Some of the insights for better development, scalability and durability of a REACT project.

Author- VENKAT AVINASH KARRA

Contents:

Introduction

What is clean code, and why do I care?

Clean code is a consistent style of programming that makes your code easier to write, read, and maintain. Often a developer spends time on a problem, and once the problem is solved, they make a pull request. I contend that you aren't done just because your code "works."

Now is your chance to clean it up by removing dead code (zombie code), refactoring, and removing any commented-out code! Strive for maintainability. Ask yourself, "Will someone else be able to understand these code six months from now?"

In simpler terms, write code that you would be proud to take home and show your mother.

Why do you care? Because if you're a good developer, you're lazy. Hear me out – I mean that as a compliment. A good developer, when faced with a situation where they must do something more than once, will generally find an automated (or better) solution to complete the task at hand. So, because you're lazy, subscribing to clean-code techniques will decrease the frequency of changes from pull-request code reviews and the need to come back to the same piece of code over and over.

Clean code passes the "smell test"

Clean code should pass the smell test. What do I mean by that? We've all looked at code (our own or others') and said, "Something's not quite right here." Remember, if it doesn't feel right, it probably isn't. Code that's well thought out just comes together. If it feels like you're trying to fit a square peg into a round hole, then pause, step back, and take a break. Nine times out of 10, you'll come up with a better solution.

Clean code is DRY

DRY is an acronym that stands for "Don't Repeat Yourself." If you are doing the same thing in multiple places, consolidate the duplicate code. If you see patterns in your code, that is an indication it is prime for DRYing. Sometimes this means standing back from the screen until you can't read the text and literally looking for patterns.

Occasionally, DRYing your code may actually increase code size. However, DRYing your code also generally improves maintainability. Be warned that it's possible to go too far with DRYing up your code, so know when to say when.

Naming things

We should all give serious thought to variable names, function names, and even filenames.

Here are a few guidelines:

- Boolean variables, or functions that return a boolean value, should start with “is,” “has” or “should.”

```
Eg:

// Dirty
const done = current >= goal;

// Clean
const isComplete = current >= goal;
```

- Functions should be named for what they do, not how they do it. In other words, don’t expose details of the implementation in the name. Why? Because how you do it may change some day, and you shouldn’t need to refactor your consuming code because of it. For example, you may load your config from a REST API today, but you may decide to bake it into the JavaScript tomorrow.

```
// Dirty
const loadConfigFromServer = () => {
  ...
};

// Clean
const loadConfig = () => {
  ...
};
```

Here are some best practices to follow when architecting your React applications:

- Use small functions, each with a single responsibility. This is called the single responsibility principle. Ensure that each function does one job and does it well. This could mean breaking up complex components into many smaller ones. This also will lead to better testability.
- Be on the lookout for leaky abstractions. In other words, don’t impose your internal requirements on consumers of your code.
- Follow strict linting rules. This will help you write clean, consistent code.

Clean code doesn’t (necessarily) take longer to write

I hear the argument all the time that writing clean code will slow productivity. That’s a bunch of hooey. Yes, initially you may need to slow down before you can speed up, but eventually your pace will increase as you are writing fewer lines of code.

And don’t discount the “rewrite factor” and time spent fixing comments from code reviews. If you break your code into small modules, each with a single responsibility, it’s likely that you’ll never have to touch most modules again. There is time saved in “write it and forget it.”

Practical examples of dirty code vs. clean code

DRY up this code

Look at the code sample below. Go ahead and step back from your monitor as I described above. Do you see any patterns? Notice that the component **Thingie** is identical to **ThingieWithTitle** with the exception of the **Title** component. This is a perfect candidate for DRYing.

```
// Dirty
import Title from './Title';
export const Thingie = ({ description }) => (
  <div class="thingie">
    <div class="description-wrapper">
      <Description value={description} />
    </div>
  </div>
);

export const ThingieWithTitle = ({ title, description }) => (
  <div>
    <Title value={title} />
    <div class="description-wrapper">
      <Description value={description} />
    </div>
  </div>
);
```

Here we’ve allowed the passing of children to **Thingie**. We’ve then created **ThingieWithTitle** that wraps **Thingie**, passing in the **Title** as its children.

```
// Clean
import Title from './Title';
export const Thingie = ({ description, children }) => (
  <div class="thingie">
    {children}
    <div class="description-wrapper">
      <Description value={description} />
    </div>
  </div>
);

export const ThingieWithTitle = ({ title, ...others }) => (
  <Thingie {...others}>
    <Title value={title} />
  </Thingie>
);
```

I hope that I’ve helped you see the benefits of writing clean code and that you can even use some of the practical examples presented here. Once you embrace writing clean code, it will become second nature. You (and your future self) will soon appreciate the “write it and forget it” way of life.

Redux and Sagas

Redux is used mostly for application state management. To summarize it, Redux maintains the state of an entire application in a single immutable state tree (object), which can’t be changed directly. When something changes, a new object is created (using actions and reducers). It also acts as global state of the application and also helps is persistence of data in session.

As any redux developer could tell you, the hardest part of building an app are asynchronous calls — how do you handle network requests, timeouts, and other callbacks without complicating the redux actions and reducers?

To manage this complexity, I’ll describe a few different approaches for handling asynchronicity in your app, ranging from simple approaches like redux-thunk, to more full-featured libraries like redux-saga.

Redux-saga is a redux middleware library, that is designed to make handling side effects in your redux app nice and simple. It achieves this by leveraging an ES6 feature called Generators, allowing us to write asynchronous code that looks synchronous, and is very easy to test.

So why should we use redux-saga? Contrary to redux-thunk, you don't end up in callback hell, you can test your asynchronous flows easily and your actions stay pure. Redux-saga is a library that aims to make side effects easier and better by working with sagas. Sagas are a design pattern that comes from the distributed transactions world, where a saga manages processes that need to be executed in a transactional way, keeping the state of the execution and compensating failed processes.

In the context of Redux, a saga is implemented as a middleware (we can't use a reducer because this must be a pure function) to coordinate and trigger asynchronous actions (side-effects). Redux-saga does this with the help of ES6 generators as follows:

```
function* myGenerator() {
 let first = yield 'first yield value';
 let second = yield 'second yield value';
 return 'third returned value';
}
```

Generators are functions that can be paused and resumed, instead of executing all the statements of the function in one pass.

When you invoke a generator function, it will return an iterator object. With each call of the iterator's *next()* method, the generator's body will be executed until the next *yield* statement and then pause:

```
const it = myGenerator();
console.log(it.next()); // {value: "first yield value", done: false}
console.log(it.next()); // {value: "second yield value", done: false}
console.log(it.next()); // {value: "third returned value", done: true}
console.log(it.next()); // {value: undefined, done: true}
```

This can make asynchronous code easy to write and understand. For example, instead of doing this:

```
fetch(url).then((val) => {
 console.log(val);
});
```

With generators, we could do this:

```
let val = yield fetch(url);
console.log(val);
```

Back to redux-saga, we generally have a saga whose job is to watch for dispatched actions:

```
function*  watchRequestDog() {
}
```

To coordinate the logic we want to implement inside the saga, we can use a helper function like takeEvery to spawn a new saga to perform an operation:

```
// Watcher saga for spawing new tasks
function*  watchRequestDog() {
 yield takeEvery('FETCHED_DOG', fetchDogAsync)
}

// Worker saga that performs the task
function* fetchDogAsync() {
}
```

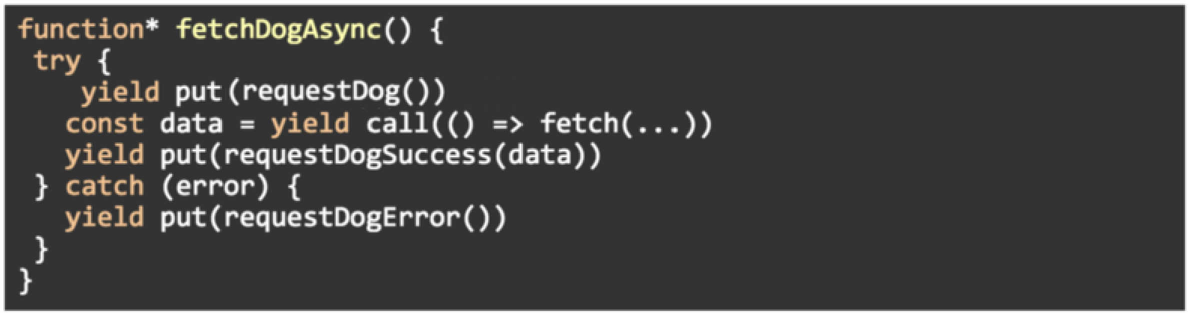
If there are multiple requests, *takeEvery* will start multiple instances of the worker saga. In other words, it handles concurrency for you.

Notice that the watcher saga is another layer of indirection that gives more flexibility to implement complex logic (but may be unnecessary for simple apps).

Now, we could implement the *fetchDogAsync()* function with something like this (assuming we had access to the *dispatch* method):

```
function* fetchDogAsync() {
 try {
    yield dispatch(requestDog())
    const data = yield fetch(...)
   yield dispatch(requestDogSuccess(data))
 } catch (error) {
   yield dispatch(requestDogError())
 }
}
```

But redux-saga allows us to yield an object that declares our intention to perform an operation rather than yielding the result of executing the operation itself. In other words, the above example is implemented in redux-saga in this way:

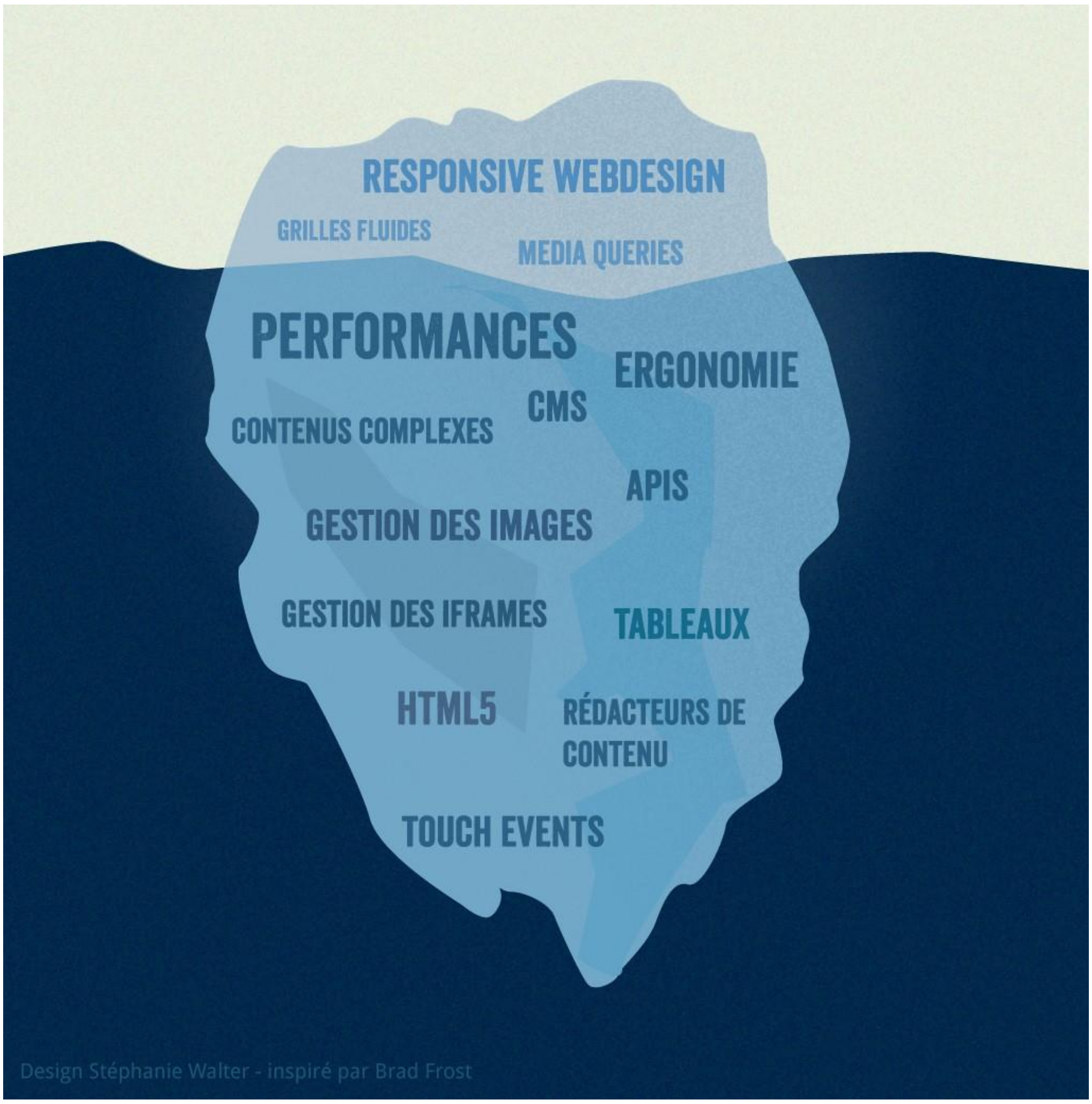


```
function* fetchDogAsync() {
 try {
    yield put(requestDog())
   const data = yield call(() => fetch(...))
   yield put(requestDogSuccess(data))
 } catch (error) {
   yield put(requestDogError())
 }
}
```

Instead of invoking the asynchronous request directly, the method *call* will return only a plain object describing the operation so redux-saga can take care of the invocation and returns the result to the generator.

The same thing happens with the *put* method. Instead of dispatching an action inside the generator, *put* returns an object with instructions for the middleware to dispatch the action.

Responsive UI with Grid system and breakpoints



Material Design's grid system is implemented in Material-UI using the <Grid /> component. Under the hood, the <Grid /> component uses Flexbox properties for high flexibility.

There are two types of grid components: containers and items. To make the layout fluid and adaptive to screen sizes, the item widths are set in percentages. Padding creates spacing between individual items. Finally, there are five types of grid breakpoints: **xs**, **sm**, **md**, **lg**, and **xl**.

Material Design layouts encourage consistency across platforms, environments, and screen sizes by using uniform elements and spacing.

The grid system consists of three components:

1. Columns — Elements on the page are placed within columns, which are defined by percentages rather than fixed values so that the elements can flexibly adapt to any screen size.
2. Gutters — The spaces between the columns are defined by a fixed value at each breakpoint to better adapt the screen size.
3. Margins — The spaces between the left and right sides of the screen are defined by a fixed value similar to gutters, at each breakpoint.

Breakpoints match with a certain screen width and depending on what value you set is how many cards will occupy the space given the amount of available space.

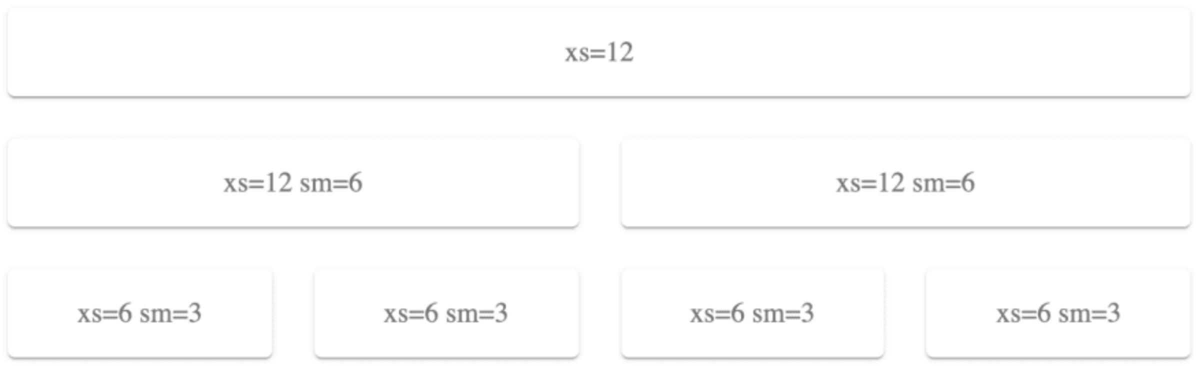
Here is a grid item from **Material-UI**:

xs, extra-small: 0px
sm, small: 600px
md, medium: 960px
lg, large: 1280px
xl, extra-large: 1920px

```
<Grid item xs={A} sm={B} md={C} lg={D} xl={E}>
   <Paper className={classes.paper}>My Grid Item!</Paper>
</Grid>
```

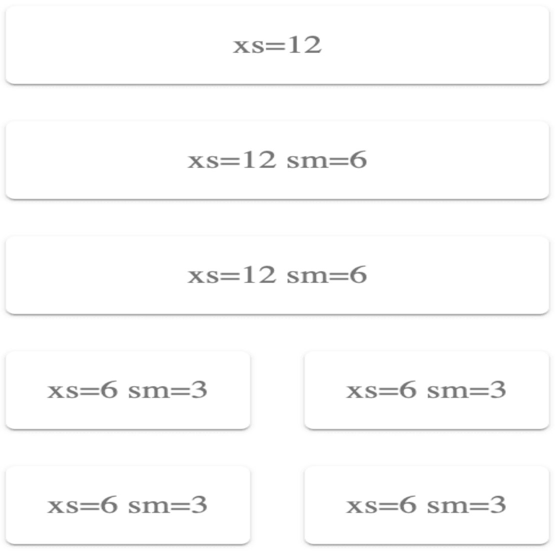
Material-UI's grid is based on a 12 column system meaning the screen at any given point has 12 columns of available space. When you change a value at ABCDE it will tell the item, "Hey, let's take up X amount of columns with this one grid item at this breakpoint!" This allows us to set up the grid in so many unique ways based on how we set the values for each breakpoint. Below I will list out some examples and their output!.

Examples:

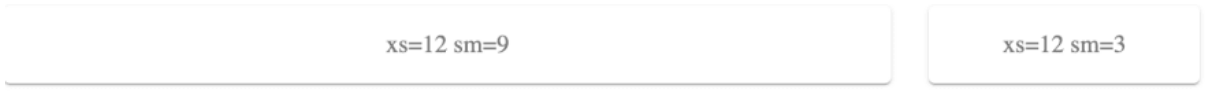


- In the above example we have the first row which is a grid item - xs(12) meaning it will always take all 12 of the columns at any screen size.
- The second row has two grid items - xs(12), sm(6) meaning at screen size small and larger (width: 600px+) each card will take up half the screen (2 cards x 6 columns each = 12 columns total) and at size x-small (width:0-600px) the card will take 12 columns, or the full screen.
- The third row has 4 grid items - xs(6), sm(3) meaning at screen size small and larger (width: 600px+) each card will take up 1/4th of the screen (4 cards x 3 columns each = 12 columns total) and at size x-small (width:0-600px) the card will take 6 columns, or half the screen.

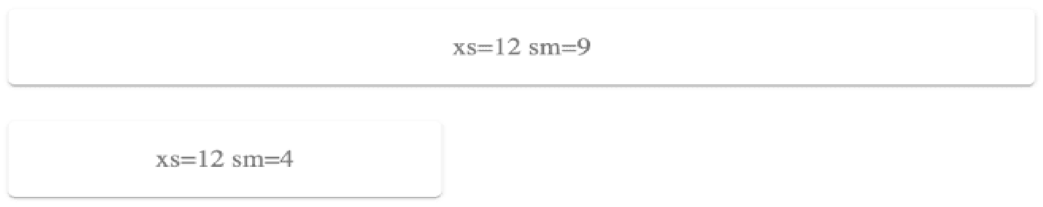
NOTE: Even if you decrease the width of the grid in the above image, the third row will never only have 1 card in it, because it is programmed to always have 2 cards for an x-small screen. An example of compressing the width to a super small screen is below.



You also do not need to have the 12 columns equally distributed between the cards in the row. If you had a grid item - xs(12), sm(9) and another grid item - xs(12), sm(3), they will be able to share the same row and take up 9 columns and 3 columns respectively, here is an image of this below.



Take note, if you distribute your cards without adding to 12 columns, the cards will not full take up the row and you'll be left with some extra space on the right-hand side like below.



Scalability & Maintenance

Key Considerations:

- Scalable to dynamic requirements.
- Modular in nature and open to multiple communication gateways.
- Single Global State management which can be traceable & testable.
- Faster first time load and consistent performance.
- Reactive to environmental changes including communication, infrastructure, bandwidth & devices.

State Management:

React State

Ideal for small apps with no nested child components and there no need of passing data between from child to parent.

Event for medium-large apps we can use to persist local component state for interactions like pagination, sorting, highlighting…

Context API

This is great when we want to store in a global place and share it across components. This way we can avoid passing loads of props to nested components.

Redux

Once your application grows neither React State nor Context API helps in ensuring maintainability & performance. Debugging becomes a nightmare as its difficult trace data flow and updates. Enter Redux which provides a pattern to manage application store / state.

Folder Structure & for API !

Here I am providing a link for a git repo which has a basic architecture of MVC framework. I request you to please go through the folder structure in **client** folder. And also segregating APIs in a separate folder gives a better opportunity for maintenance.

Why MaterialUI ?

Whether you go gaga for Material Design or gag looking at it, the "card" or "paper" concept with a focus on surfaces and edges continues to be a popular and broadly applied application style.

Material-UI, the React component library based on Google Material Design, allows for faster and easier stylized web development. With basic React framework familiarity, you can build a deliciously material app with Material-UI, and it's almost like cheating. Almost.

This MIT-licensed open source project is more than just parlour tricks though and can get deep quickly. But don't let me scare you! I recently built an app with Material-UI for the first time, and by the end I was delighted.

Here are 5 things I appreciated about Material-UI:

1. It's well documented: The official documentation is organized and easily navigable. The library's popularity means you have access to tons of code examples on the web if the documentation is confusing. You can also head over to StackOverflow for technical Q&A from Material-UI devs and the core team.

2. One-stop solution: It is a one package for replacing react-bootstrap, rmwc, styled-components and many more. This is the **foremost** reason.

3. Consistent appearance : Okay, this is kind of cheating because its a library, so of course the appearance is going to be consistent. BUT Material-UI is a HUGE library, and the benefit is you have some choices.
Aesthetic preferences for Material Design aside, your web project has a high chance of retaining similarity in appearance and functions all throughout.

4. Creative freedom: You don't have to have a consistent appearance if you don't want to! I know, I know - I just said that it creates a consistent appearance, but that's out of the box. As I hinted at earlier, there is actually quite a lot of depth to the Material-UI components and the developers encourage customization. Material-UI doesn't force Material Design style on you, it just offers it.

One delightful component was the ThemeProvider. Placed at the root of your app, you can change the colors, the typography and much more of all sub-Material-UI components! However, this is optional; Material-UI components come with a default theme. Code magic.

<u>5. Components work in isolation:</u> Material-UI components are self-supporting and will only inject the styles they need to display. They don't rely on any global stylesheets such as normalize.css! You only want to use that super cool progress spinner? Grab it! Live your best life!

Units (px, em, rem)

We have been conversant with using pixels for sizing in CSS, but why use em or rem? When building accessible websites, you need to consider inclusion. When you use px, you don't put user preferences at the forefront. When the user zooms in or change the browser font setting, websites need to adjust to fit the user's setting. Px do not scale but em and rem scales.

Sequel to this, setting the font size of the html element in percentage is recommended. Assuming the browser font size is set to 16px (i.e. the default), setting the font size of the html (root) element to 62.5% will default 1rem to 10px.

You may have used em and rem CSS units for a while but the difference between the two still appears vague. In this article, I will discuss the difference and when to use a particular unit to avoid building websites with unproportionate sizing.

- Use pre-processors to abstract the calculations using a px to rem converter function
- Use em only for sizing that needs to scale based on the font size of an element other than the html (root) element.
- Use rem unit for elements that scale depending on a user's browser font size settings. Use rem as the unit for most of your property value.
- For complex layout arrangement, use percentage (%).

The Recommended Setting

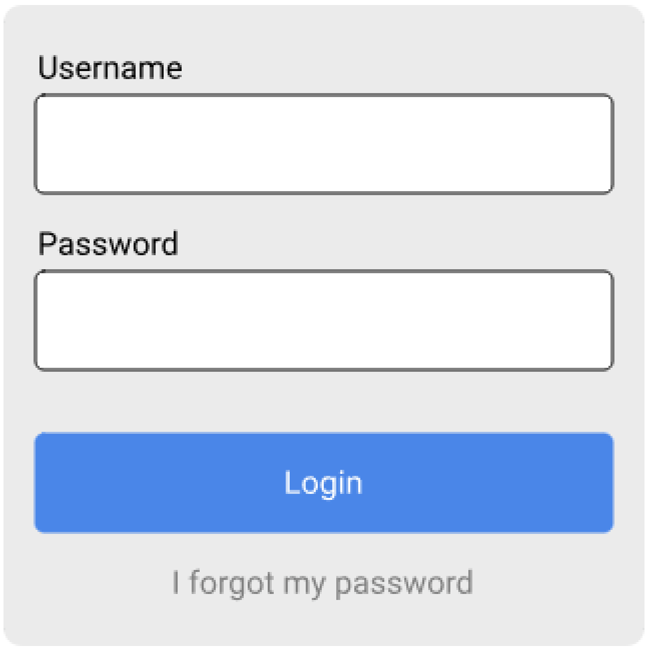
Setting the font size of the html element in percentage is recommended. This solves the problem.

Assuming the browser font size is set to 16px (i.e. the default), setting the font size of the root html element to 100% will default 1rem to 16px. This is still not the optimal solution. A better approach will be to use 62.5%. This will equate 1rem to 10px. Starting with this as the base simplifies the calculations.

Thinking in React

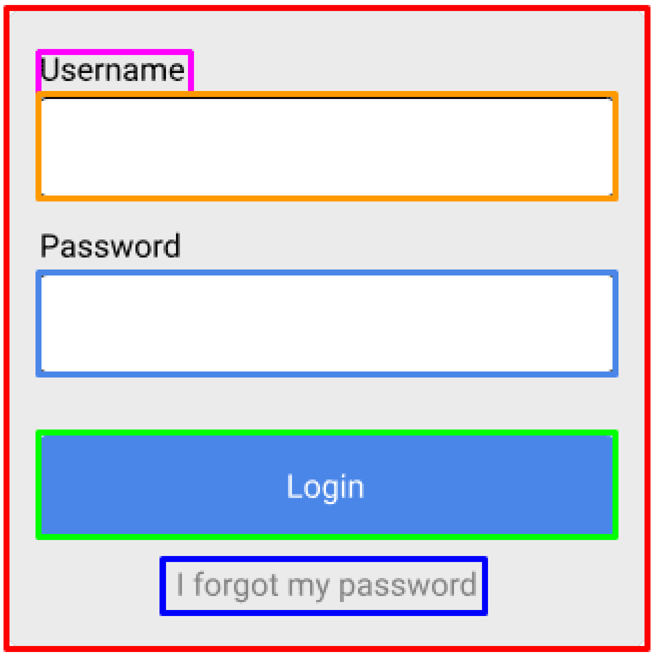
Start mocking

We should always start with a mock, either provided by a designer/design team in those big projects or made by ourselves if it's a small personal project. So, let's say we want the classic login experience:



Break mock into components

Now that we have the mock, we need to take a look at it and identify its parts:



Once identified, we should use clear names for every "component" (PascalCase by React convention):

- ***LoginForm (red):*** The whole login form.
- ***SubmitButton (green):*** The button to submit the "form".
- ***Label (pink):*** The form labels.
- ***Input (orange):*** The form inputs.
- ***PasswordInput (light blue):*** The form input with type password.

Build components

Now that we have identified the components, let's build them!

```
const Label = props => <label {...props} />;

const Input = props => <input {...props} />;

const PasswordInput = ({ type = "password", ...props }) => (
  <Input {...{ type, ...props }} />
);

const SubmitButton = ({ type = "submit", ...props }) => (
  <button {...{ type, ...props }} />
);

const LoginForm = props => <form {...props} />;
```

Notice, that we can even reuse Input inside PasswordInput.

Use components

Now that we have those components, we can use them to bring our mock to life. Let's call this wrapping component ***LoginContainer***:

```
const LoginContainer = () => (
  <LoginForm>
    <Label htmlFor="username">Username</Label>
    <Input id="username" name="username" />
    <Label htmlFor="password">Password</Label>
    <PasswordInput id="password" name="password" />
    <SubmitButton>Login</SubmitButton>
  </LoginForm>
);
```

This needs API interaction and event Handling, but first...

Early optimizations

While working on the components, we might detect optimizations such as every time we use an Input or PasswordInput component, we add a Label to it, so in order to keep DRY, let's create components to avoid repetition:

```
const FormInput = ({ name, id = `${name}-id`, title, ...props }) => (
  <>
    <Label htmlFor={id}>{title}</Label>
    <Input {...{ id, name, title, ...props }} />
  </>
);

const FormPasswordInput = ({ name, id = `${name}-id`, title, ...props }) => (
  <>
    <Label htmlFor={id}>{title}</Label>
    <PasswordInput {...{ id, name, title, ...props }} />
  </>
);
```

So now, our ***LoginContainer*** looks like this:

```
const LoginContainer = () => (
  <LoginForm>
    <FormInput name="username" title="Username" />
    <FormPasswordInput name="password" title="Password" />
    <SubmitButton>Login</SubmitButton>
  </LoginForm>
);
```

Adding state

State should generally be left for last, thinking and designing everything as stateless as possible, using props and events. It makes components easier to maintain, test and overall understand.

If you need state, it should be handled by either state containers (Redux, MobX, unistore, and so on) or a container/wrapper component. In our super simple login example, the place for the state could be LoginContainer itself, let's use React hooks for this:

```
const LoginContainer = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");

  const login = async event => {
    event.preventDefault();
    const response = await fetch("/api", {
      method: "POST",
      body: JSON.stringify({
        username,
        password,
      }),
    });
    // Here we could check response.status to login or show error
  };

  return (
    <LoginForm onSubmit={login}>
      <FormInput
        name="username"
        title="Username"
        onChange={event => setUsername(event.currentTarget.value)}
        value={username}
      />
      <FormPasswordInput
        name="password"
        title="Password"
        onChange={event => setPassword(event.currentTarget.value)}
        value={password}
    />
    <SubmitButton>Login</SubmitButton>
   </LoginForm>
  );
};
```

The approach of avoiding state is related to Functional Programming principles, but basically is to keep the components as pure as possible.

TL;DR

- Mock.
- Identify components.
- Build them.
- Use them (and optimize them when needed).
- Try to stay as stateless as possible. Add state only if needed.

Thinking in Hooks & Future Insights (PWA) : Coming SOON