React Patterns

© Edument 2020

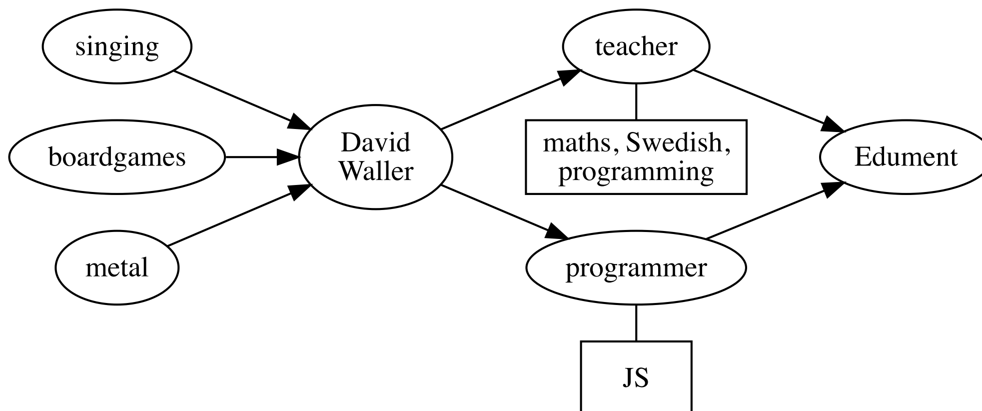
Seminar overview

1. Setting the scene
2. Organising files
3. Component model
4. Project setup
5. Component communication
6. Dependencies
7. State management
8. Error handling
9. Testing
10. Wrapping up

1-1. Setting the scene

React ftw!

Hello! *1-1-1*



https://blog.krawaller.se

david@krawaller.se

...in other words, **exactly** the kind of person you should be wary of! *1-1-2*

- Dont listen to **religion**!
- Instead look for **motivated opinion**...
- ...and then **make your own** for your project

© Edument 2020

Ⓠ How many JavaScript frameworks are there? *1-1-3*

Ⓐ https://dayssincelastjavascriptframework.com/ *1-1-4*

But, **React** is a very solid choice. *1-1-5*

Go with **React**, it is **super awesome**! *1-1-6*

But, be humble to **React's light weight**: *1-1-7*

React has **very few opinions** compared to other frameworks. *1-1-8*

This is a danger, as we **risk making implicit decisions**.

Therefore: *1-1-9*

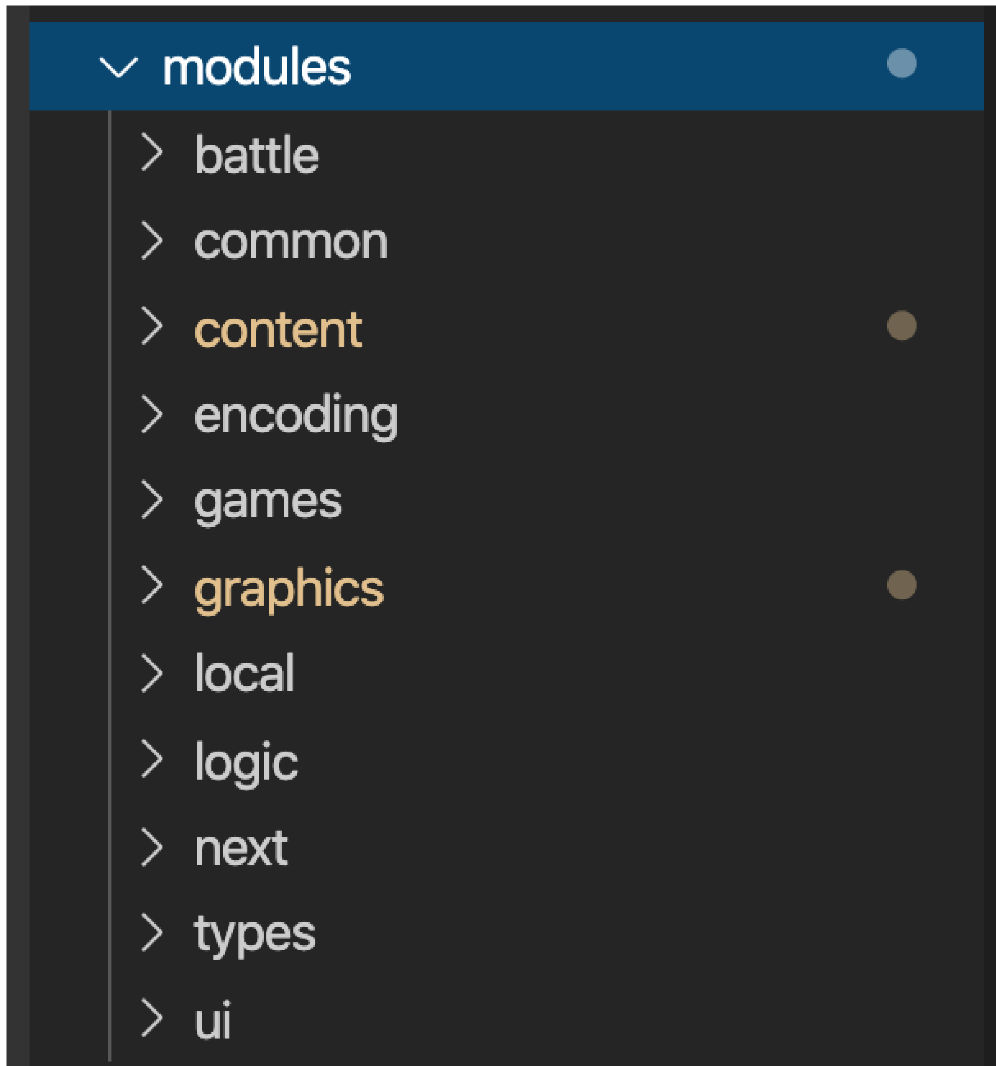
Make **active decisions**! *1-1-10*

© Edument 2020

1-2. Organising files

Arranging the sock drawer

Have a top-level **folder per concern** *1-2-1*



- Each top-level concern is a **separate API surface**.. *1-2-2*
- ..and therefore have **separate test suites**
- **Refactoring** one folder should **not affect** another..
- ..but **changing** a folder's API **should**

(If you're using Redux - more on that later - that should probably be a top-level folder!) *1-2-3*

© Edument 2020

Split concerns into **separate toplevel folders** *1-2-4*

Ⓠ Are these good rules? *1-2-5*

```
{
  "rules": {
    "max-lines": ["error", 200],
    "max-lines-per-function": ["error", { "max": 20 }],
    "max-statements": ["error", 10]
  }
}
```

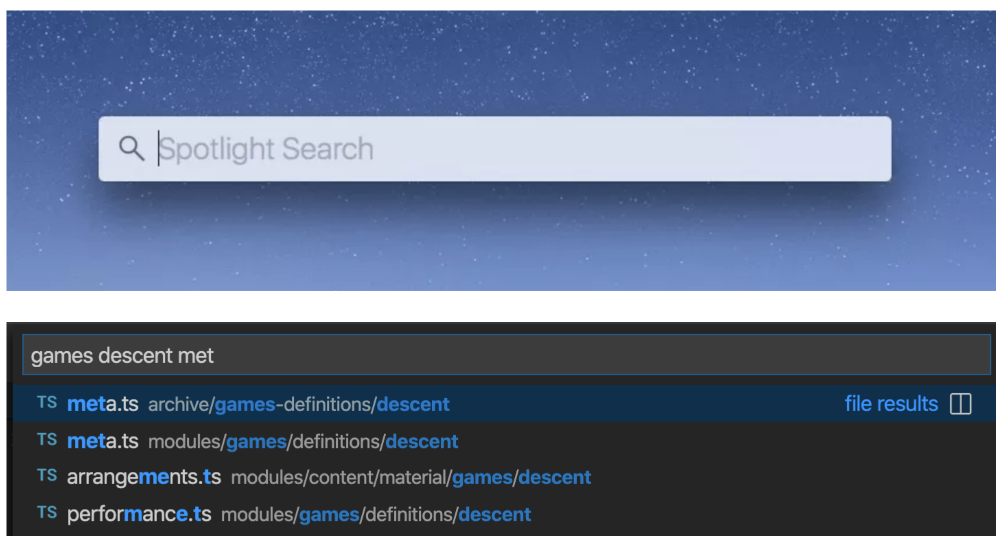
Ⓐ No. But yes! *1-2-6*

The tradeoff: *1-2-7*

- a small function/file is **easier to read**..
- ..but means **more indirection**

Good discussion in Fun fun functions - Straight-line code over functions

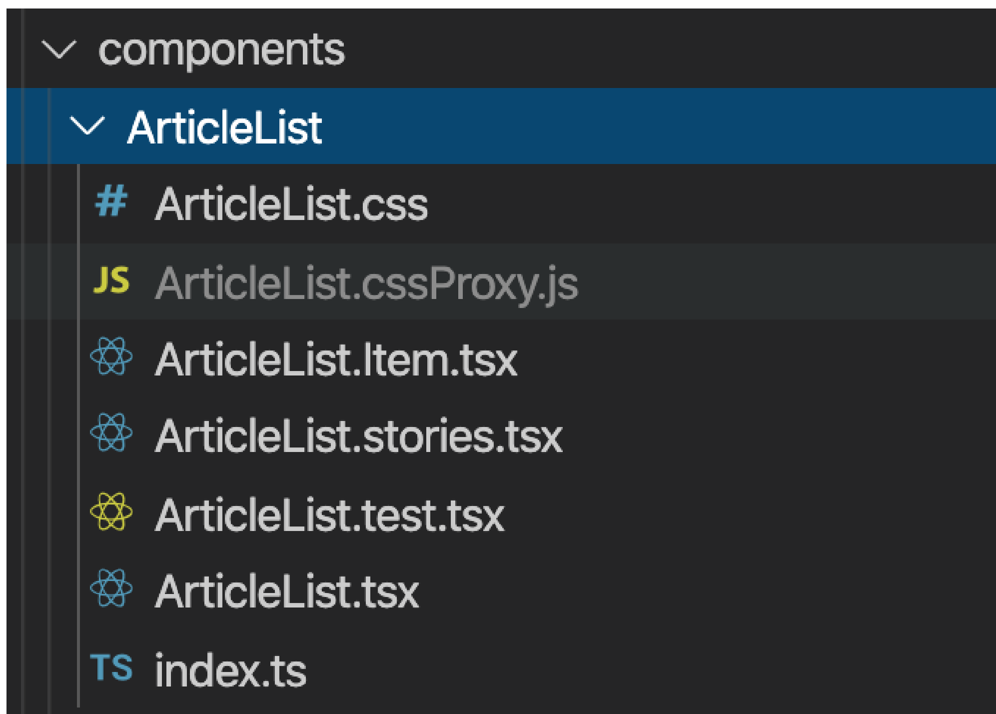
1-2-8



- **Files** (and folders!) are **cheap** *1-2-9*
- **Fuzzy file search** is a **game changer**
- **Scrolling** is **expensive**
- Ergo, **one thing (component!) per file**

© Edument 2020

Therefore: **component folders**! *1-2-10*



1-2-11

- Folder **named from component**
- Component in **file with same name**
- Reexport from **index file**
- **Subcomponents** as `MainComp.SubComp.jsx`
- Co-locate **tests** as `MainComp.test.jsx`
- Co-locate **all other component-specific stuff**!

Co-locate everything to do with a component into a **dedicated folder**, with one thing per file. *1-2-12*

If you agree with the idea of **component folders**, consider adding a **stub command** to quickly start a new component! *1-2-13*

```
npm run stubComponent MyNewComponent
```

This can be done using Yeoman, or just a **node file** with some JS copying templates! *1-2-14*

© Edument 2020

Good component scaffolding brings **two advantages**: *1-2-15*

- **Quick development**
- **Solidifies setup** and **pattern choices**

Have a **component scaffolding** setup! *1-2-16*

1-3. Component model

What dress to wear

After choosing React, there are still **lots more choices** to be made! *1-3-1*

For example...

An ancient beer fridge: *1-3-2*

```
const Clicker = React.createClass({
  getInitialState() {
    return { count: 3 };
  },
  more() {
    this.setState({ count: this.state.count + 1 });
  },
  render() {
    return (
      <div>
        <p>{this.state.count} bottles of beer on the wall</p>
        <button onClick={this.more}>Buy more</button>
      </div>
    );
  }
});
```

© Edument 2020

A classy beer fridge:

1-3-3

```
class Clicker extends React.Component {
  state = { count: 3 };
  more = () => this.setState({ count: this.state.count + 1 });
  render() {
    return (
      <div>
        <p>{this.state.count} bottles of beer on the wall</p>
        <button onClick={this.more}>Buy more</button>
      </div>
    );
  }
}
```

Hook version:

1-3-4

```
const Clicker = () => {
  const [count, setCount] = useState(3);
  const more = setCount(count + 1);
  return (
    <div>
      <p>{count} bottles of beer on the wall</p>
      <button onClick={more}>Buy more</button>
    </div>
  );
};
```

Q

Which one is the best?

1-3-5

© Edument 2020

A

Q

Really? Why?

A

Logic sharing with classes:


```
const MyComponent = recompose(
  withDetailsExpander(),
  withLocation(),
  withAuth()
)(MyComponentInner);
```

Logic sharing with hooks:


```
const MyComponent = (props) => {
  const { ... } = useDetailsExpander()
  const { ... } = useLocation()
  const { ... } = useAuth()
  // ... the rest
}
```

© Edument 2020

This means: 1-3-10

- **Flatter** render tree
- Less **props pollution**
- Less **indirection**

hooks > HoC:s

Also, with hooks, **related logic** is **bunched together** better. 1-3-11

But! 1-3-12

- Classes are **very well understood**
- Hooks **takes some getting use to**
- **Stale closure** hook bugs can be **hard to debug**

Be humble to this!

Q Can you **spot the bug**? 1-3-13

```
const readSessions = useCallback(() => {
  const sessions = getSessionList(meta.id);
  setSessionInfo({
    sessions,
    status: "loaded"
  });
}, [setSessionInfo]);
```

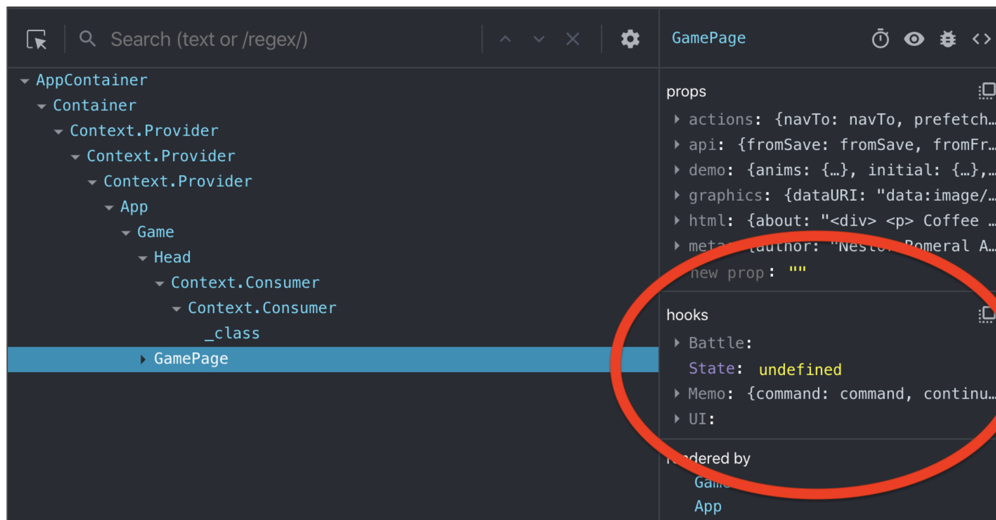
A 1-3-14

```
const readSessions = useCallback(() => {
  const sessions = getSessionList(meta.id);
  setSessionInfo({
    sessions,
    status: "loaded"
  });
}, [setSessionInfo]); // <-- Demons live here (missing meta)
```

Always pay extra attention to the **dependency array** in **code review**. 1-3-15

© Edument 2020

Make a habit of checking the **hooks section in the devtools**: *1-3-16*



And **never** use hooks without a seatbelt: *1-3-17*

```
{
  "rules": {
    "react-hooks/rules-of-hooks": "error",
    "react-hooks/exhaustive-deps": "error"
  }
}
```

Skip classes for **hooks**, but come **prepared** and be **humble** *1-3-18*

PS: custom hooks are great for code reuse, but also just for **splitting things up**! *1-3-19*

Imagine a **fat component**: *1-3-20*

```
const Modal = props => {
  const { title, content } = props;
  const [isOpen, setIsOpen] = useState(false);
  const open = useCallback(() => {
    setIsOpen(true);
  }, [setIsOpen]);
  const close = useCallback(() => {
    setIsOpen(false);
  }, [setIsOpen]);
  // render using isOpen, open, close, title, content
};
```

© Edument 2020

We move the guts into a **custom hook**: *1-3-21*

```
const useModal = () => {
  const [isOpen, setIsOpen] = useState(false);
  const open = useCallback(() => {
    setIsOpen(true);
  }, [setIsOpen]);
  const close = useCallback(() => {
    setIsOpen(false);
  }, [setIsOpen]);
  return [isOpen, open, close];
};
```

This can live in `Modal.useModal.js` in the same folder.

Now our component code is **focused on the rendering**: *1-3-22*

```
const Modal = props => {
  const { title, content } = props;
  const [isOpen, open, close] = useModal();
  // render using isOpen, open, close, title, content
};
```

A final note regarding **state** when you **convert from class to hooks**: *1-3-23*

- **`this.state`** and **`this.setState`** usage becomes **`useState`** calls
- **`this.nonRenderState`** becomes **`useRef`** calls

1-4. Project setup

Building the wardrobe

© Edument 2020

We're gonna cover:

1-4-1

- (a) Babel
- (b) NextJS
- (c) StorybookJS
- (d) CSS in JS
- (e) Typing solution

(a) First - Babel!

1-4-2

(Q) Waddya mean Babel, surely we just **use Create React App**?

1-4-3

(A) **Probably not.** And if you do, **invest time in the settings**.

1-4-4

CRA means lock-in by design!

The gist;

1-4-5

- **Owning the project setup** is **hard** and **takes time**..
- ..but it inevitably means **less compromise** and **more adaptability** down the line

© Edument 2020

Remember this advice? 1-4-6

Make **active decisions**!

It is of **double importance** for the **build setup** and related infrastructure!

Therefore: 1-4-7

Have a **home-rolled Babel/bundler setup** with dedicated maintenance

In my experience, the **freedom that brings** is worth a lot down the line.

ⓑ Next - Next! 1-4-8

Ⓠ We agree **CRA** is flaky, and yes we should **own the setup**, but we came here hoping for some **practical tips on the subject**! Don't you have any beyond *do it yourself*? 1-4-9

Ⓐ Fine: strongly **consider using NextJS** as a platform for your app! 1-4-10

© Edument 2020

NextJS gives you *1-4-11*

- **Quick setup** just like CRA
- (that can **hook into your Babel config**)
- **dev setup** with **HMR**
- **Server side rendering**
- Folder-based **routing solution**...
- ...with automatic **code splitting**

It used to be only a server-client package, but it can now **export to a static site**. *1-4-12*

Thus we can use it as a **CRA replacement**!

Take a close look at using **NextJS** as a foundation *1-4-13*

(c) Also - **check out** StorybookJS! *1-4-14*

(Q) Isn't Storybook just a **component gallery**? We're not building a UI library, we're building an app! *1-4-15*

(A) **Wrong**! The biggest value of Storybook is letting us **iterate quickly on components**. *1-4-16*

It is super useful in **every** React project, big and small (and tiny!), no matter what we're building!

Every React project should have a **Storybook**! *1-4-17*

(d) Now - **CSS in JS**! *1-4-18*

© Edument 2020

With **regular CSS**, it is hard to say

1-4-19

- **what styles** a component will get
- **which components** a style will be applied to

This **scales really badly** and leads to **specificity hell**.

1-4-20

You should apply a technique to **mitigate this**!

This can be something like **CSS Modules**, or a full-blown **CSS in JS solution**.

1-4-21

The important thing is that you **don't use naked CSS** (for a non-trivial project).

Exactly which one you choose is less important!

Have a **CSS solution**!

1-4-22

Q Surely all that only applies to **CSS n00bs**? We understand **BEM** so we don't need that stuff!

1-4-23

A Good for you! :)

1-4-24

e Finally - typing solution!

1-4-25

Q What, **TypeScript**?

1-4-26

A Past me:

1-4-27

No, TS is just a safety blanket for backenders forced to write JS. Once you truly grok JS you don't need it.

© Edument 2020

Ⓐ Current me: *1-4-28*

Hells to the yes! **Everyone** should use TS.

Ⓠ What about **Flow**? *1-4-29*

Ⓐ No. *1-4-30*

Ⓠ But we use **proptypes**! *1-4-31*

Ⓐ Not good enough. *1-4-32*

Here's the sales pitch: *1-4-33*

- Setting up TS is **super easy** (nowadays)
- Benefits even **before adding any types**
- Can **gradually introduce types**
- The React model **plays really well with strong typing**

Strongly consider **TypeScript**! *1-4-34*

If you don't - **avoid this pattern**: *1-4-35*

```
<MyComponent {...props}>
```

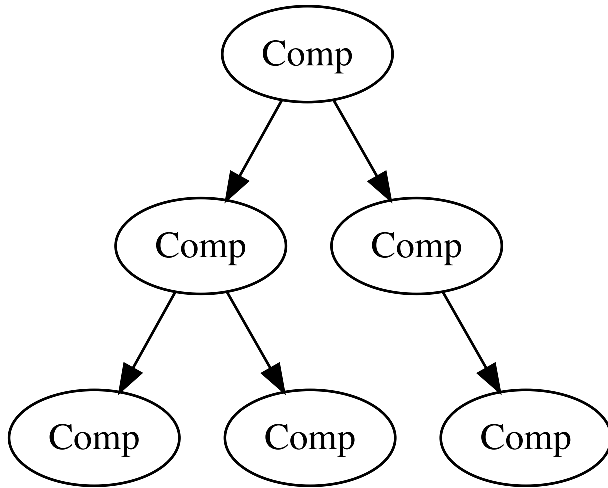
(in fact, avoid it anyway)

1-5. Component communication

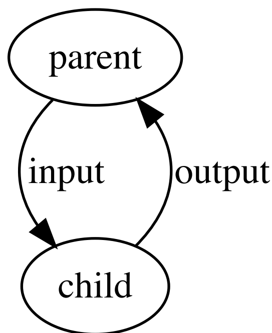
App synapses

© Edument 2020

A React app is a **pyramid of components**: *1-5-1*



This makes **component communication** a central piece of the puzzle. *1-5-2*



Here's an **Angular component** (yuck): *1-5-3*

```
@Component({
  selector: "combat"
  // ...other view stuff
})
export class CombatComponent {
  @Input() arenaId: string;
  @Output() outcome = new EventEmitter();
  // implementation
}
```

Can you spot the **inputs** and **outputs**?

© Edument 2020

And **when we use it**: *1-5-4*

```
<combat (outcome)="handleOutcome($event)" [arenaId]="aId" />
```

We can again **identify inputs and outputs**

Here we're using a **React version**: *1-5-5*

```
<Combat onOutcome={handleOutcome} arenaId={aId} />
```

Distinguishing is still ok-ish since we use **good names**.

Otherwise it can be harder: *1-5-6*

```
<Item data={item} select={register} />
```

Is `select` an **input or output**?

One possible solution - **group outputs** into a single prop! *1-5-7*

```
interface BattleActions {
  goToHistory: (step: number) => void;
  goToBattleControls: () => void;
  deleteCurrentSession: () => void;
}

type BattleProps = {
  actions: BattleActions; // <-- all outputs grouped here
  session: AlgolSession;
  battle: AlgolBattle;
};

export const Battle: FunctionComponent<BattleProps> = props => {
  // ...
};
```

Now the **distinction is clear**: *1-5-8*

```
<Battle session={session} battle={battle} actions={actions}>
```

© Edument 2020

In a **smart parent** it is very common to be passing **different actions to different children**, but the grouping pattern allows us to safely cheat! *1-5-9*

```
return (
  <Fragment>
    <Breadcrumbs actions={actions} ... />
    <Board actions={actions} ... />
    <Controls actions={actions} ... />
  </Fragment>
)
```

(if we're using a typing solution, that is)

If you **don't like the grouping pattern**, then consider **checking output names in the linter**: *1-5-10*

```
{
  "rules": {
    "react/jsx-handler-names": "error"
  }
}
```

This enforces:

```
<MyComponent onChange={this.handleChange} />
```

Make it easy to **identify outputs** *1-5-11*

1-6. Dependencies

Brain implants

© Edument 2020

Here's an **Angular component** again: *1-6-1*

```
@Component({
  selector: "combat"
  // ...other view stuff
})
export class CombatComponent {
  constructor(private battleService: BattleService) {}
  @Input() arenaId: string;
  @Output() outcome = new EventEmitter();
  // implementation
}
```

Can you spot the **dependency**?

In **React**, which **doesn't have a dependency injection system**, we are probably **importing the dependency**: *1-6-2*

```
import { BattleService } from "../services";

const Combat = props => {
  // implemented using BattleService
};
```

In a **Jest unit test** we can mock the imported dependencies if needed. But in a **Storybook** we **can't**! At least not easily. *1-6-3*

This is one of many reasons for having a **central dependency strategy**. Which we can accomplish via the Context API! *1-6-4*

Q **Context**? Hold on. Isn't that **just for library authors**? *1-6-5*

A So people like me have said. For which **I'm sorry**! *1-6-6*

I think of Context as a **streamlined DI system** (that blows Angular's out of the water).

© Edument 2020

If we have a typesystem we define an **interface for our dependencies**: *1-6-7*

```
interface Dependencies {
  battleService: BattleService;
  localStorage: LocalStorage;
  // ... and more
}
```

We make a **dummy version** with the same shape that only logs to console: *1-6-8*

```
const dummyDeps: Dependencies = {
  battleService: dummyBattleService,
  localStorage: dummyLocalStorage
  // ...and more
};
```

The **dependency context** is then created using the **dummy as default**: *1-6-9*

```
const DependencyContext = React.createContext(dummyDeps);
DependencyContext.displayName = "DependencyContext";
```

In consumers we **access the deps** via `useContext`: *1-6-10*

```
const CombatComponent = props => {
  const { battleService } = useContext(DependencyContext);
  // ...do stuff with battleService
};
```

Btw, consuming contexts is **butt ugly in class components**. Go hooks! *1-6-11*

© Edument 2020

We put a **provider at the top of the tree** 1-6-12

```
const App = props => (
  <DependencyContext.Provider value={realDeps}>
    <Main />
  </DependencyContext.Provider>
);
```

Storybook scenarios will work out of the box since they'll use the **dummy dependencies**. 1-6-13

For **tests involving deps** we wrap the component with a provider of our mocks. 1-6-14

```
const result = (
  <DependencyContext.Provider value={mockDeps}>
    <SomeComponent other={stuff} />
  </DependencyContext.Provider>
);
```

An additional benefit of this pattern is to **clearly identify and catalog all dependencies**, and just having that discussion often means better organisation. 1-6-15

Q But, **waitaminute**. What do we mean by **component dependency**? Would that be **all imports that aren't child components or hooks**? 1-6-16

A Pretty much! 1-6-17

- non-component non-hook imports...
- ...that could mess up test and/or storybook scenario...
- ...or that we want test the usage of

But if **2nd and 3rd points are void**, then just import the damn thing. 1-6-18

If no one cares then no one cares!

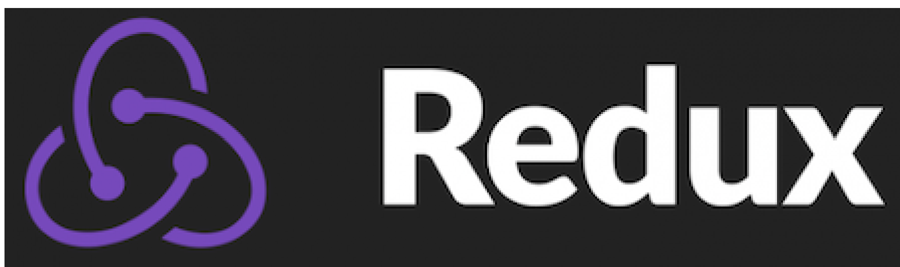
© Edument 2020

(..although there's something satisfying about having all imports from other top-level folders come via the context..) *1-6-19*

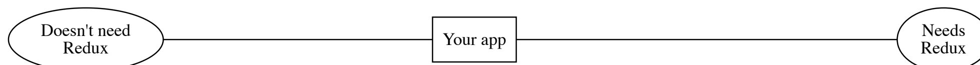
Provide **dependencies** in an organised manner *1-6-20*

1-7. State management

Q Nothing to say except **use Redux**, right? *1-7-1*



A It **depends**, of course: *1-7-2*



Where is the line?

Past me: *1-7-3*

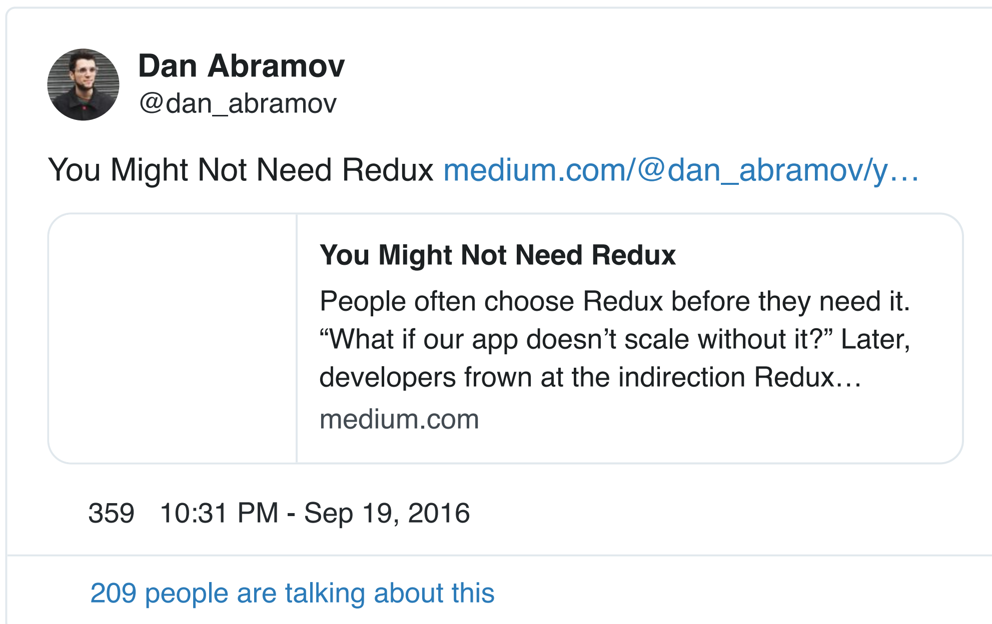
If you know Redux you **benefit even for small apps**

Current me: *1-7-4*

You very likely **don't need Redux**

© Edument 2020

1-7-5


Dan Abramov
@dan_abramov

You Might Not Need Redux medium.com/@dan_abramov/y…

You Might Not Need Redux
People often choose Redux before they need it. “What if our app doesn’t scale without it?” Later, developers frown at the indirection Redux…
medium.com

359 10:31 PM - Sep 19, 2016

209 people are talking about this

1-7-6

Complexity ≠

Redux shines if...

- you have **complex state**
- that is used in **multiple places**

1-7-7

Think hard before adopting Redux

1-7-8

Q So **what are we supposed to do** then?

After all, **React is just a view layer** thing?

1-7-9

A **React's state management** - especially through hooks and context - is **more than enough** for most apps!

1-7-10

© Edument 2020

The gist: *1-7-11*

- **keep local state local**
- if a cousin needs the same state, **hoist to common ancestor**
- **don't be afraid to propdrill** a generation or two
- if that gets out of hand, **use context**

For **common ancestor** state keepers, a good pattern is to make **state hooks**: *1-7-12*

```
useMyState = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  const actions = useMemo(
    () => ({
      // ...obj with methods calling dispatch
    }),
    [dispatch]
  );
  return [state, actions];
};
```

Consumed like this: *1-7-13*

```
const SmartComponent = () => {
  const [state, actions] = useMyState();
  // render passing (selected) state and actions to children
};
```

Strategically **place the state in the pyramid** *1-7-14*

This **scales way better** than past me could possibly imagine!

1-8. Error handling

Making things behave

Sometimes, there are bugs. *1-8-1*

© Edument 2020

Here's what we'll cover: *1-8-2*

- (a) Render errors
- (b) Non-render errors
- (c) Error metadata

(a) If an error is throw **during render** in React, we get a **white page of death**. *1-8-3*

We can catch them using **Error boundaries**! *1-8-4*

- Probably definitely at the **top of the pyramid**
- maybe at **other strategic points**

There's **no hook yet** for this (boo!). *1-8-5*

```
class ErrorBoundary extends React.Component {
  state = { error: null }

  static getDerivedStateFromError(error) {
    return { error };
  }

  render() {
    return this.state.error
      ? <ErrorDisplay error={this.state.error}>
      : this.props.children
  }
}
```

The **ErrorDisplay** can just be an apologetic message, or a UI to submit a bug report. *1-8-6*

Often it will also **report to** Sentry or a similar service.

© Edument 2020

Employ **Error Boundaries** *1-8-7*

ⓑ But **non-render errors** need a different strategy! *1-8-8*

These are typically

- **event handlers**
- **useEffect** calls

A powerful pattern is to have the top-level Error Boundary **provide a setter through a context**: *1-8-9*

```
report = (error) => this.setState({ error }),
render() {
  if (this.state.error) {
    return <ErrorDisplay error={this.state.error}>
  }
  return (
    <ErrorContext.Provider value={this.report}>
      { this.props.children }
    </ErrorContext.Provider>
  )
}
```

Now **careful children** can use that in dangerous handlers calls: *1-8-10*

```
const CarefulComp = props => {
  const report = useContext(ErrorContext);
  const handler = () => {
    try {
      dangerousThing();
    } catch (e) {
      report(e);
    }
  };
  // ...
};
```

Ⓠ Bah. My silly teammates will **forget to do that**, or they can't be bothered! *1-8-11*

Ⓐ Truth! So we need to do some clever **prep work**. *1-8-12*

© Edument 2020

Probably you have a **`Button`** ui component? Do this: *1-8-13*

```
const Button = props => {
  const { onClick, text } = props;
  const report = useContext(ErrorContext);
  const handler = useCallback(e => {
    try {
      onClick(e);
    } catch (error) {
      report(error);
    }
  });
  return <button onClick={handler}>{text}</button>;
};
```

For **`useEffect`** calls you can make an **autoreporting version**! *1-8-14*

```
const useDangerousEffect = effect => {
  const report = useContext(ErrorContext);
  try {
    useEffect(effect);
  } catch (error) {
    report(error);
  }
};
```

Provide an **error reporter**, and hook it into the boundary! *1-8-15*

(c) A nice pattern compatible with this is to **decorate your errors**! *1-8-16*

You **don't need inheritance** for this, just **add stuff directly onto the error**!

© Edument 2020

For example, make the `Button` **watermark the error** with an id: *1-8-17*

```
const Button = props => {
  const { onClick, text, buttonId } = props;
  const report = useContext(ErrorContext);
  const handler = useCallback(e => {
    try {
      onClick(e);
    } catch (error) {
      error.buttonId = buttonId;
      report(error);
    }
  });
  return <button onClick={handler}>{text}</button>;
};
```

This goes for **non-react logic** too. Provide good contexts to your errors! *1-8-18*

```
try {
  dangerousThing();
} catch (err) {
  err.meta = usefulStuffForDebugging;
  throw err;
}
```

Decorate your errors for debugging bliss *1-8-19*

1-9. Testing

BDD, TDD, TBD?

First off - Jest **is awesome**. *1-9-1*

You should have a very good reason for not using it.

So, for testing React we use **Enzyme**, right? *1-9-2*

© Edument 2020

Ⓐ Past me: *1-9-3*

Of course! We **shallow render** to focus on the current unit.

Ⓐ Current me: *1-9-4*

Hell no! The shallow renderer is **very different** from the actual render cycle.

Also have to **jump through hoops** to make it **work with hooks**.

The React Testing library is pretty sweet! *1-9-5*

But it might be enough to just **use `react-test-renderer` directly**.

Main point being: the more your **tests resemble actual use**, the better. *1-9-6*

Ⓠ But, that means **rendering the full tree**! What if my component has **unruly children**? *1-9-7*

You know, the problem that shallow rendering so elegantly solves?

Ⓐ If a child misbehaves, just **mock it**: *1-9-8*

```
import { MyComponent } from "./MyComponent";
import { AnotherComponent } from "../AnotherComponent";
jest.mock("../AnotherComponent", () => ({
  AnotherComponent: () => <div />
}));

// now test MyComponent without AnotherComponent messing things up
```

Jest mocking is a **game changer**.

© Edument 2020

Q On the same subject - is this a good **Redux test**? *1-9-9*

```
describe('the Notification reducer', () => {
  test('handles addNotification correctly', () => {
    const initialState = { ... }
    const action = { ... }
    const result = notificationReducer(initialState, action)
    expect(result).toMatchSomeExpectation()
  })
})
```

A Nope! *1-9-10*

- Fake state (and action), **risk testing nonexisting scenarios**
- Testing **implementation detail** (the app never calls the reducer)

Here's a **better version**: *1-9-11*

```
describe('the Notification reducer', () => {
  test('handles addNotification correctly', () => {
    const store = newTestStore()
    store.dispatch(someAction()) // These two actions are just...
    store.dispatch(anotherAction()) // ...to build initial state
    store.dispatch(addNotification({ ... }))
    const result = store.getState()
    expect(result).toMatchSomeExpectation()
  })
})
```

The **API surface** of the Redux layer is the **store**, so that's what we should test! *1-9-12*

Have your **tests match reality** as closely as possible. *1-9-13*

© Edument 2020

So. **Snapshot testing**. *1-9-14*

```
describe("the GameList component", () => {
  test("renderes full list correctly", () => {
    const output = ReactTestRenderer.create(
      <GameList list={allGames} />
    ).toJSON();
    expect(output).toMatchSnapshot();
  });
  test("renders ok with empty list", () => {
    const output = ReactTestRenderer.create(<GameList list={[]} />).toJSON();
    expect(output).toMatchSnapshot();
  });
});
```

Q Isn't this **dumb**? *1-9-15*

A Past me: *1-9-16*

Yes, very.

A Current me: *1-9-17*

Nope! Same thing as checking stuff in the output manually, only **way easier** and **covers more**.

Also developers must **actively update snapshots**, and **render changes shows in git diff**.

Q You now how **Dan Abramov is always right**, right? *1-9-18*

1-9-19

Dan Abramov
@dan_abramov

Unpopular opinion: component unit testing is overrated.

324 9:43 PM - Jul 24, 2016

155 people are talking about this

© Edument 2020

And remember, domain logic should be a **separate top-level concern** and not live in the component anyway.

1-9-20

Use **snapshot testing**, and **maybe nothing else**!

1-9-21

1-10. Wrapping up

http://edument.se

1-10-2

Code **`EduReact`** for 30% off our React courses

david@krawaller.se

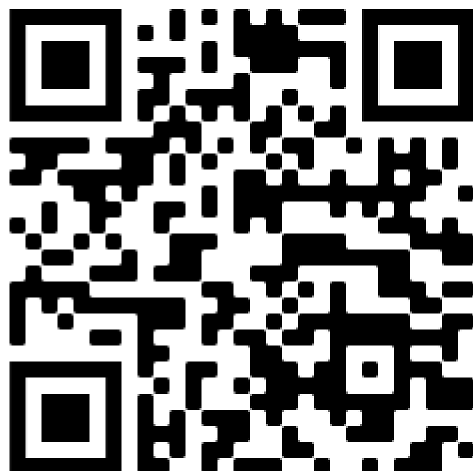
1-10-3

Don't be a stranger!

We ❤️ feedback!

1-10-4

https://edument.typeform.com/to/FKWQbU



© Edument 2020

External links

1-2-7 Fun fun functions - Straight-line code over functions: https://youtu.be/Bks59AaHe1c

1-2-14 Yeoman: https://yeoman.io/

1-4-2 Babel: https://babeljs.io/

1-4-10 NextJS: https://nextjs.org/

1-4-14 StorybookJS: https://storybook.js.org/

1-4-29 Flow: https://flow.org/

1-4-31 proptypes: https://reactjs.org/docs/typechecking-with-proptypes.html

1-7-5 September 19, 2016: https://twitter.com/dan_abramov/status/777983404914671616?ref_src=twsrc%5Etfw

1-8-6 Sentry: https://sentry.io/

1-9-1 Jest: https://jestjs.io/en/

1-9-5 React Testing library: https://testing-library.com/docs/react-testing-library/intro

1-9-19 July 24, 2016: https://twitter.com/dan_abramov/status/757315414284201985?ref_src=twsrc%5Etfw

© Edument 2020