



React & Redux advanced

Let's mix some HTML and JS together

Agenda

Project update - design system and usage

Design systems

Component lifecycle

React - additional hooks & optimization

Redux advanced

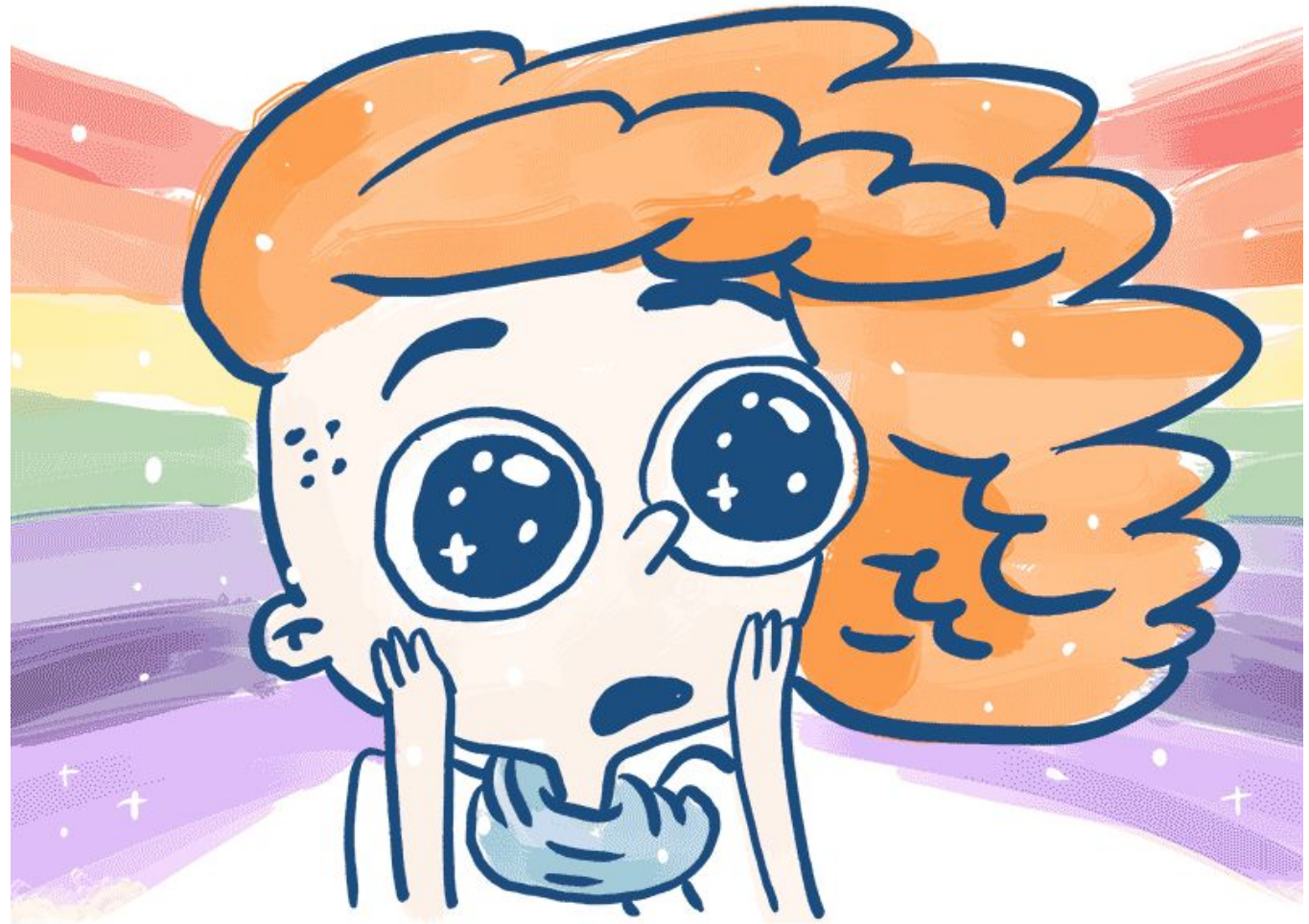
Landing and dashboard

Project update - design system and usage

What are Design systems?

The **single source of truth** which groups all the elements that will allow the teams to design, realize and develop a product.

It's not a deliverable, but a set of deliverables.



“A kit of UI components without accompanying philosophy, principles, guidelines, processes, and documentation is like dumping a bunch of IKEA components on the floor and saying
“Here, build a dresser!”

The guidelines and documentation accompanying the components serve as the instruction manual that come with the IKEA components to help the user properly and successfully build furniture.”

Brad Frost

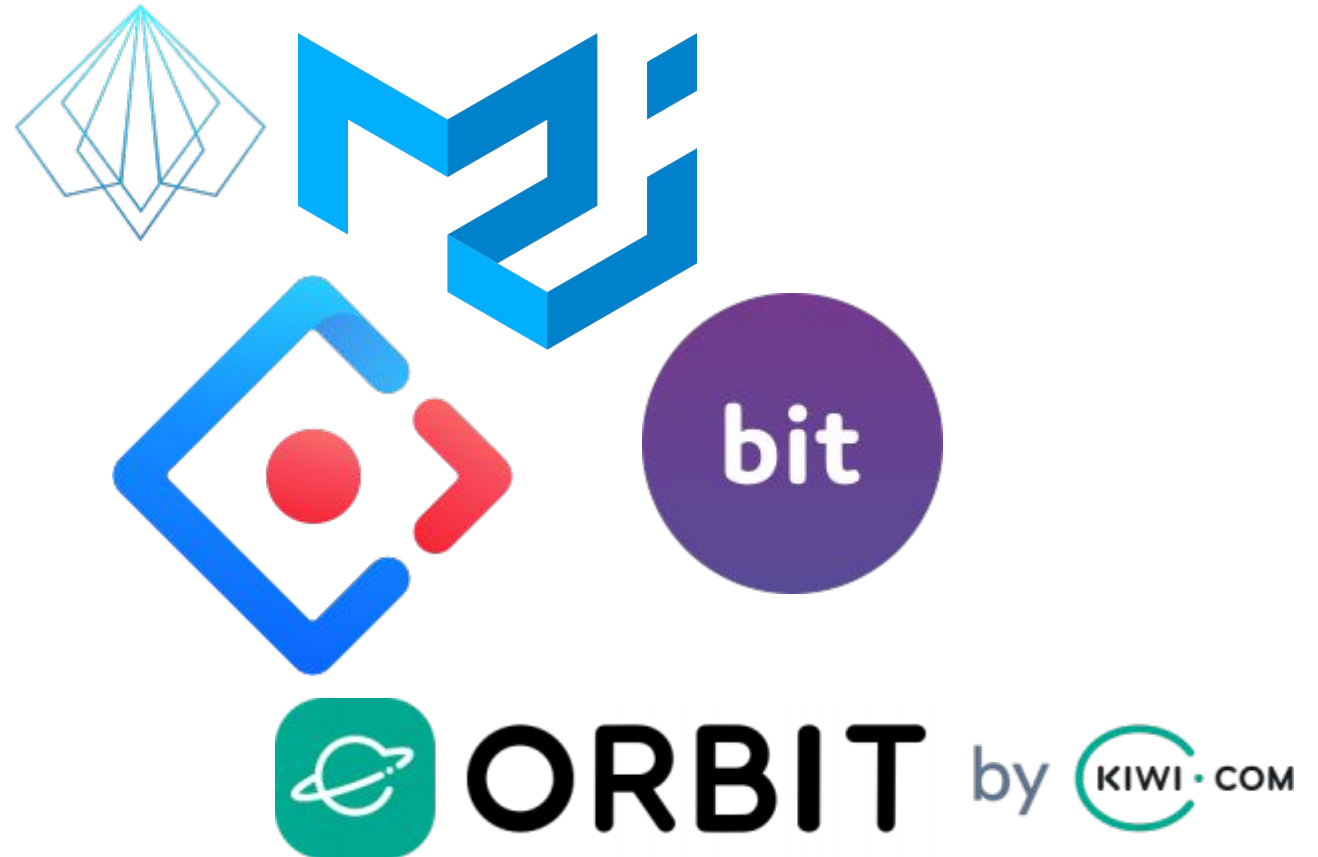
Why working with Design System?

- Reduce inconsistency
- Focus on the user
- Faster prototyping
- Quick iteration



Multiple free and open source design systems

- [Material UI](#) (Google)
- [PatternFly](#) (Red Hat)
- [Orbit](#) (Kiwi)
- [Ant design](#)
- [Bit](#)

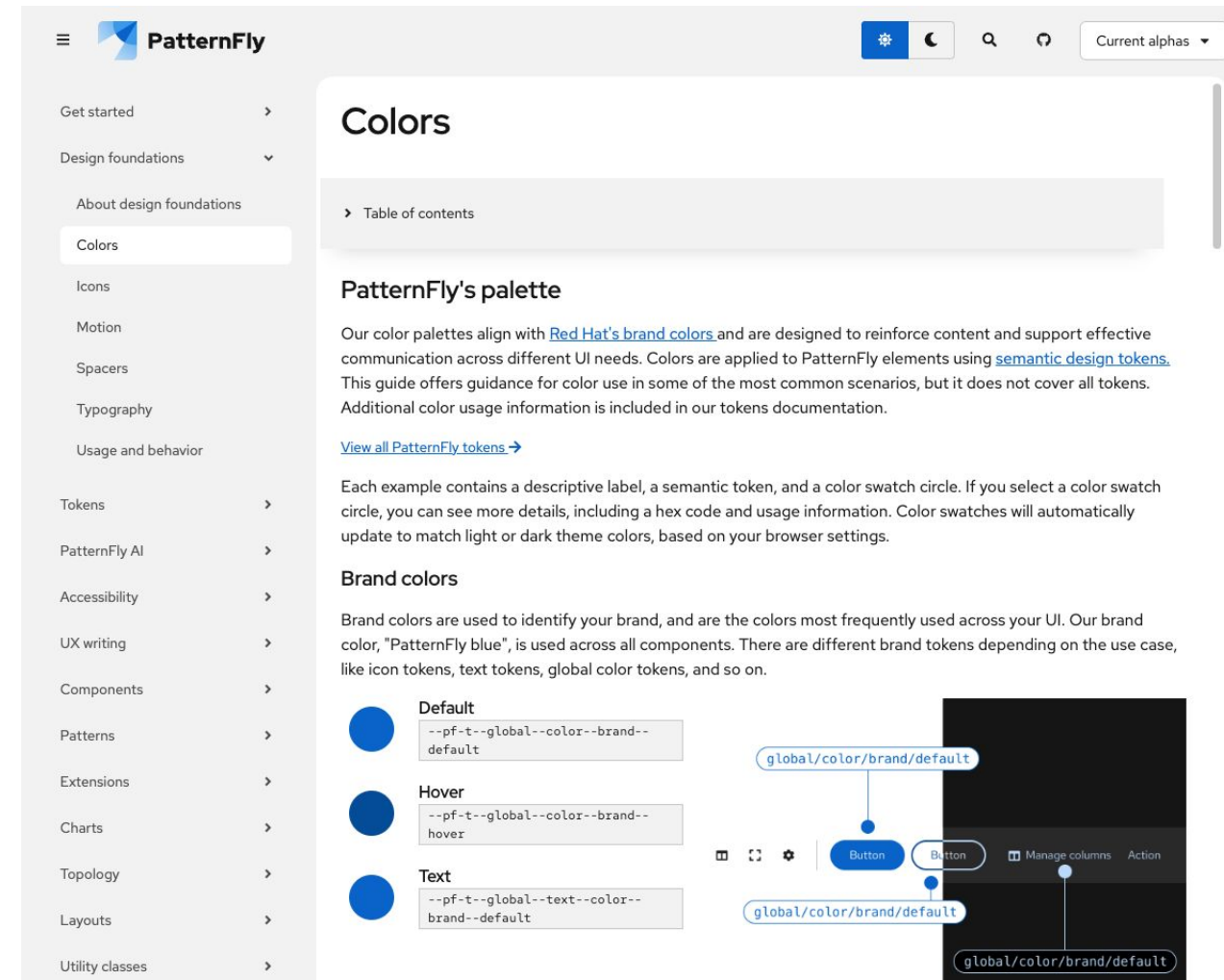


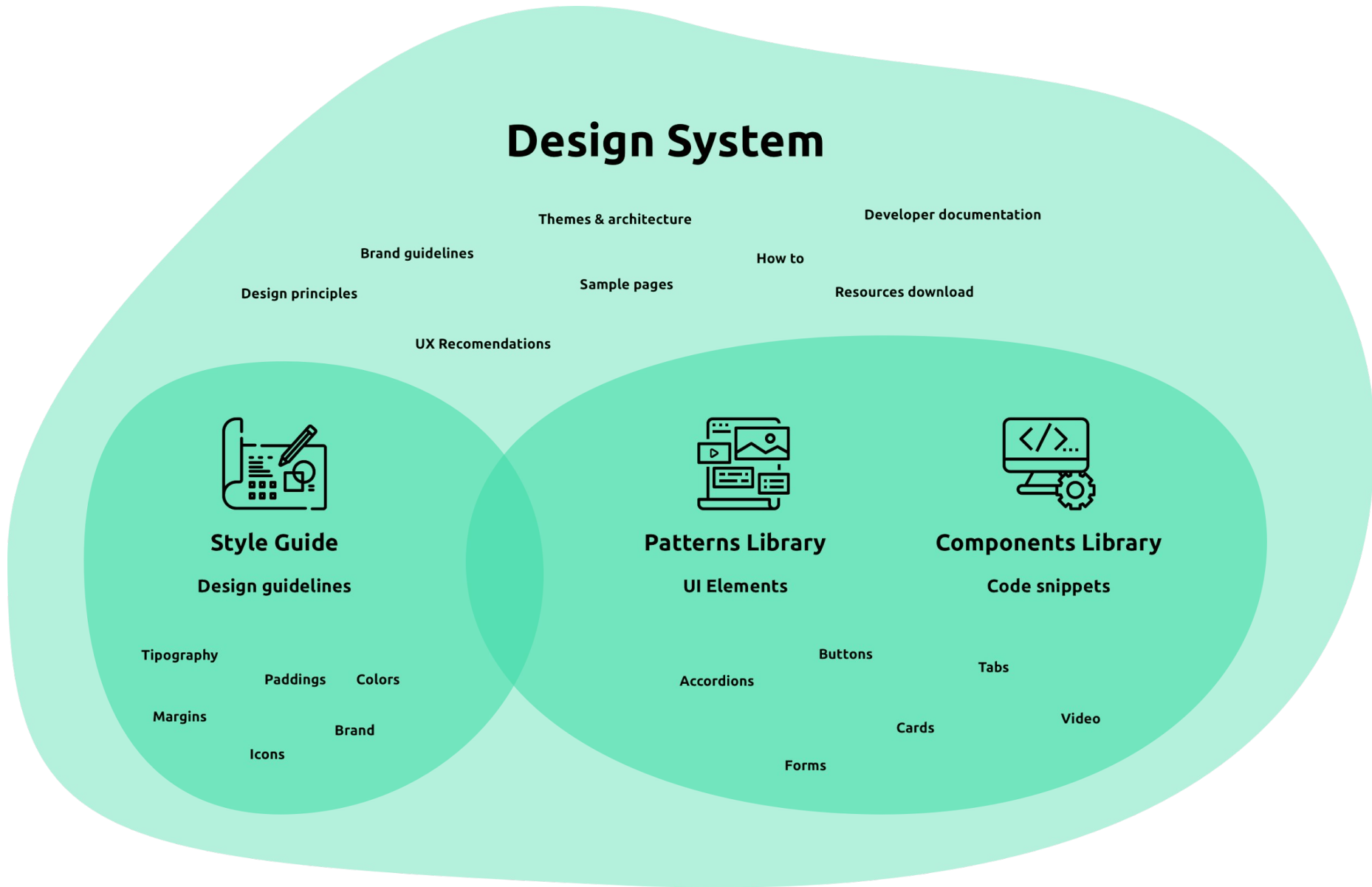
Style guide x Pattern library

Style guide – focuses on graphic styles (colors, spacers, icons, content...) and their usage.

Pattern library – integrates functional components and their usage.

Design system usually contains both.





MATERIAL DESIGN Components > Buttons > Anatomy

Anatomy

Buttons contain one required element and four optional elements.

1
A
+ BUTTON
C

2
A
+ BUTTON B
C

3
A
+ BUTTON B
C

4
A
B
C

Carbon Design System

- Get started
- Tutorial
- Guidelines
- Components
 - Overview
 - Component status
 - Accordion
 - Breadcrumb
 - Button
 - Checkbox
 - Code snippet
 - Content switcher
 - Data table
 - Date picker
 - Dropdown
 - File uploader
 - Form
 - Inline loading
 - Link
 - List
 - Loading
 - Modal
 - Notification

Link
Text link

List

Loading

Multiselect

Notification

Number input

PatternFly

Clipboard copy
Code block
Code editor
Content
Data list
Date and time
Description list
Divider
Drag and drop
Drawer
Dual list selector
Empty state
Expandable section
File upload
Forms
Form
Form control
Form select
Checkbox
Radio
Text area
Text input
Helper text
Hint
Chip (Deprecated)
Icon

Form

A **form** is a group of related elements that allow users to provide data and configure options in a variety of contexts, such as within modals, wizards, and pages.

React HTML HTML demos Design guidelines

Table of contents

Examples

When using helper text inside a `FormGroup`, the `HelperText` component should be wrapped with the `FormHelperText` component to provide additional spacing.

Basic

Full name *

Include your middle name if you have one.

Email *

Phone number *

How can we contact you?
 Email Phone Mail

Time zone
 Eastern Central Pacific

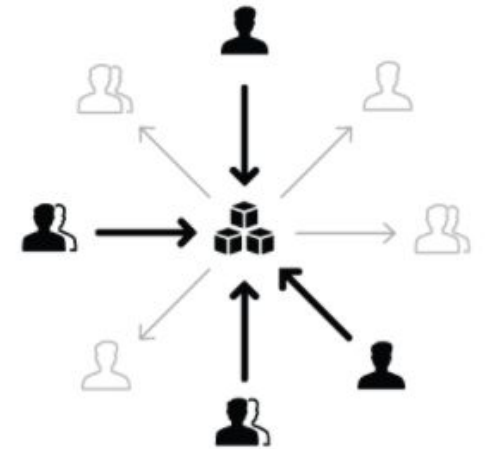
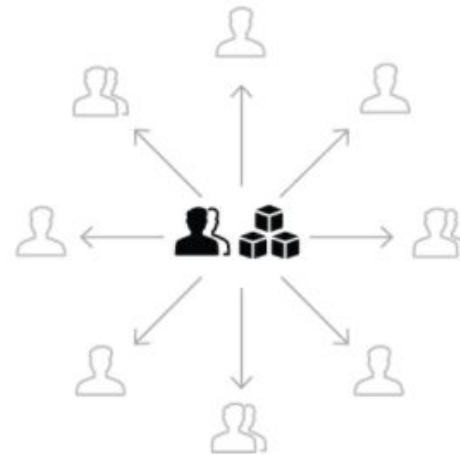
Additional note

I'd like updates via email.

Centralized vs. Distributed system

Centralized - one team is in charge of the system and makes it evolve.

Distributed - several people of several teams are in charge of the system. The adoption of the system is quicker because everyone feels involved.



How do we use PatternFly?

Components as elements

Dropdown ▾

Details | YAML | Environment | Events | Terminal

Ned Username ▾

Label

Text input !

First step ✓
This is the first thing to happen

Second step 🎯
This is the second thing to happen



Third step ○
This is the last thing to happen

🔔 10

Button

Group label 🔔 Label 1 🔔 Label 2 🔔 Label 3 2 more ✕

Type styles

		
Size 4xl	Size 4xl	Size 4xl
Size 3xl	Size 3xl	Size 3xl
Size 2xl	Size 2xl	Size 2xl
Size xl	Size xl	Size xl
Size lg	Size lg	Size lg
Size md	Size md	Size md
Size sm	Size sm	Size sm
Size xs	Size xs	Size xs
Size xs bold	Size xs bold	Size xs bold
Heading label	Chart sublabel (standard)	
Chart label	Chart sublabel (small)	

Color palette



Show library and template

The screenshot displays a web application interface with a dark header and a sidebar. The header includes the 'PATTERNFLY' logo, a user profile 'Ned Username', and notification and settings icons. The sidebar lists navigation items: 'System panel', 'Policy', 'Network services', and 'Server'. The main content area features a breadcrumb trail with five items: 'tertiary nav item 1' through 'tertiary nav item 5'. Below the breadcrumb is a section titled 'Stacked form demo' with the text 'Below is an example of a stacked form.' The form itself is a vertical stack of components: three text input fields, each with a 'Label' and a 'Text box' placeholder; a row of three checkboxes labeled 'Email', 'Phone', and 'Please don't contact me'; a single checkbox labeled 'I like updates via email'; and a final row with a blue 'Submit form' button and a 'Cancel' button.

Template preview

Handoff to developers

Blog posts

[Hello world! This is my first blog post](#)

2 minute read | Beginner

This is a short description of my post. It really doesn't contain much though. This is a short description of my post. It really doesn't contain much though.

[Painting a wooden dresser: my journey](#)

Painting dressers, or any furniture for that matter, can be a strggule. In this article, I explain the details of my process and what went well and what did not. You'll leave this read with some good tips!

[Why are millenials getting a bad rep?](#)

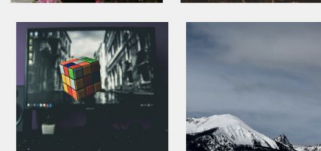
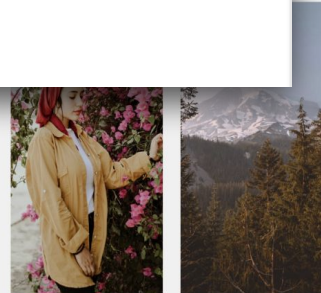
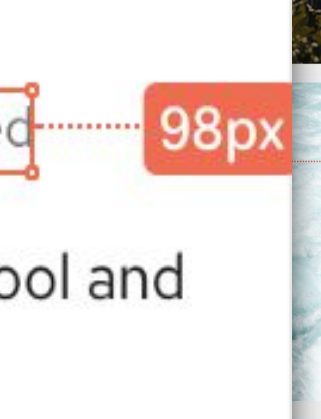
3 minute read | Beginner

This is a short description of my post. But to answer the question I'm not really sure. Are you?

[Tech savvy moments!](#)

5 minute read | Beginner

A list of all the tech savvy moments I've experienced within the past decade. Some are funny. Some are impressive. Most are not cool.



Twitter feed

- Bonginkosi Meladlana tweeted
This is the tweet. It's pretty awesome. 430px
- Deveeprasad Acharya retweeted
This is the tweet. It's pretty cool and awesome.
- Sidnee Gye tweeted
This is the tweet. It's pretty awesome.
- Jacqueline Likoki retweeted 98px
This is the tweet. It's pretty cool and awesome.
- Wim Willems tweeted
This is the tweet. It's pretty awesome.
- Zhan Huo retweeted
This is the tweet. It's pretty cool and awesome.
- Bonginkosi Meladlana tweeted
This is the tweet. It's pretty awesome. 607px
- Deveeprasad Acharya retweeted
This is the tweet. It's pretty cool and awesome.
- Bonginkosi Meladlana tweeted
This is the tweet. It's pretty awesome.

← Jacqueline Likoki re A

PROPERTIES

Width	Height
176px	21px
X Position	Y Position
1226px	430px

APPEARANCE

Colour	#6A6E73
--------	---------

TYPOGRAPHY

Typeface
RedHatText-Regular

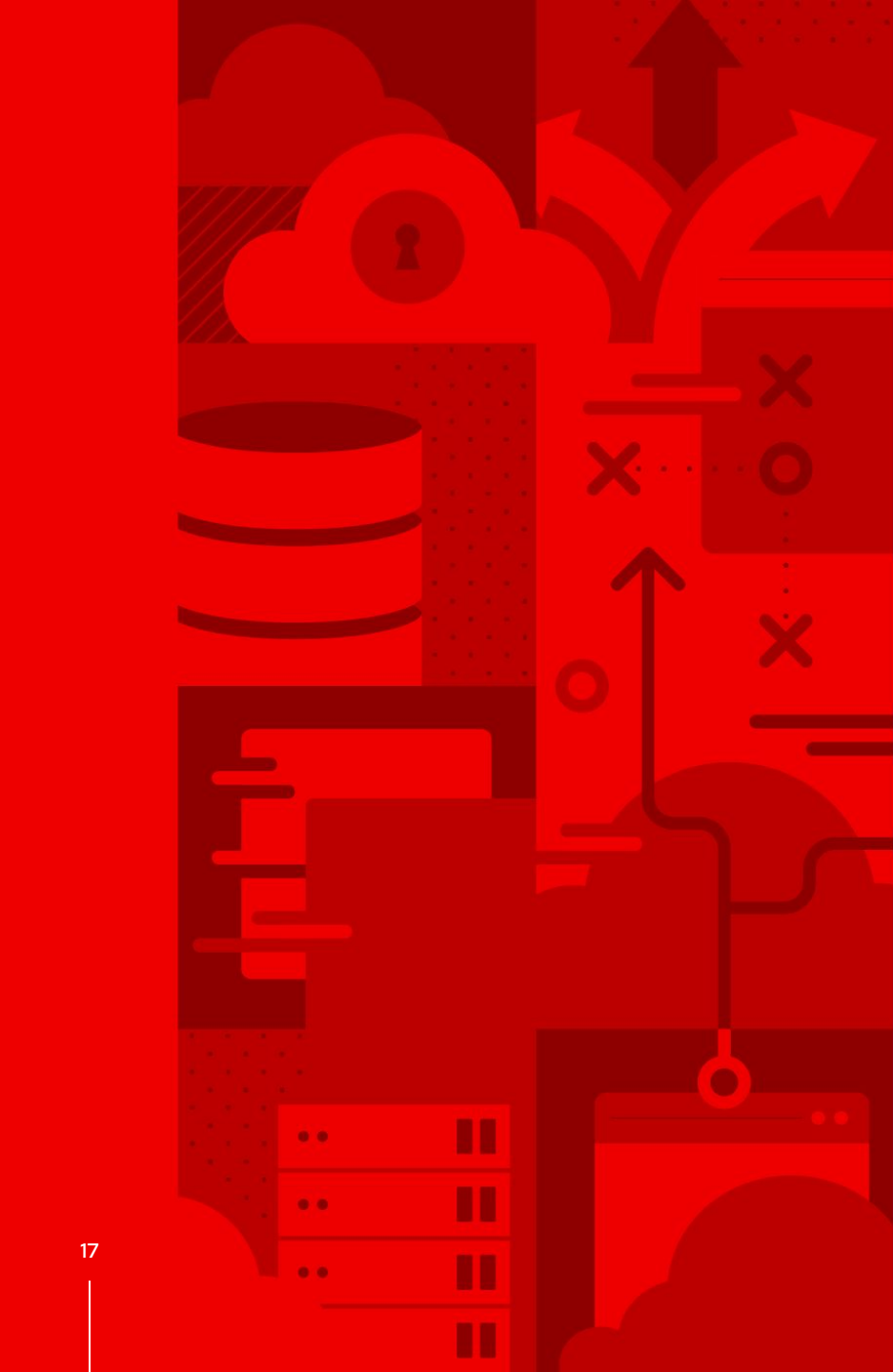
Size	Align
14px	Left
Font Weight	Line Spacing
500	21px

CONTENT Copy

Jacqueline Likoki retweeted

CSS Copy

```
.jacqueline-likoki-re {
  color: #6A6E73;
  font-family: RedHatText;
  font-size: 14px;
  font-weight: 500;
  line-height: 21px;
  text-align: left;
}
```

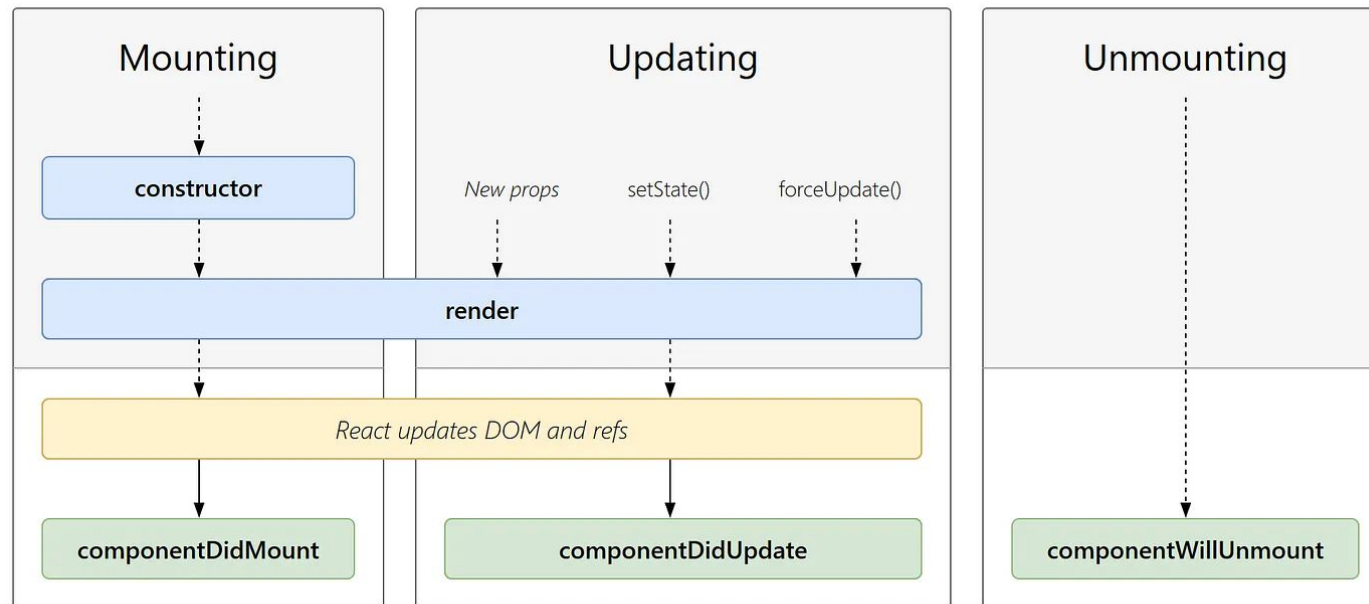



React advanced

Component lifecycle & optimization

Component lifecycle

- After React dev team introduced hooks, the life cycles of react component has changed significantly - preparation for concurrent mode and async rendering
- The management is different for React **Class** and **Functional** components



Component lifecycle - Classes

There are several predefined class functions with a static trigger sequence

- **constructor**
 - Is triggered when the class is instantiated
 - Useful for setting initial component state, registering listeners...
 - In most cases is not really necessary
- **componentWillMount** (deprecated)
 - After constructor, but before first render
 - Was used for initial data fetch
 - Misuse will cause faulty state changes in async mode
- **componentDidMount**
 - Is called **once** only after **initial render**
 - New place for initial data fetch

Component lifecycle - Classes

- **componentWillReceiveProps** (deprecated)
 - Component is about to receive new props, but it did not re-render
- **componentDidUpdate**
 - Props or component internal state were updated
 - Based you can compare new and previous props/state to make some updates, api calls, logs, etc.
 - Danger of infinite loops
 - State changes must be **wrapped in condition**
- **componentWillUnmount**
 - Clean up phase before the component is removed from virtualDOM

Component lifecycle - Classes

- **shouldComponentUpdate**
 - Rendering is the most expensive operation in DOM an any library/framework
 - Developer can use this method to programmatically check whether to trigger render method or not
 - This will affect component children and not send new props to them
 - If children rely on parent component you cannot use this in most cases
- **Render** - main rendering method
 - **Must be implemented** in every class component and return something renderable
 - Render is triggered on prop and state changes by default
 - Can be handled via shouldComponentUpdate
- **componentDidCatch** - handler for unpredicted errors in virtual DOM
 - The “Whoops! Something has happened.” screen

Component lifecycle - functions

You can handle functional component lifecycle via **useEffect** hook

- Possible in React version 16.8.x and later
- Allows performing **side effects** in your functions
 - Side effect triggers something outside of a function scope
 - Breaks the pure function rule - same input may give you different output
 - Necessary for efficiently reacting on (user) events
- Can replace all lifecycle methods, except `componentDidCatch`

Component lifecycle - functions

useEffect arguments:

- Effect function
- List of triggers (dependencies)

```
const Component = ({ username }) => {  
  useEffect(() => {  
    API.getUserDataAndUpdateAppGlobalState(`api/...`)  
    return () => {  
      cleanAppGlobalState()  
    }  
  }, [username])  
  return (  
    <h1>{username}</h1>  
  )  
}
```

There can be multiple effects, reacting on different triggers.

That way, we can mimic the life cycles of classes.

Component lifecycle - functions

- No list of triggers means that the effect will trigger on **every** props/state change
 - Will cause infinite loop if state is changed here
 - Same use cases as **componentDidUpdate**
 - Will also trigger like **componentDidMount**
- Empty dependencies list means that it will trigger only once, after initial mount
 - **componentDidMount**
 - Does not react on any props/state updates

```
useEffect(() => {  
  /**effect */  
})
```


Component lifecycle - functions

- Effect will be called when any variable in the list is changes
 - **componentDidUpdate**, **componentDidMount**
 - Non primitive types must follow immutable pattern to trigger effect -> must return new instance
 - Also will be triggered in first render

```
useEffect(() => {  
  /**effect */  
}, [propName, stateName])
```

Component lifecycle - functions

- If effect returns a function, it will be called before component is unmounted from DOM
- **componentWillUnmount**

```
useEffect(() => {  
  /**effect */  
  return () => {  
    /**clean up */  
  }  
  
}, [propName, stateName])
```

Render cycle - when rendering happens

- Render function is triggered when
 - Component props has changed
 - Component state has changed
 - Component context has changed
 - Parent has re-rendered
- Everything can be optimized/block to save rendering cycles
- Render will not trigger if props/state are

MUTATED VARIABLES OF THE SAME INSTANCE

Different data binding (one vs two way)

- One way data binding
 - Used by all modern UI libraries/frameworks
 - Data can be send only in one direction in the DOM
 - Data is “bubbling” down through nodes to the leaves of the DOM tree
 - Predictable behaviour
 - Forces component independence
 - **From parents to children (React)**
 - From children to parents (not a good idea)
Although technically parents can access its children data it's not a good idea
 - **DO NOT TOUCH CHILDREN!**
 - It opens pandora's box of bugs

Two way data binding

- Used in older libraries/frameworks
- One of the reasons why original Angular was abandoned
- Developers ignored good practices and were accessing parent data from children
- Components lost their independence

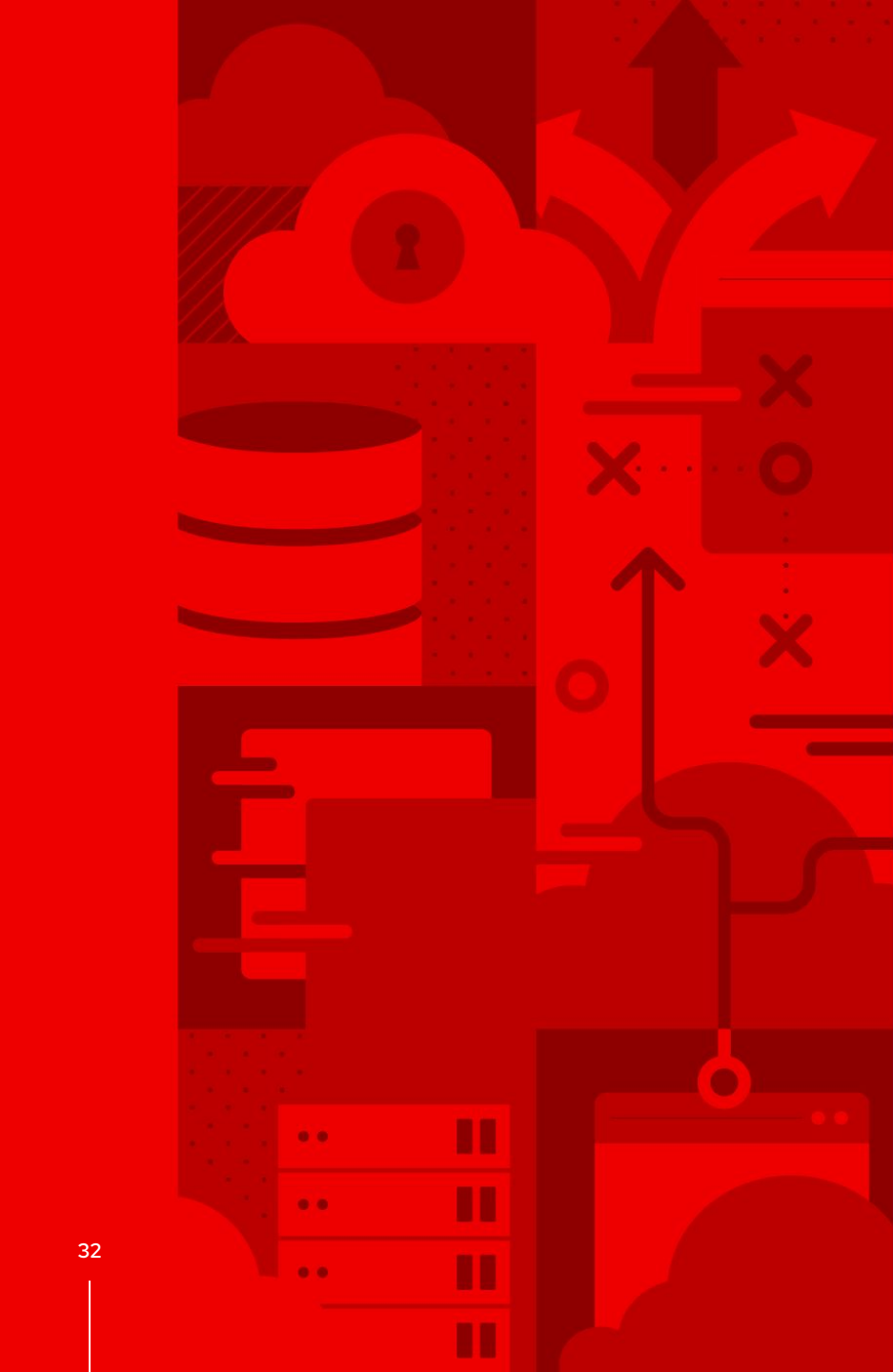
```
const InputComponent = () => {
  return (
    <input
      name="controlled-input"
      value={parent.parent.data.value}
      placeholder="No value"
      onChange={parent.parent.parent.parent.handleInputChange}
    />
  )
}
```

Optimization hooks

- There are several hooks that can help you optimize your code
 - As always, before optimizing, check if you actually need it
- **useCallback**
 - Used to define a function that has referential equality between renders - changes when its dependencies change
- **useMemo**
 - Used to calculate a value that has referential equality between renders - changes the value when their dependencies change
- **useRef**
 - Special hook, that behaves similar to useState, but does not trigger re-render

Additional hooks

- **useContext**
 - React.provider hook to consume context
 - Easy to use multiple context providers in one component
- **useReducer**
 - State management hook, if you need to store big chunk of data in component
- **Custom hooks**
 - You can write your own hooks, and share them in your library
 - Great examples: useDispatch, useSelector from react-redux library
 - Any function can be "hook" as long as it uses any React's hooks



Redux

Thinking with redux - one state to rule them all

Agenda

What is redux - principles

Going from action type through action to state update

How to use connect - 3 parts of connect function

How to use hooks with Redux

Middlewares and other cool tricks

Redux Toolkit

You might not need Redux - useReducer

Let's do some coding...

What is Redux - definition

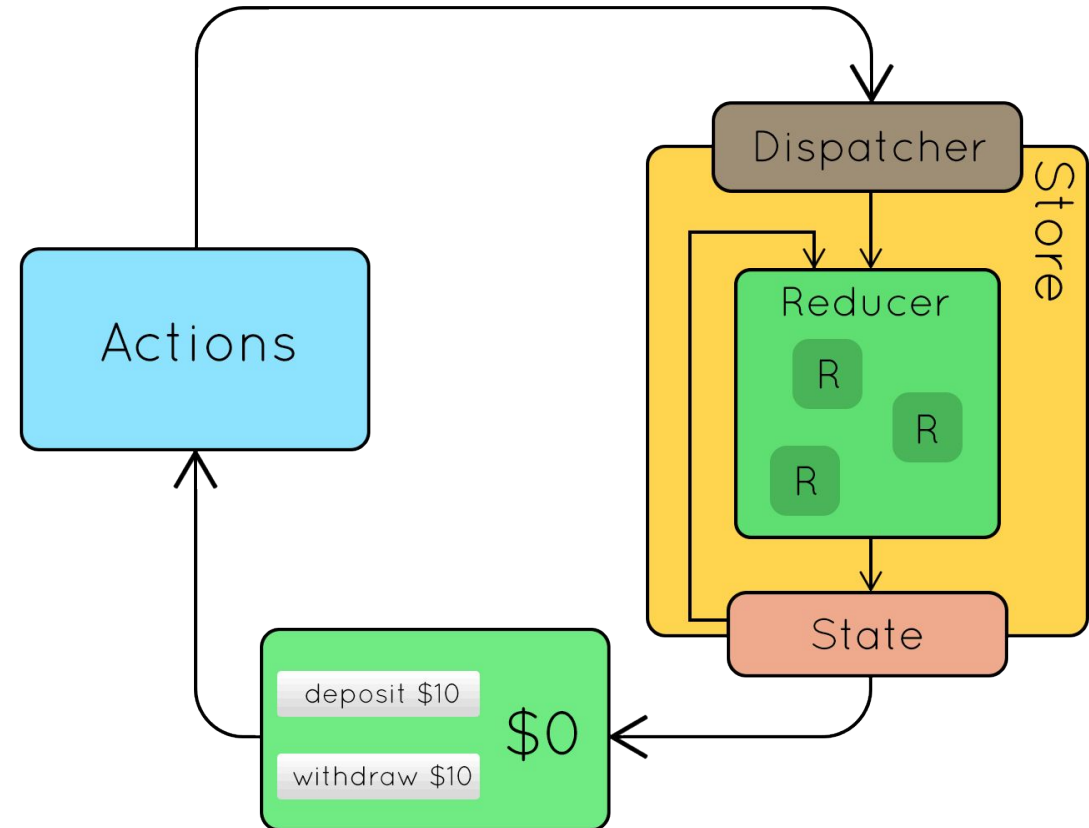
- A predictable state container for JavaScript applications.

Key concepts:

- **State** - a single object that represents the application's state
- **Store** - single source of truth (object) where the global state of the entire application is stored
 - provides methods: *getState()*, *dispatch(action)*, *subscribe()*
- **Action** - plain objects that describe what should happen
 - must have a **type** (identifies the action) and can carry **payload** to change the state
- **Reducer** - function taking the current state and an action and returning a new state
 - pure function => always returns the same output for the same inputs
- **Dispatch** - method used to send an action to the reducer which updates the state
- **Subscriber** - function that gets called every time the state changes
 - React Redux provides higher-level abstractions that take care of it (*useSelector*, *connect*)

Draw it, and it all makes sense...

1. Something triggers an action - pure function that returns object
2. Action is dispatched into store (carrying payload)
3. Reducers pass this action around and mutate state based on type and payload
4. You can track history of actions because they are pure functions



Going from action type

- Action type - string constant to identify action, kinda like name

```
export const SOME_ACTION = 'SOME_ACTION_TYPE';
```

Going from action type through action

- **Action** - combined type, payload, error, meta
 - **Type** - to identify action
 - **Payload (optional)** - actual data, usually object
 - **Meta (optional)** - additional data, usually to identify records
 - **Error (optional)** - boolean value to indicate error

```
const doSomeAction = (data, entityId) => ({
  type: SOME_ACTION,
  payload: {
    entityId,
    data
  },
  meta: { entityId },
  error: false
});
```

Going from action type through action to state update

- Reducer is function that takes state and action
- Reacts to action and mutates state in expected way

```
const reducer = (state, action) => {  
  return {  
    ...state,  
    entities: state.entities.map(item => ({  
      ...item,  
      ...item.id === action.entityId && action.data  
    })))  
  };  
};
```

Going from action type through action to state update and use them

- createStore - **the OLD way** - to use reducers in a store
 - Reducer function
 - Default state
 - Enhancers
- combineReducers - to namespace your state and split reducers
- applyMiddleware - enhancer function to use middlewares (logger, async functions, etc.)

```
createStore(  
  combineReducers({ appState }),  
  { appState: {} },  
  applyMiddleware(logger)  
);
```

How to use redux in react app

- Connect function - the OLD way
 - `mapStateToProps`
 - `mapDispatchToProps` (optional)
 - `mergeProps` (optional)
- **Hooks**
 - **`useDispatch`** - provides access to the dispatch function
 - **`useSelector`** - allows you to extract data from the state
 - takes a selector that receives the entire store state and returns the piece of state you need
 - **`useStore`** - direct access to the store (actions not tied to component's rendering)

How to use hooks with redux - useSelector

- Replaces mapStateToProps and mergeProps functions
- Allows to pluck pieces of state from store
- It's recommended to use multiple selectors in component to improve performance
 - Always try to requests only primitive values, not whole objects (not always possible)

```
const Component = ({ userId }) => {
  const name = useSelector(({ userReducer: { profile: { name } } }) => name);
  const email = useSelector(({ userReducer: { profile: { email } } }) => email);
  const orders = useSelector(({ orderReducer: { orders } }) => orders.find(order => order.userId === userId));

  return (
    <div>...</div>
  )
}
```

How to use hooks with redux - useDispatch

- Replaces mapDispatchToProps
- Returns a dispatch function from closest store
- Use this function to call redux actions

```
const Component = () => {
  const dispatch = useDispatch()
  const handleThemeToggle = themeVariant => dispatch({ type: 'TOGGLE_THEME', payload: themeVariant })

  return (
    <div>
      <button onClick={() => handleThemeToggle('blue')}>Change UI to blue theme</button>
      <button onClick={() => handleThemeToggle('orange')}>Change UI to orange theme</button>
    </div>
  )
}
```

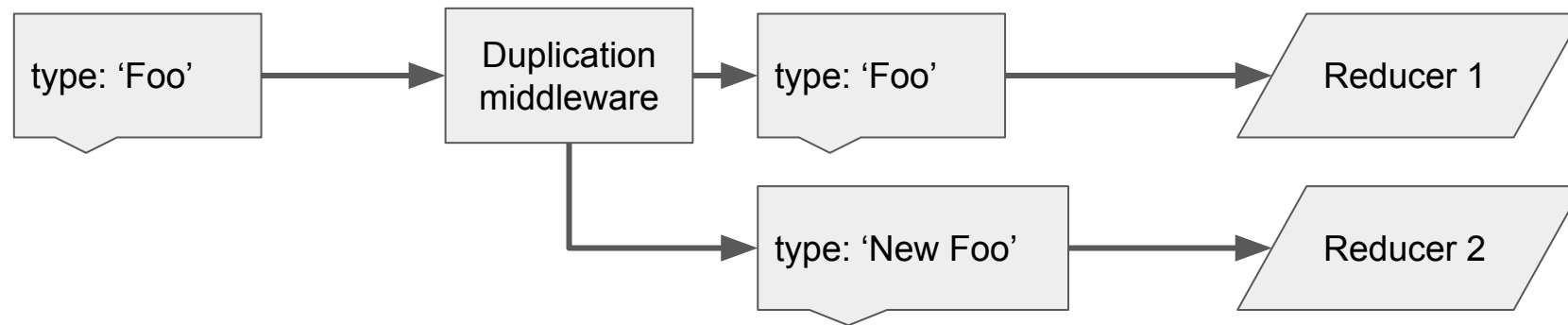
How to use hooks with redux - useStore

- In order to access store

```
const Component = () => {  
  const store = useStore();  
  
  console.log(store);  
  return 'FooBar';  
}
```

Middlewares and other cool tricks

- As reducer, listens on actions, but catches them before they are passed to reducers
- Can observe, modify action or even prevent it from reaching reducers
- Usually middleware is used to add some side effect to action

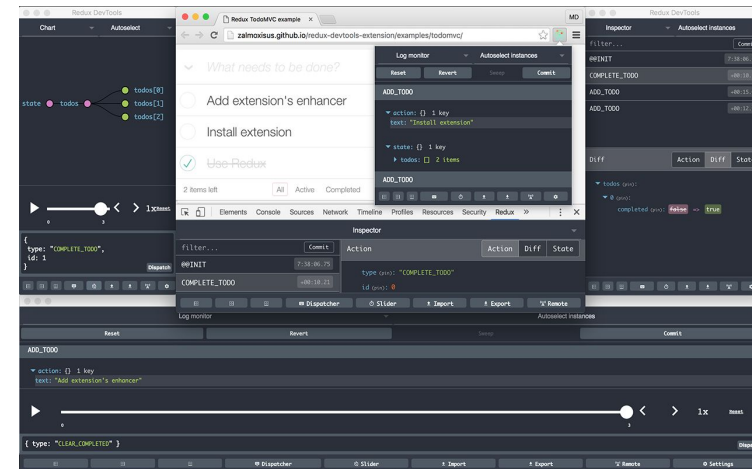


Redux toolkit

- All in one library
- Heavily opinionated
- Reducers replaced with slices
 - Map of reducers
- API creator - allows you to easily setup endpoints connected to redux
- A lot of abstraction, quite an overkill for small apps, opinionated = less flexibility.
(Jotai - easier and minimalistic, granular state, more flexible, less boilerplate, lightweight)

Redux DevTools

- a browser extension
- useful for debugging application's state changes



You might not need redux - useReducer

- Hook introduced in React version 16.8.0
- Introduces reducers to core React
- Its meant to be used for complex component state updates
 - More than two "setState" calls in one callback
 - Every setState triggers one render always
 - Multiple setState have negative performance impact
- useReducer is here to prevent developers store objects in state (useState)
 - Trigger unnecessary re-renders
- useReducer on its own cannot replace redux
 - Lacks optimizations, middlewares, namespacing, context, etc.
 - Would require additional functions to fully replace redux
 - But at that point, you have implemented redux library
- On its own (with clever context and memo usage), can replace redux in smaller scale applications

Let's do some coding

- State vs Redux vs UseReducer -
<https://codesandbox.io/s/friendly-lovelace-sf7s59>

Homework

- Deploy your application

Table - let's write some code

- Deploy application
- Log out
- Table