

Presentation by Uplatz

Contact Us: <https://training.uplatz.com/>

Email: info@uplatz.com

Phone: +44 7836 212635

Table Of Contents:

- User interaction
- Input events
- Controlled vs. uncontrolled
- Styles
- Styling libraries

User interaction

- We'll be using the **onClick** prop quite a bit all throughout our apps quite a bit, so it's a good idea to be familiar with it.
- In our activity list header, we have a search icon that we haven't hooked up yet to show a search box.
- The interaction we want is to show a search `<input />` when our users click on the search icon.
- Recall that our **Header** component is implemented like this:

```
class Header extends React.Component {  
  render() {  
    return (  
      <div className="header">
```

```
<div className="menulcon">
  <div className="dashTop"></div>
  <div className="dashBottom"></div>
  <div className="circle"></div>
</div>
<span className="title">
  {this.props.title}
</span>
<input
  type="text"
  className="searchInput"
  placeholder="Search ..." />
<div className="fa fa-search searchIcon"></div>
```

```
</div>
```

```
)
```

```
}
```

```
}
```

- Let's update it a bit so that we can pass dynamic className prop to the <input /> element

```
class Header extends React.Component {
```

```
  render() {
```

```
    // Classes to add to the <input /> element
```

```
    let searchInputClasses = ["searchInput"];
```

```
    return (
```

```
      <div className="header">
```

```
        <div className="menuIcon">
```

```
          <div className="dashTop"></div>
```

```
<div className="dashBottom"></div>
  <div className="circle"></div>
</div>
<span className="title">
  {this.props.title}
</span>
<input
  type="text"
  className={searchInputClasses.join(' ')}
  placeholder="Search ..." />
<div className="fa fa-search searchIcon"></div>
</div>
)
```

```
}
```


```
}
```

- When the user clicks on the **<div className="fa fa-search searchIcon"></div>** element, we'll want to run a function to update the state of the component so the **searchInputClasses** object gets updated.
- Using the **onClick** handler, this is pretty simple.
- Let's make this component stateful (it needs to track if the search field should be showing or not).
- We can convert our component to be stateful using the **constructor()** function:

```
class Header extends React.Component {  
  constructor(props) {  
    super(props);
```

```
this.state = {  
    searchVisible: false  
}  
}  
// ...}
```

What is a constructor function?

- In JavaScript, the constructor function is a function that runs when an object is created.
- It returns a reference to the Object function that created the instance's prototype.
- In plain English, a constructor function is the function that runs when the JavaScript runtime creates a new object.
- We'll use the constructor method to set up instance variables on the object as it runs right when the  object is created.

- When using the ES6 class syntax to create an object, we have to call the `super()` method before any other method.
- Calling the `super()` function calls the parent class's `constructor()` function.
- We'll call it with the same arguments as the `constructor()` function of our class is called with.
- When the user clicks on the button, we'll want to update the state to say that the `searchVisible` flag gets updated.
- Since we'll want the user to be able to close/hide the `<input />` field after clicking on the search icon for a second time, we'll toggle the state rather than just set it to true.

Let's create this method to bind our click event:

```
class Header extends React.Component { // ...  
  showSearch() {  
    this.setState({  
      searchVisible: !this.state.searchVisible  
    })  
  }  
  // ...}
```

- Let's add an if statement to update searchInputClasses if this.state.searchVisible is true

```
class Header extends React.Component {  
  // ...  
  render() {
```

```
// ...  
  // Update the class array if the state is visible  
  if (this.state.searchVisible) {  
    searchInputClasses.push("active");  
  }  
  // ...  
}  
}
```

- Finally, we can attach a click handler (using the `onClick` prop) on the icon element to call our new `showSearch()` method.
- The entire updated source for our Header component looks like this:

```
class Header extends React.Component {  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      searchVisible: false  
    }  
  }  
  
  // toggle visibility when run on the state  
  showSearch() {  
    this.setState({  
      searchVisible: !this.state.searchVisible  
    })  
  }  
}
```

```
render() {  
    // Classes to add to the <input /> element  
    let searchInputClasses = ["searchInput"];  
    // Update the class array if the state is visible  
    if (this.state.searchVisible) {  
        searchInputClasses.push("active");  
    }  
    return (  
        <div className="header">  
            <div className="menuIcon">  
                <div className="dashTop"></div>  
                <div className="dashBottom"></div>  
                <div className="circle"></div>  
            </div>
```

```
<span className="title">
  {this.props.title}
</span>
```

```
<input
  type="text"
  className={searchInputClasses.join(' ')}
  placeholder="Search ..." />
```

```
{/* Adding an onClick handler to call the
showSearch button */}
```

```
<div
  onClick={this.showSearch.bind(this)}
  className="fa fa-search searchIcon"></div>
</div>
```

```
)  
}}
```

- Try clicking on the search icon and watch the input field appear and disappear (the animation effect is handled by CSS animations).

Input events

- Whenever we build a form in React, we'll use the input events offered by React.
- Most notably, we'll use the **onSubmit()** and **onChange()** props most often.
- Let's update our search box demo to capture the text inside the search field when it updates.
- Whenever an `<input />` field has the **onChange()** prop set, it will call the function every time the field changes.

- When we click on it and start typing, the function will be called.
- Using this prop, we can capture the value of the field in our state.
- Rather than updating our `<Header />` component, let's create a new child component to contain a `<form />` element.
- By moving the form-handling responsibilities to its own form, we can simplify the `<Header />` code and we can call up to the parent of the header when our user submits the form (this is a usual React pattern).
- Let's create a new component we'll call `SearchForm`.

- This new component is a stateful component as we'll need to hold on to the value of the search input (track it as it changes):

```
class SearchForm extends React.Component {  
  // ...  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      searchText: "  
    }  
  }  
  // ...  
}
```

- Now, we already have the HTML for the form written in the `<Header />` component, so let's grab that from our Header component and return it from our **`SearchForm.render()`** function:

```
class SearchForm extends React.Component {
```

```
// ...
```

```
render() {
```

```
  const { searchVisible } = this.props;
```

```
  let searchClasses = ["searchInput"];
```

```
  if (searchVisible) {
```

```
    searchClasses.push("active");
```

```
  }
```

```
return (
```

```
  <form>
```

```
    <input
```

```
type="search"
  className={searchClasses.join(" ")}
  placeholder="Search ..."
/>
</form>
);
}
}
```

- Now that we've moved some code from the Header component to the SearchForm, let's update its render method to incorporate the SearchForm

```
class Header extends React.Component {
  // ...
  render() {
    return (
```

```
<div className="header">
  <div className="menuIcon">
    <div className="dashTop"></div>
    <div className="dashBottom"></div>
    <div className="circle"></div>
  </div>
  <span className="title">{this.props.title}</span>
  <SearchForm />
  {/* Adding an onClick handler to call the showSearch
  button */}
  <div
    onClick={this.showSearch.bind(this)}
    className="fa fa-search searchIcon"
  ></div>
</div>
```

```
);  
}}
```

- Notice that we lost the styles on our `<input />` field.
- Since we no longer have the `searchVisible` state in our new component, we can't use it to style the `<input />` any longer.
- However, we can pass a prop from our Header component that tells the SearchForm to render the input as visible.
- Let's define the `searchVisible` prop (using `PropTypes`, of course) and update the render function to use the new prop value to show (or hide) the search `<input/>`.
- We'll also set a default value for the visibility of the field to be `false` (since our Header shows/hides it nicely):

```
class SearchForm extends React.Component {  
  // ...  
}  
  
SearchForm.propTypes = {  
  searchVisible: PropTypes.bool  
}  
  
SearchForm.defaultProps = {  
  searchVisible: false  
};
```

- In case you forgot to include PropTypes package in your page just add the following script tag in your page

```
<script src="https://unpkg.com/prop-  
types@15.6/prop-types.min.js"></script>
```

Finally, let's pass the searchVisible state value from Header as a prop to SearchForm

```
class Header extends React.Component {  
  render() {  
    return (  
      <div className="header">  
        <div className="menuIcon">  
          <div className="dashTop"></div>  
          <div className="dashBottom"></div>  
          <div className="circle"></div>  
        </div>  
        <span className="title">{this.props.title}<SearchForm  
searchVisible={this.state.searchVisible} />  
      </div>  
    );  
  }  
}
```

```
{/* Adding an onClick handler to call the showSearch  
button */}
```

```
<div
```

```
onClick={this.showSearch.bind(this)}
```

```
className="fa fa-search searchIcon"
```

```
></div>
```

```
</div>
```

```
);
```

```
}}
```

- Now we have our styles back on the `<input />` element, let's add the functionality for when the user types in the search box, we'll want to capture the value of the search field.

- We can achieve this workflow by attaching the onChange prop to the <input /> element and passing it a function to call every time the <input /> element is changed.

```
class SearchForm extends React.Component {
```

```
// ...
```

```
updateSearchInput(e) {
```

```
  const val = e.target.value;
```

```
  this.setState({
```

```
    searchText: val
```

```
  });
```

```
}
```

```
// ...
```

```
render() {
```

```
const { searchVisible } = this.state;
let searchClasses = ['searchInput']
if (searchVisible) {
  searchClasses.push('active')
} return (
  <form>
    <input
      type="search"
      className={searchClasses.join(" ")}
      onChange={this.updateSearchInput.bind(this)}
      placeholder="Search ..."
    />
  </form>
); } }
```

- When we type in the field, the `updateSearchInput()` function will be called.
- We'll keep track of the value of the form by updating the state. In the `updateSearchInput()` function, we can call directly to `this.setState()` to update the state of the component.
- The value is held on the event object's target as `event.target.value`.

```
class SearchForm extends React.Component {
```

```
// ...
```

```
  updateSearchInput(e) {  
    const val = e.target.value;  
    this.setState({  
      searchText: val  
    });  
  }
```

```
}// ...}
```

Controlled vs. uncontrolled

- We're creating what's known as an uncontrolled component as we're not setting the value of the `<input />` element.
- We can't provide any validation or post-processing on the input text value as it stands right now.
- If we want to validate the field or manipulate the value of the `<input />` component, we'll have to create what is called a controlled component, which really just means that we pass it a value using the `value` prop.
- A controlled component version's `render()` function would look like:

```
class SearchForm extends React.Component {  
  render() {  
    return (  
      <input  
        type="search"  
        value={this.state.searchText}  
        className={searchInputClasses}  
        onChange={this.updateSearchInput.bind(this)}  
        placeholder="Search ..." />  
    );  
  }  
}
```

- As of now, we have no way to actually submit the form, so our user's can't really search.

- Let's change this. We can capture the form submission by using the onSubmit prop on the <form /> element.
- Let's update the render() function to reflect this change.

```
class SearchForm extends React.Component {  
  // ...  
  submitForm(event) {  
    event.preventDefault();  
  }  
  // ...  
  render() {  
    const { searchVisible } = this.props;  
    let searchClasses = ['searchInput']  
    if (searchVisible) {
```

```
searchClasses.push('active')
  }return (
    <form onSubmit={this.submitForm.bind(this)}>
      <input
        type="search"
        className={searchClasses.join(' ')}
        onChange={this.updateSearchInput.bind(this)}
        placeholder="Search ..." />
      </form>
    );
  } }
```

- We immediately call `event.preventDefault()` on the `submitForm()` function.

- This stops the browser from bubbling the event up which would causes the default behavior of the entire page to reload (the default function when a browser submits a form).
- Now when we type into the `<input />` field and press enter, the `submitForm()` function gets called with the event object.
- So... great, we can submit the form and stuff, but when do we actually do the searching?
- For demonstration purposes right now, we'll pass the search text up the parent-child component chain so the Header can decide what to search.
- The SearchForm component certainly doesn't know what it's searching, so we'll have to pass the responsibility up the chain. We'll use this callback strategy quite a bit.

- In order to pass the search functionality up the chain, our **SearchForm** will need to accept a prop function to call when the form is submitted.
- Let's define a prop we'll call `onSubmit` that we can pass to our **SearchForm** component.
- Being good developers, we'll also add a default prop value and a `propTypes` for this **onSubmit** function.
- Since we'll want to make sure the `onSubmit()` is defined, we'll set the `onSubmit` prop to be a required prop:

```
class SearchForm extends React.Component {  
  // ...  
}
```

```
SearchForm.propTypes = {  
  onSubmit: PropTypes.func.isRequired,
```

```
searchVisible: PropTypes.bool
```

```
}
```

```
SearchForm.defaultProps = {
```

```
  onSubmit: () => {},
```

```
  searchVisible: false
```

```
}
```

- When the form is submitted, we can call this function directly from the props.
- Since we're keeping track of the search text in our state, we can call the function with the searchText value in the state so the onSubmit() function only gets the value and doesn't need to deal with an event.

```
class SearchForm extends React.Component {
```

```
  // ...
```

```
  submitForm(event) {
```

```
// prevent the form from reloading the entire page
event.preventDefault();
// call the callback with the search value
this.props.onSubmit(this.state.searchText);
}
}
```

- Now, when the user presses enter we can call this `onSubmit()` function passed in the props by our Header component.
- Let's add the `onSubmit` prop to the `SearchForm` in the Header component:

```
class Header extends React.Component {
  // ...
  render() {
    return (
```

```
<div className="header">
  <div className="menuIcon">
    <div className="dashTop"></div>
    <div className="dashBottom"></div>
    <div className="circle"></div>
  </div>
  <span className="title">{this.props.title}</span>
  <SearchForm searchVisible={this.state.searchVisible}
onSubmit={this.props.onSearch}/>
  {/* Adding an onClick handler to call the showSearch
button */}
  <div
    onClick={this.showSearch.bind(this)}
    className="fa fa-search searchIcon"
  ></div>
```

`</div>`

`);`

`}}`

- Now we have a search form component we can use and reuse across our app.
- Of course, we're not actually searching anything yet. Let's fix that and implement search.

Implementing search

- To implement search in our component, we'll want to pass up the search responsibility one more level from our Header component to a container component we'll call Panel.
- First things first, let's implement the same pattern of passing a callback to a parent component from within a child component from the Panel to the Header component.

- On the Header component, let's update the propTypes for a prop we'll define as a prop called onSearch:

```
class Header extends React.Component {  
  // ...}  
Header.propTypes = {  
  onSearch: PropTypes.func  
}
```

Here's our Panel component:

```
class Content extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      activities: data,  
    };  
  }  
}
```

```
render() {  
  const { activities } = this.state; // ES6 destructuring  
  return (  
    <div>  
      <Header  
        title="Github activity" />  
      <div className="content">  
        <div className="line" />  
        { /* Timeline item */}  
        {activities.map(activity => (  
          <ActivityItem key={activity.id}  
            activity={activity} />  
        ))}  
      </div>  
    </div>); } }
```

- In any case, our Panel component is essentially a copy of our Content component we previously built on day .
- Make sure to include the ActivityItem component in your page.
- Also don't forget to include Moment.js in your file as it's used by ActivityItem to format dates.

Add the following script tag in your page

```
<script  
src="https://unpkg.com/moment@2.24.0/min/moment.min.js"></script>
```

Notice that our virtual tree looks like this:

```
<Panel>  
  <Header>  
    <SearchForm></SearchForm>  
  </Header>
```


</Panel>

- When the `<SearchForm />` is updated, it will pass along its awareness of the search input's change to its parent, the `<Header />`, when it will pass along upwards to the `<Panel />` component.
- This method is very common in React apps and provides a good set of functional isolation for our components.
- Back in our Panel component, we'll pass a function to the Header as the `onSearch()` prop on the Header.
- What we're saying here is that when the search form has been submitted, we want the search form to call back to the header component which will then call to the Panel component to handle the search.

- Since the Header component doesn't control the content listing, the Panel component does, we have to pass the responsibility one more level up, as we're defining here.
- In order to actually handle the searching, we'll need to pass an `onSearch()` function to our Header component.
- Let's define an `onSearch()` function in our Panel component and pass it off to the Header props in the `render()` function:

```
class Panel extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      activities: data,  
    };  
  }  
}
```

```
handleSearch(val) {  
  // handle search here  
}  
  
render() {  
  const { activities } = this.state; // ES6 destructuring  
  return (  
    <div>  
      <Header  
        title="Github activity"  
        onSearch={this.handleSearch.bind(this)}  
      />  
      <div className="content">  
        <div className="line" />  
      </div>  
    </div>  
  );  
}
```

```

{ /* Timeline item */
    {activities.map(activity => (
        <ActivityItem key={activity.id} activity={activity}
    />
        )))
    </div>
</div>
);
}}

```

- All we did here was add a `handleSearch()` function and pass it to the header.
- Now when the user types in the search box, the `handleSearch()` function on our Panel component will be called.

- Let's update our `handleSearch` method to actually do the searching:

```
class Panel extends React.Component {  
  // ...  
  handleSearch(val) {  
    // resets the data if the search value is empty  
    if (val === '') {  
      this.setState({  
        activities: data  
      });  
    } else {  
      const { activities } = this.state;  
      const filtered = activities.filter(  
        a => a.actor && a.actor.login.match(val)  
      );
```

```
this.setState({  
  activities: filtered  
});  
}  
}  
// ...  
}
```

- All the `activities.filter()` function does is run the function passed in for every element and it filters out the values that return falsy values, keeping the ones that return truthy ones.
- Our search function simply looks for a match on the Github activity's `actor.login` (the Github user) to see if it `regex-matches` the `val` value.

- With the **handleSearch()** function updated, our search is complete.

Styles

- No application is complete without style.
- We'll look at the different methods we can use to style our components, from traditional CSS to inline styling.
- Through this point, we haven't touched the styling of our components beyond attaching Cascading **StyleSheet** (CSS) class names to components.
- Let's look at a few of the different ways we can style a component.

Cascading StyleSheets (CSS)

- Inline styles
- Styling libraries

CSS

- Using CSS to style our web applications is a practice we're already familiar with and is nothing new.
- If you've ever written a web application before, you most likely have used/written CSS.
- In short, CSS is a way for us to add style to a DOM component outside of the actual markup itself.
- Using CSS alongside React isn't novel. We'll use CSS in React just like we use CSS when not using React.
- We'll assign ids/classes to components and use CSS selectors to target those elements on the page and let the browser handle the styling.
- As an example, let's style our Header component we've been working with a bit.

- Let's say we wanted to turn the header color orange using CSS.
- We can easily handle this by adding a stylesheet to our page and targeting the CSS class of .header in a CSS class.
- Recall, the render function of our Header component currently looks like this:

```
class Header extends React.Component {  
  render() {  
    return (  
      <div className="header">  
        <div className="menuIcon">  
          <div className="dashTop"></div>  
          <div className="dashBottom"></div>  
          <div className="circle"></div>
```

```
</div>
<span className="title">
  {this.props.title}
</span>
<input
  type="text"
  className="searchInput"
  placeholder="Search ..." />

  <div className="fa fa-search searchIcon"></div>
</div>
) }
```

- We can target the header by defining the styles for a .header class in a regular css file.

- As per-usual, we'll need to make sure we use a `<link />` tag to include the CSS class in our HTML page.
- Supposing we define our styles in a file called `styles.css` in the same directory as the `index.html` file, this `<link />` tag will look like the following:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

Let's fill in the styles for the Header class names:

```
.demo .notificationsFrame .header {  
    background: rgba(251, 202, 43, 1);  
}  
.demo .notificationsFrame .header .searchIcon,  
.demo .notificationsFrame .header .title {  
    color: #333333;  
}
```

```
demo .notificationsFrame .header .menulcon  
.dashTop,  
.demo .notificationsFrame .header .menulcon  
.dashBottom,  
.demo .notificationsFrame .header .menulcon .circle {  
  background-color: #333333;  
}
```

- One of the most common complaints about CSS in the first place is the cascading feature itself.
- The way CSS works is that it cascades (hence the name) parent styles to its children.
- This is often a cause for bugs as classes often have common names and it's easy to overwrite class styles for non-specific classes.

- Using our example, the class name of `.header` isn't very specific.
- Not only could the page itself have a header, but content boxes on the page might, articles, even ads we place on the page might have a class name of `.header`.
- One way we can avoid this problem is to use something like CSS modules to define custom, very unique CSS class names for us.
- There is nothing magical about CSS modules other than it forces our build-tool to define custom CSS class names for us so we can work with less unique names. We'll look into using CSS modules a bit later in our workflow.

- React provides a not-so-new method for avoiding this problem entirely by allowing us to define styles inline along with our JSX.

Inline styles

- Adding styles to our actual components not only allow us to define the styles inside our components, but allow us to dynamically define styles based upon different states of the app.
- React gives us a way to define styles using a JavaScript object rather than a separate CSS file.
- Let's take our Header component one more time and instead of using css classes to define the style, let's move it to inline styles.
- Defining styles inside a component is easy using the style prop.

- All DOM elements inside React accept a style property, which is expected to be an object with camel-cased keys defining a style name and values which map to their value.
- For example, to add a color style to a `<div />` element in JSX, this might look like:

`<div style={{ color: 'blue' }}>`

This text will have the color blue

`</div>`

- Notice that we defined the styles with two braces surrounding it.
- As we are passing a JavaScript object within a template tag, the inner brace is the JS object and the outer is the template tag.

- Another example to possibly make this clearer would be to pass a JavaScript object defined outside of the JSX, i.e.

```
render() {  
  const divStyle = { color: 'blue' }  
  return (<div style={divStyle}>  
    This text will have the color blue  
    </div>);  
}
```

- In any case, as these are JS-defined styles, so we can't use just any ole' css style name (as background-color would be invalid in JavaScript).
- Instead, React requires us to camel-case the style name.

- camelCase is writing compound words using a capital letter for every word with a capital letter except for the first word, like backgroundColor and linearGradient.
- To update our header component to use these styles instead of depending on a CSS class definition, we can add the className prop along with a style prop:

```
class Header extends React.Component {  
  render() {  
    const wrapperStyle = {  
      backgroundColor: "rgba(251, 202, 43, 1)"  
    };const titleStyle = {  
      color: "#111111"  
    };  
  };  
}
```

```
const menuColor = {  
  backgroundColor: "#111111"  
};  
return (  
  <div style={wrapperStyle} className="header">  
    <div className="menuIcon">  
      <div className="dashTop"  
style={menuColor}></div>  
      <div className="dashBottom"  
style={menuColor}></div>  
      <div className="circle"  
style={menuColor}></div>  
    </div>  
  </div>  
)
```

```
<span style={titleStyle} className="title">
  {this.props.title}
</span>
```

```
<input
  type="text"
  className="searchInput"
  placeholder="Search ..."
/>
```

```
<div style={titleStyle} className="fa fa-search
searchIcon"></div>
</div>
);
}}
```

Styling libraries

- The React community is a pretty vibrant place (which is one of the reasons it is a fantastic library to work with).
- There are a lot of styling libraries we can use to help us build our styles, such as Radium by Formidable labs.
- Most of these libraries are based upon workflows defined by React developers through working with React.
- Radium allows us to define common styles outside of the React component itself, it auto-vendor prefixes, supports media queries (like :hover and :active), simplifies inline styling, and kind of a lot more.

- We won't dive into Radium in this post as it's more outside the scope of this series, but knowing other libraries are good to be aware of, especially if you're looking to extend the definitions of your inline styles.
- Now that we know how to style our components, we can make some good looking ones without too much trouble.

Summary:



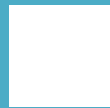
User interaction



Input events



Controlled vs. uncontrolled



Styles



Styling libraries

Thank You.....

If you have any queries please write to info@uplatz.com".