
middleware/
  promise.js
reducers/
  posts.js
  promise.js <- from middleware/promise.js for tracking statuses
  ui.js      <- stores all UI state for each component
scenes/
  dashboard/
    index.js
    dashboard.js
    dashboard.css
    selectors.js <- component-specific selectors
  posts/
    index.js
    posts.js
    posts.css
  post/
    index.js
    post.js
    post.css
selectors/
  posts.js     <- general selectors (if desired)
utils/
  index.js
container.css
container.js
consts.js      <- consts used throughout the app
index.js       <- loads ReactDOM and injects routes
normalizers.js <- Normalizr configruation
records.js     <- Immutable.js record definitions
routes.js
```

# State management

The toughest thing to get right is state management. Unfortunately that's a lesson we learned over time, and without good planning you can lose your sanity.

Luckily, Redux/React is so good at abstracting away typical frontend work that we only need to focus on how to manage state.

## Isn't redux meant to solve state issues?

Yes. Kind of. Well, not really. Redux is a suite to manage state. It gives you a global state store; a connector to pass state into components; middleware; and actions.

It does **not** give you any rules for structuring your state. Or for requesting data from the server within nested components. Or for managing UI state across unrelated components.

## State problems

These are the core state issues that you'll to run in to when building apps:

1. Syncing data with the server
    i. Tracking AJAX request statuses
    ii. Deduping actions within child components
    iii. Cache invalidation
    iv. Universal apps
2. Transforming state per component
3. UI state

We tackle each of these problems in the following subchapters.

# Syncing data

Every app we build retrieves its data from the single source of truth — the server. React solves UI issues such as event delegation, code reuse, architecture and data binding. It leaves data syncing as the last major unsolved issue; syncing data in an easy and optimized manner is what new frameworks like Falcor and Relay attempt to solve.

## Needs (and the problems they bring about)

The basic need is to **ensure each component receives the data it needs to render correctly**. This need hides other implications:

1. We must be able to track and show loading statuses
2. We must either:
    i. be able to request data on a component-by-component basis (declaratively)
    ii. pass down requested data from parents to children continually
    iii. If multiple components request the same data that request should be optimized (ie. only requested once)
3. We should be able to track cache statuses within our reducer
4. Data loading should work for universal apps rendered server side

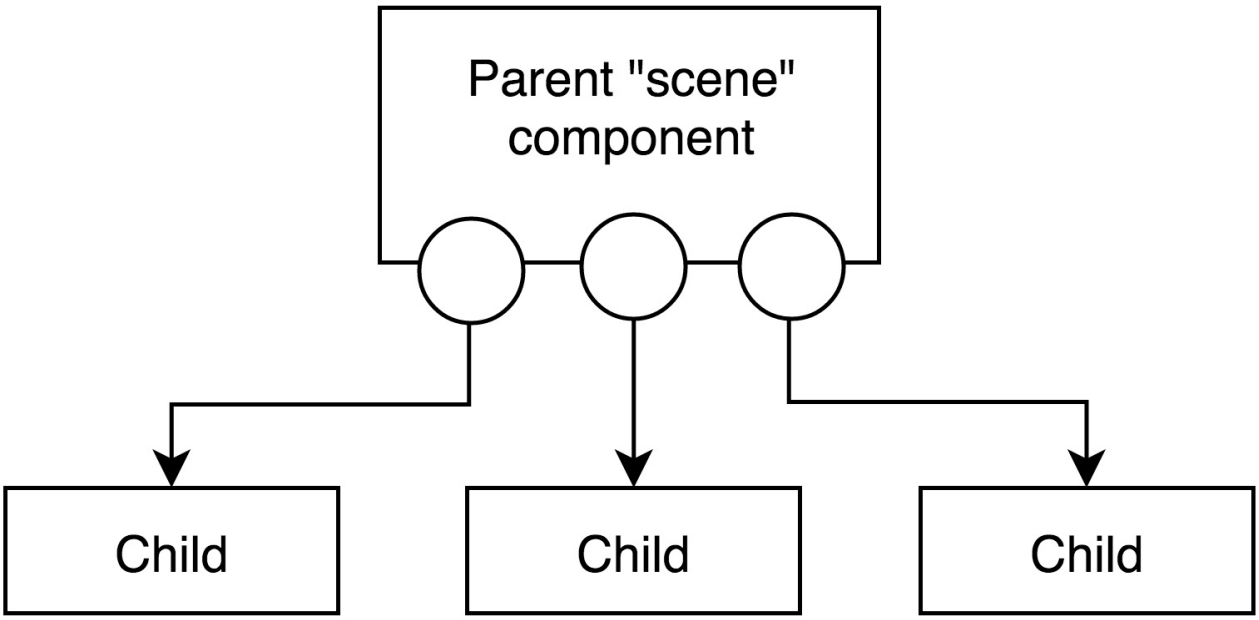
#### Need 1: Loading statuses

Tracking the status of AJAX requests is the only way we can accurately show loading statuses in our UI. The simplest (and most typical) case is tracking whether an individual request is one of `PENDING`, `SUCCESS` or `ERROR`.

When tracking a series of events, however, things are more difficult. If we `debounce` a user's search to show live results we need to ensure that **only the latest request is tracked**. A user would expect to see search results for the current input value - not for a prior search term.

#### Need 2: Data loading and architecture

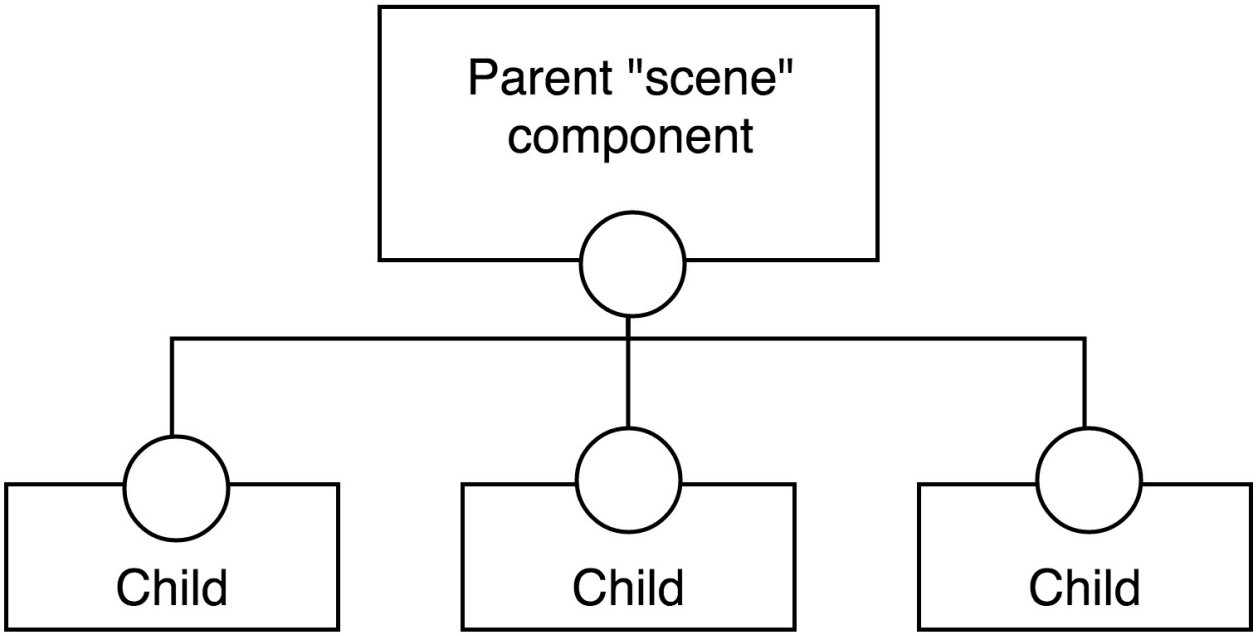
There are two architectural styles for loading data within your Redux app. The first is to load all data in a parent component and pass data down to children (eg. within a route or the parent scene component):




This is often the simplest method for small scenes, involving queries triggered from `componentWillMount` and `componentWillUpdate`. Pros and cons are:

- `+` It eliminates request duplication as there's a single component making requests.
- `+` It is conceptually easy to understand
- `-` Passing data down from parent to child becomes tedious and verbose.
- `-` Dependent data loading can become difficult (ie. one request depends on another request)

The second is to make each component request the data they need declaratively:




In this example, components request data separately while still being able to pass data to their children.

- `+` It is easy to read, write, refactor and update these components

- `+` Each component becomes fully independent and decoupled
- `-` Naive implementations can duplicate requests

The trend towards declarative data loading favours this model, mainly as this is easier to work with. Newer React frameworks such as Falcor, GraphQL and Resolver also batch and dedupe requests automatically. It's also possible to implement using plain Redux actions combined with autoaction.

**Need 3: Data caching**

In Redux our reducers store a local copy of the server's state; that is, they cache data from the source of truth. As soon as we store the data in Redux it's possible that it is out of date.

If a component requests data we need to:

1. Determine whether we have the data cached in our store already
2. Determine whether the cached data is stale
3. Make a new request for the data

Caching is a hard problem. It's often easier (but less performant) to assume that all existing data is stale and re-request data than to implement a caching layer that works well.

## Need 4: Universal data loading

The data loading methods used should work the same when rendered universally on both the client and server.

## Tools for syncing data

- https://netflix.github.io/falcor/
- https://github.com/graphql/graphql-js
- https://github.com/ericclemmons/react-resolver
- https://github.com/andreypopp/react-async
- https://github.com/rackt/async-props
- https://github.com/tonyhb/autoaction

# Normalizer

The idea behind normalizr is to take an API response that has nested resources and flatten them. It's a simple idea with a great upside - **it becomes much easier to query and manipulate data for your components**. And this is even better combined with reselect, which we'll talk about soon.

Some quick rules and reasoning for using Normalizr:

1. Use it directly within a callback from your API in your actions
2. After normalizing data convert entities to Immutable.js records. We'll introduce these in the next chapter
3. Pass the normalized records to reducers. Reducers should only store these
4. When reducers only store normalized records it becomes super easy to query using reselect.

Essentially, Normalizr is a utility for making Reselect even easier.

## An example: building a blog

Let's say you're building a blog, and the API response for `/posts/1` lists the post. The post embeds the post author and all comments, then each comment embeds an author. It might look like this:

```
{
    "id": 1,
    "title": "Blog Title",
    "content": "Some really short blog content.  Actually the least interesting post ever
    "created_at": "2016-01-10T23:07:43.248Z",
    "updated_at": ""2016-01-10T23:07:43.248Z",
    "author": {
        "id": 81,
        "name": "Mr Shelby"
    }
    "comments": [
        {
            "id": 352,
            "content": "First!",
            "author": {
                "id": 41,
                "name": "Foo Bar"
            }
        }
    ]
}
```

If we wanted to pull out all comment authors for a component we'd have to request the post, iterate through each comment, and pull out `author.name`. This gets tedious fast, and it's even worse with more deeply-nested resources.

Let's see how normalizr improves this.

### Setting up normalizr

The first thing to do so we can normalize this API response is set up normalizr. This means we need to describe our resource relationships:

```
'use strict';

import { Schema, arrayOf } from 'normalizr';

// Here we create the normalizr schemas.  These define the entity names
// (entities.posts, for example) and how we get the ID for each entity.
//
// We could define a function for idAttribute which returns a key (such as
// combining two fields).
const postSchema = new Schema('posts', { idAttribute: 'id' });
const postAuthorSchema = new Schema('postAuthors', { idAttribute: 'id' });
const commentSchema = new Schema('comments', { idAttribute: 'id' });
const CommentAuthorSchema = new Schema('commentAuthors', { idAttribute: 'id' });

// Here we define relationships between each resource/schema/entity
// (or whatever you feel like calling them these days)

// The post resource in our API response has an author and comments as children
postSchema.define({
    author: postAuthorSchema,
    comments: arrayOf(commentSchema)
});

// Each comment has an author
commentSchema.define({
    author: commentAuthorSchema
});
```

Now this is set up Normalizr can do it's thing. When we call normalizr on a post it'll recursively extract all data. Calling normalizr like so:

```
normalize(response.body, postSchema);
```

Will produce this:

```
{
    result: [1],
    entities: {
        posts: {
            1: {
                "id": 1,
                "title": "Blog Title",
                "content": "Some really short blog content.  Actually the least
                "created_at": "2016-01-10T23:07:43.248Z",
                "updated_at": ""2016-01-10T23:07:43.248Z",
                "author": 81
                "comments": [352]
            }
        },
        postAuthors: {
            81: {
                "id": 81,
                "name": "Mr Shelby"
            }
        },
        comments: {
            352: {
                "id": 352,
                "content": "First!",
                "author": 41
            }
        },
        commentAuthors: {
            41: {
                "id": 41,
                "name": "Foo Bar"
            }
        }
    }
}
```

Kinda magic! Let's see the three core features:

1. Each resource type is yanked into the entities object.
2. The 'Post' resource no longer embeds each comment. Instead it's replaced with an array of comment IDs. This makes it easy and fast to look up comments from entities.comments.
3. `results` stores an array of IDs of posts in the order of the API response (objects do not guarantee orders)

**Benefits**

Let's say we now wanted to list all comment authors. Instead of recursing through each comment and pulling the authors we already have a list defined. It's super easy to do. It's super easy to figure out just how many comments we also have.

**Caveats**

### Maintaining API ordering

Normalizr places your data inside an object within `entities`. Unfortunately JS objects **do not guarantee any specific order**. If you're normalizing a search result, for example, this might completely break the search results you display.

To maintain the order of data from the API you must **iterate through the `result` property of the normalized object**. This is an array which lists the IDs of entities from the search result. And arrays safely guarantee the order of its values.

If you're iterating through an embedded resource normalizr replaces these with an array of IDs also in the same order as the API — see below.

For example:

```
{
    // This is an array of the top-level schema passed to normalize()
    result: [1],
    entities: {
        posts: {
            1: {
                ...
                // This is an array of comment IDs in the same order
                // as the API. Iterate through entities.comments to pull
                // the resources out.
                "comments": [352]
            }
        },
        comments: {...}
        ...
    }
}
```

And here's some code to pull this out of the `posts` reducer:

```
// Returns an array of page entities in order from the API response
const getPages = (state) => {
    const result = state.posts.get('result');
    return result.map(id => state.entities.getIn(['posts', id]));
}
```

# Using Immutable.js Records

Immutable.js records are an awesome addition to your app. Why?

1. **They're self-documenting**. Any developer will be able to see all the fields within a resource/record
2. **They enforce strict code quality**. You can't add undocumented fields to a record.
3. **They combine the benefits of ImmutableJS' `Map()` with normal JS objects**. They have standard accessors ( `post.name` vs `post.get('name')` ), you can use destructuring, and they have Immutable's strict equality.
4. You can use `getIn` within maps and records. If a map contains records you can read record's values using `getIn` .

## Setting up Records

Immutable records are easy to set up and define. For each resource in your app you define a new record as follows:

```
'use strict';

import { Record, Map } from 'immutable';

export const Post = new Record({
    id: undefined,
    title: '',
    content: '',
    author: undefined
    comments: new Map()
});
```

So you're passing in an object to define a record. The keys of this object are the *only* fields you can set on the new record; the values are used as defaults. Which means you can't set `post.foo` - it'll fail.

All said, because you need to explicitly state each field as you define a record it becomes self-documenting. Any developer joining your team can look at the record definitions to see which data to use in their selectors. This saves them from having to dig around in your API looking for responses.

## Using Records

Records are immutable. That means that once set you can't change any of the values. Instead, you can do this:

```
const post = new Post({ title: 'foo', content: 'misc' });
const edited = post.set('title', 'bar');
```

Records inherit all Immutable.Map functions such as `set`, `setIn`, `merge` etc, and each of these return a *new* instance of the record.

This also means that you can use `setIn` from a parent Map to access record values:

```
const data = new Map({
  somePost: new Post({ title: 'some post' })
});

console.log(data.getIn(['somePost', 'title'])); // === 'some post';
```

# Transforming state per component

Every component you write has different needs. Some might show data in the same format your as API, while others might need to amend, blend or selectively combine data for its UI.

According to our general rule, a component '*should never transform data itself*' as we '*should never write logic inside a UI component*'.

Here's why shouldn't you transform data in a component:

- Coupling your component to a particular data structure means your component **isn't reusable**
- The transformation itself is single-use (and may well be copied to another component)
- The transformation is harder to test inside a component
- It clutters components, making them far less readable

We get around this problem by using reselect, which sits in between your Redux store and your components to query and transform data.

## Transforming state via reselect

Reselect is a powerful library which allows you to build a query and transform pipeline for your components. It allows you to:

1. Create `selectors` - functions which select and return pieces of state
2. Combine `selectors` to return multiple resources at once
3. Modify data based on existing state

This allows you to process state in any manner for your components. As a bonus, reselect also memoizes each function's outputs for increased performance.

Say we have a route with the path `/posts/:slug`, and we're using redux-router to store route params in global state. We can then create a selector to query for the current post:

```
currentPageSelector = (state) => {
  // immutable.js uses "get" to query maps:
  // https://facebook.github.io/immutable-js/docs/#/Map/get
  return state.pages.get(state.router.params.slug, new PostRecord());
};
```

We can then combine this into another selector to return the current page and comments

```
// This returns an immutable map of all comments
allComments = state => state.comments;

currentPageWithComments = createSelector(
  [currentPageSelector, allComments],
  (page, comments) => {
    // We're sticking to our hard and fast rule of storing data in a map
    // keyed by IDs commonly used in selecting.
    // In this example we're storing all comments in a map keyed by page ID.
    let pageComments = comments.get(page.id, new Immutable.Map());
    return {
      page,
      comments: pageComments
    };
  }
);
```

This moves all state logic into separate reusable, performant, testable functions. For more information see the reselect documentation.

# UI state

UI state is imperative to get right within your app at an early stage. It has a direct impact on a user's experience and influences the architecture of your apps actions, reducers and other business logic.

## Needs

The fundamentals of UI state are different to normal state. **UI state is shared across the entire chrome of the browser**. This means that UI state for a given page should be shared and editable by all components. When adding UI state to a component state should:

1. Be stored in a reducer, to allow interactivity within decoupled components
2. Be automatically passed down to any component
3. Be easy to update in any component, via actions
4. Have a set of default props for the current view, preferably set in the component
5. Automatically reset to the default state when navigating away from the current view
6. Optionally persist state upon navigation

This is a complex set of requirements for transient stated. Using `setState` within a component doesn't meet many of our needs; it's hard to pass down as props and even harder to manipulate cleanly.

## Managing UI state with redux-ui

redux-ui is a higher-order decorator which wraps each component with UI state and actions to change state. It solves all the above needs, and works similarly to block level scoping.

## redux-ui fundamentals

In the following example each curly brace pair represents a React component, and each variable represents a piece of UI state:

```
{
  let display = "SHOW_ALL";
  let filter = '';

  {
    let isEditing = false;
  }
}
```

We'd like to define a parent view which has `display` and `filter` UI state. The child view defines its own `isEditing` state. Keeping block-scope fashion, the child view *can still access and change* UI variables from the parent. This makes editing UI state easy from any component.

Here's how we set it up within redux-ui:

```
// Use redux-ui's @ui decorator to create a new context for saving UI state
// in this component This is the root component of our app.
@ui({
  // Save all state within the 'contacts' key of the UI reducer
  key: "contacts",
  // Define default state. All state vars must be defined in a UI decorator
  state: {
    display: "SHOW_ALL", // enum, SHOW_ALL/SHOW_STARRED
    filter: '' // The search filter to live search contacts
  }
})
@connect(contactsSelector)
class Contacts extends Component {

  static propTypes = {
    ui: PropTypes.object,
    updateUI: PropTypes.func,
    contacts: PropTypes.object
  }

  updateFilter(evt) {
    this.props.updateUI('filter', evt.target.value);
  }

  // Return a function with a closure referencing the string to change to
  updateDisplay = (filter) => (evt) => {
    this.props.updateUI('display', filter);
  }

  render() {
    const { contacts, ui } = this.props;
    return (
      <div>
        <input type='text' value={ ui.filter } onChange={ ::this.updateFilter } />
        <a href='#' onClick={ ::this.updateDisplay('SHOW_ALL') }>All</a>
        <a href='#' onClick={ ::this.updateDisplay('SHOW_STARRED') }>Starred</a>
        { Object.keys(contacts).map(c => <ContactItem contact={ contacts[c] } />) }
      </div>
    );
  }
}

// This is our child component. It is given **no** key so each component will
// generate a key at instantiation. As there are many of these components each
```

```
// will have its own context; if we gave this a key each of these components
// would share UI state.
@ui({
  state: {
    isEditing: false
  }
})
class ContactItem extends Component {

  static propTypes = {
    ui: PropTypes.object,
    updateUI: PropTypes.func,
    contact: PropTypes.object.isRequired
  }

  render() {
    const { contact, ui } = this.props;
    return (
      <div>
        <p>{ contact.name }</p>
        {/* Note that this is also given all parents UI state */}
        {/* We could even call `updateUI('filter', 'something') to update it */}
        { (ui.filter !== '') && <small>Matches { ui.filter }</small> }
      </div>
    );
  }
}
```

In this small example we've achieved a lot via the decorator:

- Our views define UI state variables with defaults in the component
- We automatically receive UI state as props
- We're given an updateUI method to change UI state
- All child components inherit their parent's UI context
- Child components are still given their own UI context for independence
- The UI state for each component will be deleted on unmount
- We've written no reducers or actions ourselves

This leads to incredibly powerful, reusable componnets which still manipulate UI state such as drop-in pagination.

See redux-ui for more information.

# Writing a redux app

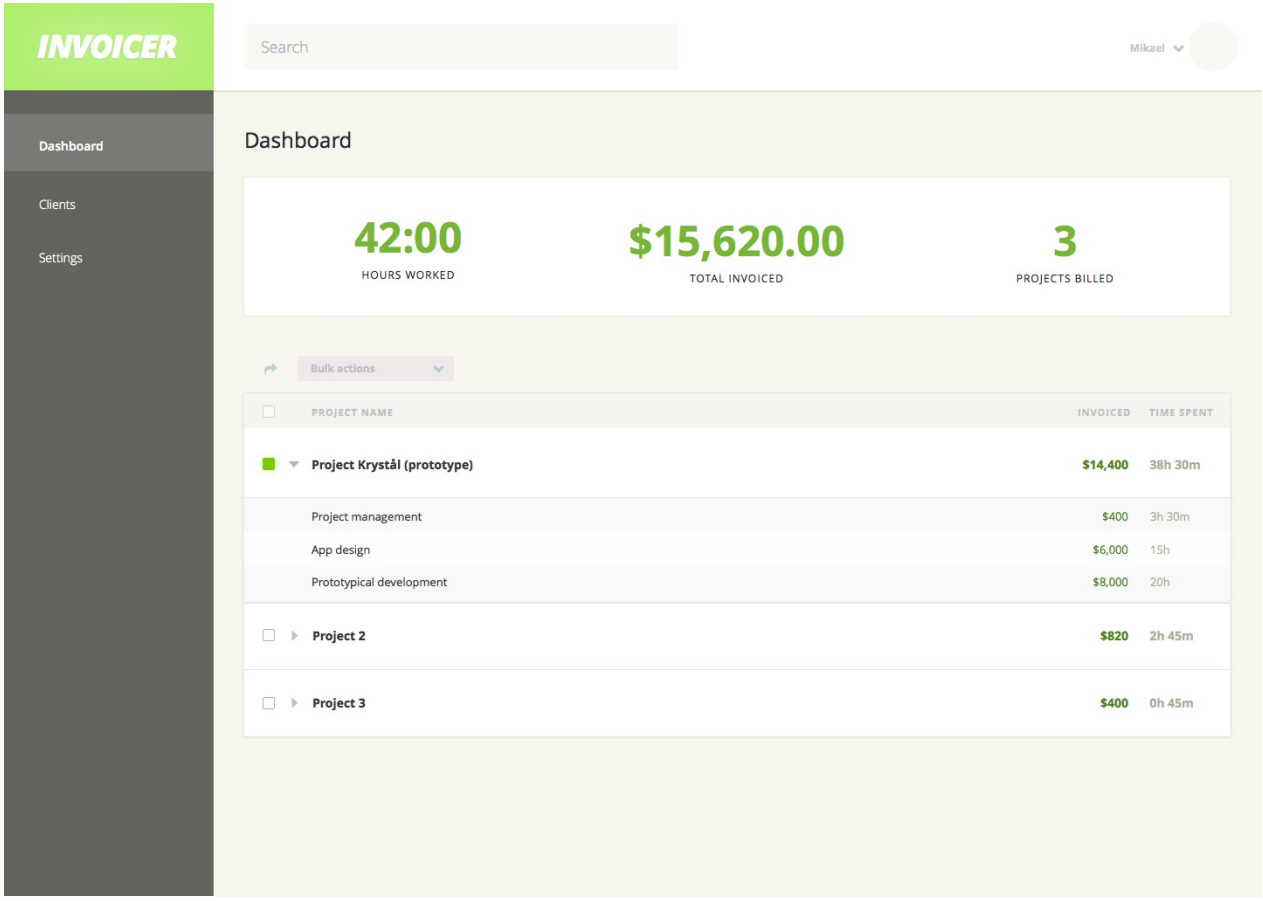
Ain't no better way of learning than by doing. So let's build an app!

Each chapter will tackle a separate stage of building the app. We'll start with scaffolding, move on to data fetching, then reducers, all the way to shared components.

Each chapter's also going to summarize the key rules *before* it gets started. It's important to know what we're about to do, and it's even more important to understand *why* we're doing it.

## The app

We'll build the dashboard for a basic invoicing app:




We're only going to tackle this dashboard — no other screens. There's enough complexity with it to show all of the rules within our app.

Let's get started!

# ES7, Harmony and Future JS

Before development let's jump start into some upcoming features of JS which increase expressiveness.

An important part of good code is readability. It's important to have concise code with good comments so you can understand it immediately... especially when you come back to code after a few months.

## Making UI code readable: future JavaScript

ES7 — also known as strawman proposals — combine ES6 with more features that improve the expressiveness of JS. Write your source as ES6/7 and use the Babel transpiler to convert your code to JS that browsers currently support.

## Learning ES6/7

ES6/7 isn't a new language... the features are easy enough to learn. Babel, the transpiler, has a great resource for jumping into ES6: https://babeljs.io/docs/learn-es2015/. They also have an online REPL — you can paste in ES6/7 to see transpiled ES5 immediately. It's great for understanding how the transformation works.

## A few common ES6 features within UI code

### Destructuring

Destructuring allows us to yank values from an array or object, which grants a readability bonus. Here's how it works:

```
let [a, b] = [1, 2];
// a === 1
// b === 2

let { ui, name } = this.props;
// ui === this.props.ui;
// name === this.props.name

let { blogpost: { title, slug } } = this.props;
// title === this.props.blogpost.title;
// slug === this.props.blogpost.slug;
```

### Function Binding ( `::` )

Function binding (simply `::`) is a time-saver and readability bonus for binding scope to a callback within your components.

```
class Link extends Component {

  click() {
    console.log(this.props);
    console.log('Clicked with scope', this);
  }

  render() {
    return <a href='#' onClick={ ::this.click }>Click me!</a>
  }

}
```

Adding the function bind syntax to `::this.click` converts down to `this.click.bind(this)`, ensuring that your callbacks in a component are called with `this` scoped to the component.

The documentation and proposal can be read at https://github.com/zenparsing/es-function-bind.

#### Spread operator ( `...` )

The spread operator allows us to expand an object or array without iteration:

```
import * as postActions from 'actions/posts';
import * as userActions from 'actions/users';

let actions = (dispatch) => {
  return bindActionCreators({ ...postActions, ...userActions}, dispatch);
}
```

In this example we create a new object containing all values from postActions and userActions in one line.

# Scaffolding a new project

The first thing you're going to need to do is scaffold the project. "What's scaffolding?", I hear you ask. Good question. It's essentially the same as using a boilerplate to get your project off the grounds.

We need a blank project which has our tooling and libraries installed. A list of the things we'll use is below, with a tiny description about what they do. We'll talk about the important ones below the list:

**Builders & JS**:

- webpack: module bundler
- babel: transpiles ES7 to ES2015
- eslint: keeps our ES standards in check
- esdoc: generates JS documentation and ensures we comment properly

**CSS**

- postcss: css preprocess manager with many awesome plugins
- postcss-nested: to write nested css as within sass
- postcss-browser-reporter: for reporting errors directly within your browser
- cssnext: css transpiling using future css syntax
- stylelint: for keeping our css standards in check
- cssnano: for minified, optimized production builds

**Libraries**

- react: ui and rendering library
- redux: state management
- immutable: keeping state immutable
- react-router: routing
- react-router-redux: storing state in redux
- react-css-modules: for applying css via css-modules to classes
- redux-form: to manage form state
- redux-ui: to manage component ui state

# Setup

Let's walk through some basic setup so you understand how to put this together.

A heads up, though: the entire setup is available for you to clone via `git clone git@github.com:tonyhb/redux-boilerplate.git`. You can see the project here: https://github.com/tonyhb/redux-boilerplate. Cheat all you want by cloning this — I think laziness and speed are great when programming.

## Step 1: Webpack

## What is webpack?

Webpack is a *module bundler* that glues together all our other tools. It does the following:

- Combines separate JS files into one single bundle (with all dependencies loaded)
- Uses plugins (such as babel) to preprocess files
- Uses plugins (such as uglify) to postprocess files for minifications
- Adds sourcemaps for debugging during development
- Configures environment variables within your JS app

It's a great tool and, as you might expect, has a lot of features and configuration options. Pete Hunt put together a great webpack guide here: https://github.com/petehunt/webpack-howto. It covers a lot of webpack's advanced features which you should use in your app, though it'd be futile to copy/paste here. Definitely check that howto out if you want to learn more.

## Setting up webpack

Webpack's configration is handled by a `webpack.config.js` file. We're going to make two: one for development and one for production. They're going to share common configuration which will be extended with changes between prod and dev (such as minification).

[TODO]

# Data loading: building an SDK

The first thing we're going to need to do is build a **JS SDK** for accessing our API.

Our JS SDK is a separate project. This project interfaces with our API on our projects behalf. **It's the sole place which makes AJAX calls to our API** - kept away from our UI project.

Why?

This ensures that:

1. Our actions are decoupled from our API. They call the SDK – not endpoints directly.
2. The SDK can be tested alongside our API; if the API responses change shape unexpectedly our SDK can fail tests. This means that we have zero surprise breaking changes across teams. And we can test this separately from our UI tests.
3. The SDK can implement data mocking transparently to our UI code
4. Mocking via our SDK means we can use hot reloading and ignore proxying requests to our API
5. SDK mocking and tests are self-documenting and can be reused across projects

All said, creating a separate SDK makes things easier to code (via hot module replacement), easier to test and easier to share. Let's start by building one for our API.

# Building the SDK

## Best practices

We have three best practices for building an API:

**1. Use superagent and callbacks to manage async logic**

This is a nicer way of saying "don't use promises". Promises consume errors which makes your code much harder to work with. Also, we're going to write some redux middleware to automatically track the status of our AJAX calls from the SDK. Working with callbacks here is *much* easier than workign with promises.

**2. Each API call should be one SDK call**

We'll keep control flow, such as linking two API calls, within our actions. If two calls need to be tightly coupled within the SDK you've probably got a faulty API design.

**3. Normalization should happen within your UI project in Normalizr**

The API should be response-agnostic. We'll handle normalization of the API response within our UI code.

## Writing the SDK code

# Components

We classify components into four categories when building our apps:

1. Higher Order Components (decorators): things such as Redux' `@connect` or redux-ui's `@ui`
2. Reusable components: things such as SeatGeek's infinite scroll or an abstract pagination component
3. Functional components: small, undecorated dumb components written as functions (see here for more info)
4. Standard components: any other component that doesn't fit the above. Typically custom for your app

## Higher Order Components (HOCs)

Higher order components are decorators that add powerful functionality to your standard components. These do things like connect to Redux' state store to automatically pass down props ( `@connect` ), set up UI state for a component and pass down props ( `@ui` ), or create a form with any validation ( `@reduxForm` ).

A general rule of thumb is that a component should be higher order if it:

- Doesn't render directly to the DOM
- Passes down props to a standard component you're defining
- Hooks into react lifecycles to manage, manipulate, or improve a standard component

It's rare to need to build a higher order component; typically most components you build will directly render nodes to the DOM.

Building a higher order component that's abstract enough to reuse takes planning. It's often easier to write a HOC after you see recurring verbose patterns within your components.

Once you see patterns emerge you can plan to build a HOC that works for all use cases across your app.

## Reusable components

Reusable components are different from higher-order components in that these are standard components that you add as JSX to render. Things such as an infinite list, pagination or tag list comprise a set of reusable UI components for use across your entire app.

There are a good suite of libraries for reusable components such as material-ui to speed up development.

The core of building good reusable components is thinking about the minimum amount of data that needs to be passed in for the component to work. When defining a tag list, for example, we could specify the propType API as:

```
class TagList extends Component {
  static propTypes = {
    tags: PropTypes.oneOfType([PropTypes.array, PropTypes.object).isRequired,
    max: PropTypes.number
  }

  static defaultProps = {
    max: 10
  }
}
```

Reusable components should always be saved in the `components` directory of your source.

## Functional components

Functional components are lightweight functions that return a JSX tree. They're suited for small components such as single elements or groups of small DOM nodes.

An example of a shared Button functional component:

```
import React from 'react';
import css from 'react-css-modules';
import styles from './button.css';

/**
 * Button represents a default button which accepts three props: text,
 * onClick and variant.
 *
 * Note that this Button needs styles applied through react-css-modules;
 * to use default styles declared here import the 'Button' component.
 */
const UnstyledButton = ({ text, onClick, variant }) => (
    <Button
      styleName={ styles[variant] }
      onClick={ onClick }>{ text }</Button>
);
// Note: 'styleName' is react-css-module's version of the 'className' prop

// Create a Button with the default styles applied
const Button = css(Button, styles);

export default Button;
export UnstyledButton;
```

You'll see a few awesome things about functional components:

1. They're super concise, which means they're easy to read and write
2. They define all of their props using destructuring as opposed to PropTypes. This is awesome as they're self-documenting... all props are included in the function signature along with defaults

Because functional components are so small they're also great to use for building shared components. We'll talk about this more in the 'sharing components' chapter, though the above is a great example of a shared component.

## Standard components

Standard components represent the core of your app. These are typically single-purpose components designed and built for a particular route in your app.

Standard components have no internal state — they receive state as props through a combination of Redux' `@connect` and reselect.

They are typically stored within a folder representing the current route, for example `scenes/pages/index.js`.

# Sharing Components

TODO: Expand this.

## Rules for sharing components:

1. Strive **not** to share components which integrate framework-specific HOCs.
2. Instead, make all components dumb and add a wrapper which passes props from your framework
3. If possible use small functional components
4. Bundle CSS with the component using react-css-modules; export the 'naked' and 'styled' component together.
5. Bundle selectors with the component by default. These should be overridable within the 'naked' component

## Process for sharing components across teams/products

1. Build components using above rules
2. Don't worry about making things abstract enough to share immediately
3. When a team/product needs to integrate a component look at use cases and figure out data needed for component in each scenario.
4. Generally, component should be able to be used in-place while selectors may need to be re-written

# Writing middleware

TODO

# Tests

There are a few testing frameworks out there for JS — mocha, jasmine, jest (which is React specific).

Each of them has their pros and cons, and at Docker we've decided to use a combination of Mocha (for tests) and Chai (for assertions). We also use JSDom to create a virtual DOM environment which we use to render React components.

Why?

- Mocha and Chai are easy to use, fast, and compatible with ES7 via Babel
- Mocha doesn't automatically mock everything in the way Jest does. That's a ton of magic you normally end up undoing. Basically, don't use Jest.
- JSDom is lighter weight than spinning up a browser or PhantomJS instance
  - This means faster tests
  - This means faster continuous integration cycles
- JSDom means we can virtually render without shallow rendering.
  - This is *super* important because shallow rendering means you can't test decorators. Shallow rendering only renders one level deep. Decorators typically wrap your custom component with a parent — so *only the parent decorator will render*.

## The setup

Setting up Mocha, Chai and JSDom to work with a Redux project is relatively easy. There are three steps:

1. Install everything via NPM
2. Change the package.json test command to use Mocha
3. Add a JSDom setup

#### 1. Installing packages

We need a few packages to get our tests running:

```
npm install --save-dev mocha chai jsdom
```

#### 2. Changing the test command of package.json

Our `package.json` commands should look like this;

```
  "scripts": {
    "test": "$(npm bin)/mocha  --compilers js:babel/register --recursive --require ./test
    ...
  }
```

Whoa. Complex AF, right? Here's what it does:

- Sets up Mocha to use BabelJS' "register" hook. Essentially every included file is put through Babel's preprocessor, which means you can write your tests in ES7 **and use ES7 includes**.
- Requires a test setup file which fakes a DOM using JSDom. This file is included below.
- Uses a splat to search for every folder which contains a `_tests` directory. All JS files in these folders will be run as tests.

#### 3. The JSDom setup

The `test.setup.js` file contains this:

```
import { jsdom } from 'jsdom';

global.document = jsdom('<!doctype html><html><body></body></html>');
global.window = document.defaultView;
global.navigator = global.window.navigator;
```

Kinda simple - set up a few global variables which use JSDom to create a fake document. Boom, you're good to render in pure JS with React.

## Writing tests

If you're not so familiar with writing tests first read the Mocha documentation. Writing tests for a Redux app isn't that different from what you'd find in lodash.

Some pointers for things you should test:

- **Test your middleware**. This is basically everything that DRYs up your app.
- Test your reducers to ensure they're modifying state as expected.
- Test your selectors to ensure they're working as expected. You don't want to have to render a component to validate selectors.
- Test to ensure that your components render correctly given variants of data (for example, if something is hidden/shown depending on props). This is for bonus marks if you want an A+ from your teacher.
- Test your utility files and functions that are shared.

## Bonus: optimizing CI tests with Docker

When you run tests via CI you'll need to `npm install` every time you run tests. This can be *hella* slow. Instead, we can build a docker image which uses a hash of package.json to cache build images. We only end up waiting for a `docker pull` instead of an `npm install`, which is much faster.

# TODO: Explain how to create a docker image that caches NPM installs then uses this image for tests

# Styles and Standards

### Actions

We stick to the Flux Standard Action spec. This states that an action must take the following shape:

```
{
    type: 'CONST',
    meta: {
        // Everything in 'meta' will *only* handled by middleware
    },
    payload: {
        // Payload contains all data for use within a reducer
    },
    error: '' // An optional error parameter
}
```

By sticking to this format we separate middleware logic from reducer payloads.

### Selectors

Name selectors with a `get` prefix. For example, if your selector gets an array of blog post titles, it might be named `getPostTitles`.

Why? If we name it `postTitles` we'll have an import like `import { postTitles } from './selectors';` at the top of our code, and nothing else will be able to be called postTitles within that entire file. Essentially we'll have to rename our props from `postTitles` to something else.

Giving a prefix to all selectors means:

1. We don't pollute component namespaces with the above issue
2. It's easy to tell when something's a selector via the `get` prefix
3. It's easy to search and replace these names within your project

# Examples

TODO:

- SDK callback middleware
- Example components sharing selectors
- Example HOC adapters
- Example functional components with default CSS