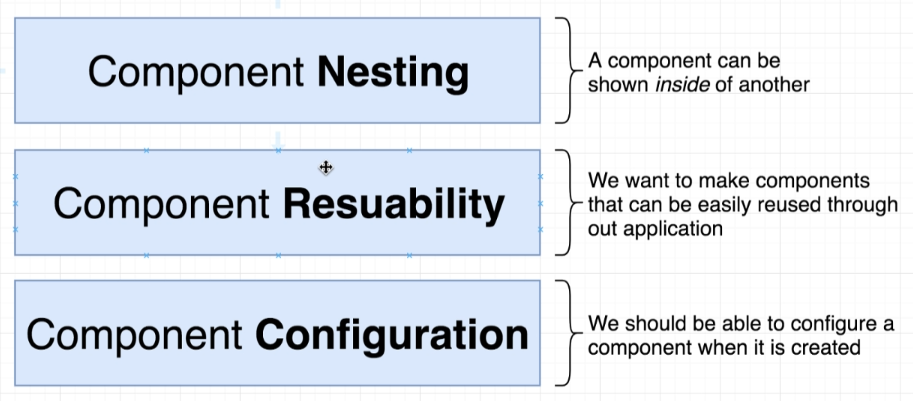
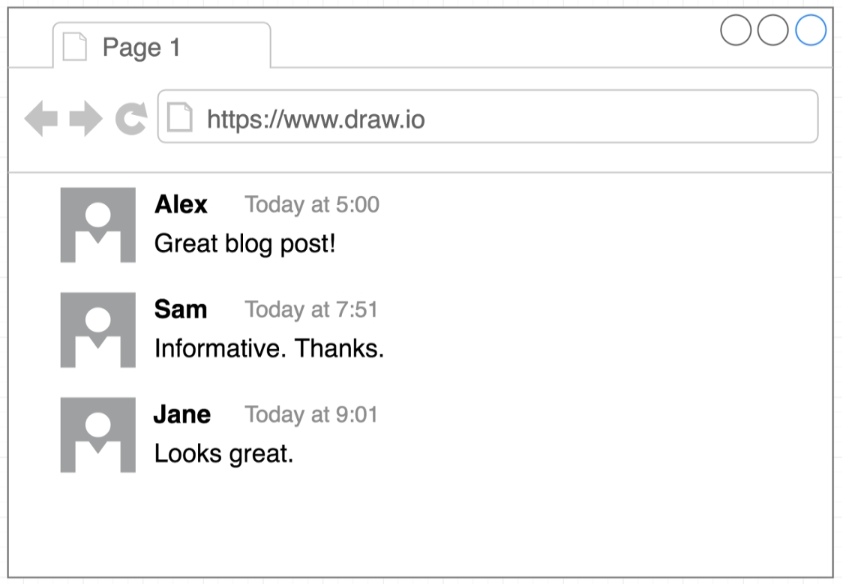
THREE TENETS OF COMPONENTS –



* For a real React component, we make multiple React Components and then nest them i.e. place them inside each other.
* Real-world applications have certain buttons and text fields. All such buttons are similar to each other (as every button inside of our application is going to have similar functionality). Rather than making a separate component/write different code for every button, we instead try to make reusable components.
* Most buttons look very similar but there’s tiny little pieces about every single button that need to be customized in very specific way. So, we need to make sure that different components, we should be able to customize them as and when they are created.



Application Mockup that we’re going to create.

Each one of these comments look similar to each other. The only difference is they tend to have different text.

SPECIFYING IMAGES IN JSX –

We’ll use a library known as **faker.js** that makes development of little applications much easier. (useful in own personal projects)

Anytime we want to use some code from another file or another open-source project/dependency, we must use an import statement.

import faker from ‘faker’;

faker.image.avatar()



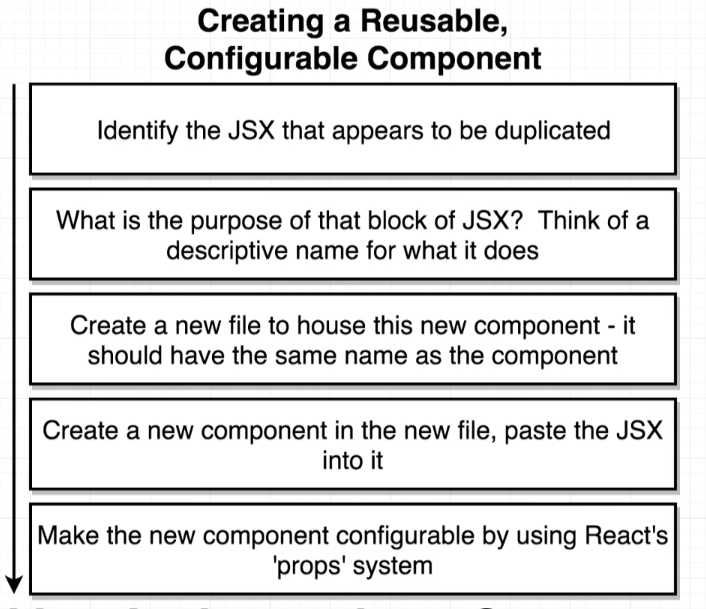
Go and find the second comment and edit the title.

By duplicating all the JSX in a single component, we end up in a pretty bad spot.

It becomes very challenging for us to make any update to this app component.

Now, we’ll refactor this into several components and customize each of the individual components.

We’ll now walk through series of steps to be followed to create a reusable component.



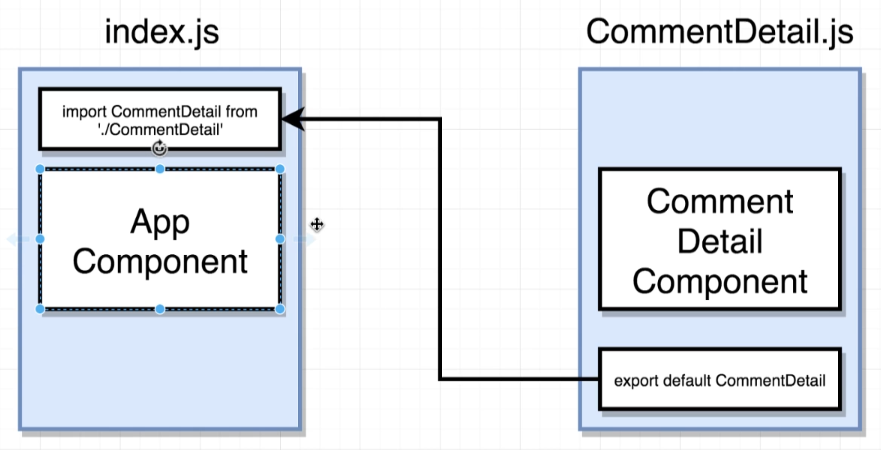
* In our code base, we can see that div with class name as comment is duplicated three times.
* We select the name of duplicated jsx as CommentDetail. Now we’ll put it into a brand new standalone component.
* Now we create js file having same name as the duplicated jsx name we decided. (CommentDetail.js)
* We create a new component inside this file and move the duplicated jsx once into it.
* Main thing we have to do is make this component configurable using **props**.

The JSX we’ve currently put into this new component has a hard-coded name, time and content. Anytime we want to show this comment detail on the screen as a component, we probably are going to want to customize it when we create the component. This is what the **prop** system is all about.

* Allows to configure how a component behaves/content that it shows.

Now, we want to render the CommentDetail component inside of our App.

1. We want to make sure this component is available inside of our App component. First off, we’re going to place an export statement inside of the CommentDetail. Then, to access the file, we’re going to add an import statement inside of index.js file.
   * **export default CommentDetail** – If any file wants to get access to the comment detail component, I got you covered. (makes this component available to every other file inside our project)
   * **import CommentDetail from ‘./CommentDetail’** – This is what forms a link between these two files and makes sure that CommentDetail is available inside index.js file. Once a link is formed, then we can use this component inside of our index.js file through a technique referred to as Component Nesting.



Note –

WebPack, the library that is doing the import stuff and tying all the different files together is going to automatically attempt to find files ending in js for us. Thus, we do not need to put actual file extension i.e. CommentDetail**.js** so long as it’s a JS file we’re trying to import.

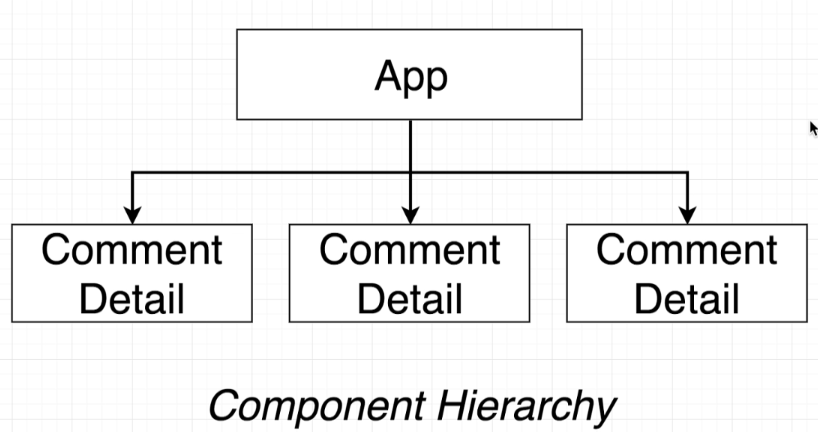
1. Now, we need to take the component we just imported, and use it inside our app. To do so, we treat comment detail component as though it is a **JSX tag** i.e. <CommentDetail />

Note –

To refer to JS in JSX, we need to put {…} braces around them. *Components* are the one **exception** to this rule. To show one component inside another, we treat them as though they are a JSX tag.

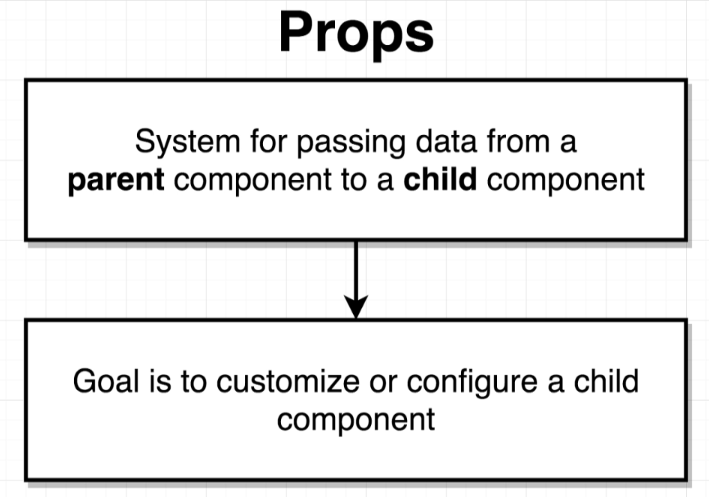
USING PROP SYSTEM TO CONFIGURE EACH COMPONENT –

The current issue with our application is that every single comment detail has the same name, time and content. We’re going to fix that using PROP system.

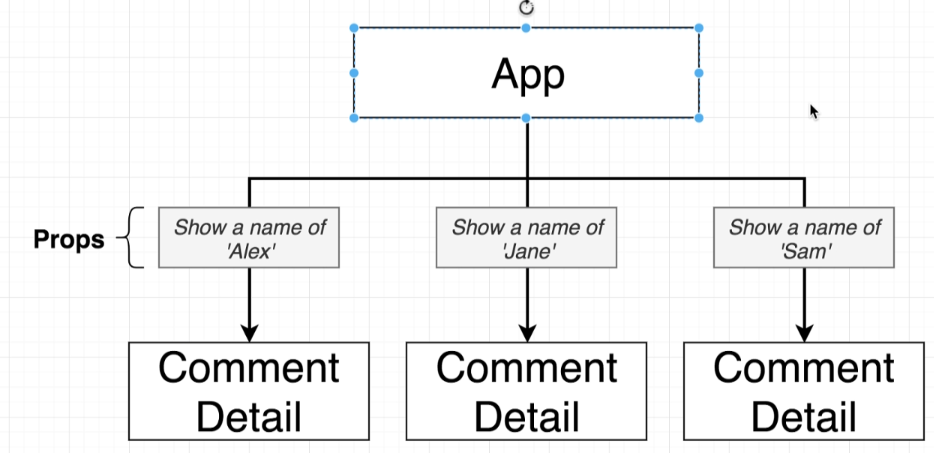


A diagram that tells us the different components that are currently being displayed inside of our application and relationship between those different components.

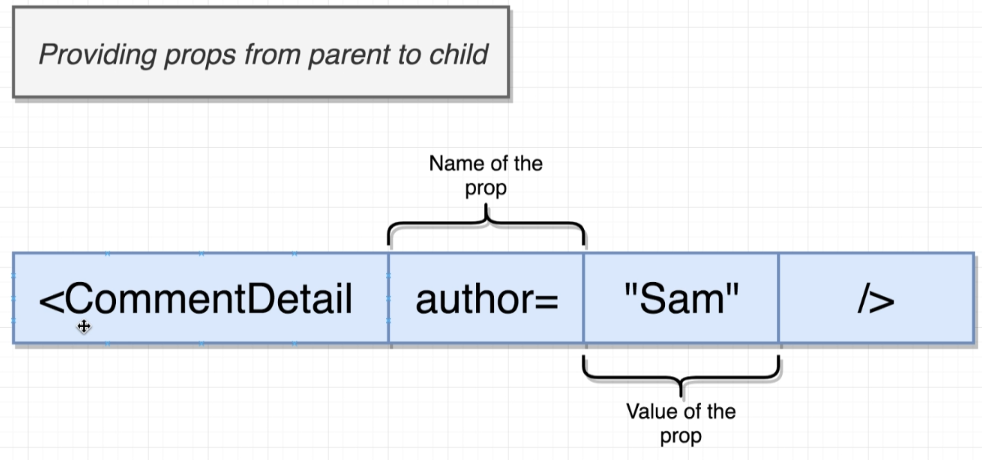
1 instance of App Component & 3 instances of Comment Detail Component.



The overall goal of the components is to show some content to the user or react to user-interaction. Thus the whole purpose of props is to customize these two actions.



When the App shows the components, at the same time, it’s going to pass in a little bit of configuration to each of those different components. These little configurations is referred to as props (tell the component to show name as Alex/Jane/Sam). Similarly, we can customize the time and content.



In the parent component, where we show an instance of Comment Detail component, to provide some information down to child, we’ll write out just about any other property/attribute as in JSX.

**Property\_Name = Value**

This property is then going to show up inside the Comment Detail. It’s totally up to us on how we want to name this property we’re passing.

Now, we must consume the information passed in the child component.

The prop is going to show up inside of an object that is provided as the first argument to the function of child component. 

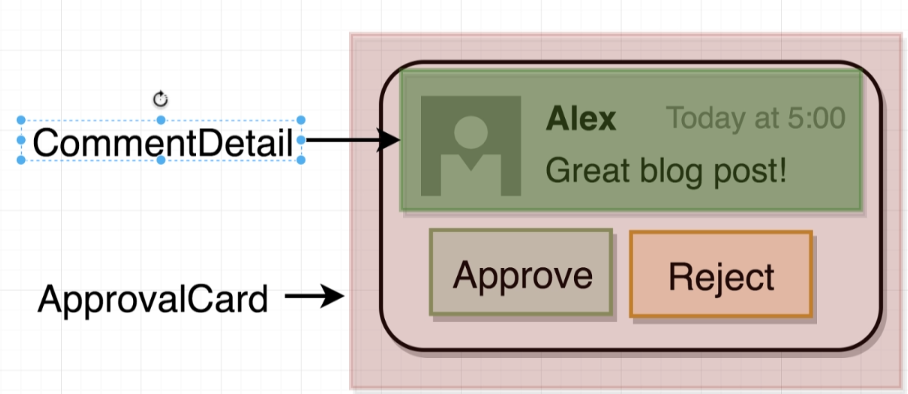
**console.log(props)** – props is an object having key value pair inside of it, where key is property name(author) and value is the value we’ve passed(Sam).

Now where we want to use the name of author, we’ll write {props.author}

Now, we want to reference the props.avatar and pass it to src. As we’re referencing a JS variable, we’ll continue to use **{…}.** (we’re not hardcoding the source so we do not use “…”)

COMPONENT REUSE –

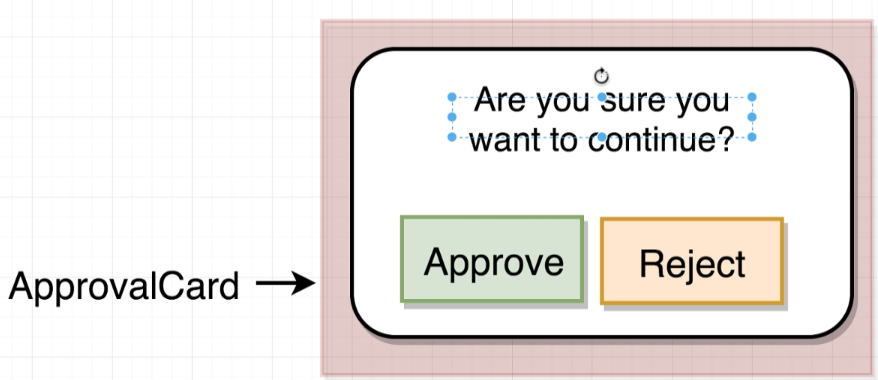
Now, we want to wrap all the different comments inside an approve/reject box for blog owner to control the comments. 



We’re going to create a new component – the ApprovalCard component to show component reusing.

Note –

When we’re going to make ApprovalCard component, we’re not going to hardcode it to show an instance of CommentDetail always. We might want to reuse the ApprovalCard component i.e. a box with approve and reject button, but having completely different content inside. Like,



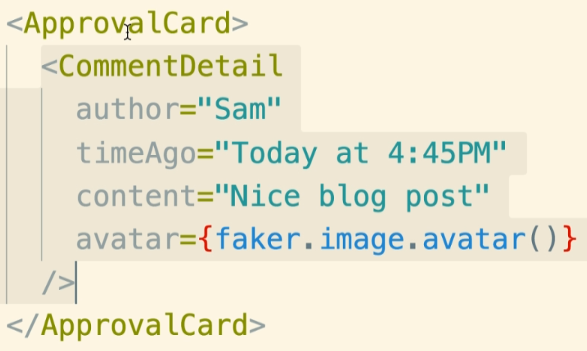
To do this, we’re going to make use of a little feature in the props system. At this point, we’ve only discussed props as being some property\_name = value, which we pass directly to a component inside of the component tag.

**But**, there’s another way we can communicate information into some child component.

Right now, we have Approval Card as a self-closing tag i.e. <ApprovalCard />. BUT we’ll use it as a normal HTML tag i.e.

<ApprovalCard> …. </ApprovalCard>

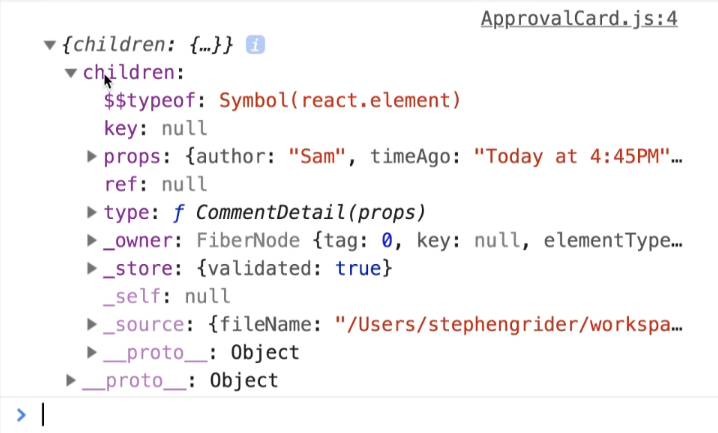
Now, to communicate the component down to Approval Card and tell the card that we want it to show Comment Detail component inside of it, we’re going to nest CommentDetail inside ApprovalCard.



Thus, we are now passing the CommentDetail component and passing it as a **prop** down into ApprovalCard, as we’re providing some configuration to the ApprovalCard component.

This component passed in now exists as props object in ApprovalCard component.

*console.log(props)* in ApprovalCard now shows –



Thus, when we pass one component to another using <Parent> <Child /> </Parent> (syntax), the child component is going to show up inside the parent component on the **props** object, and specifically on a property of props object called the *children* property.

Now, we can show the component inside of the body of our ApprovalCard.

{props.children}