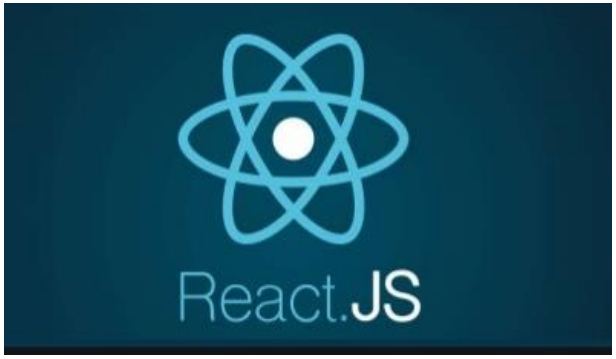


Presentation by Uplatz

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

Email: info@uplatz.com

Phone:+44 7836 212635

Uplatz

Table Of Contents:

Controlled Components

- Controlled form components are defined with a value property.
- The value of controlled inputs is managed byReact, user inputs will not have any direct influence on the rendered input.
- Instead, a change to the value property needs to reflect this change.

```
class Form extends React.Component {
 constructor(props) {
 super(props);
this.onChange = this.onChange.bind(this);
this.state = {
 name: ''
 };
```

```
}onChange(e) {
this.setState({
name: e.target.value
}); }
render() {
return (
<div>
<label for='name-input'>Name: </label>
<input
id='name-input'
onChange={this.onChange}
value={this.state.name} />
</div>)
}}
```

- The above example demonstrates how the value property defines the current value of the input and the **onChange**
- event handler updates the component's state with the user's input.
- Form inputs should be defined as controlled components where possible.
- This ensures that the component state and the input value is in sync at all times, even if the value is changed by a trigger other than a user input.

Uncontrolled Components

- Uncontrolled components are inputs that do not have a value property.
- In opposite to controlled components, it is the application's responsibility to keep the component state and the input value in sync.

```
class Form extends React.Component {
 constructor(props) {
 super(props);
this.onChange = this.onChange.bind(this);
this.state = {
 name: 'John'};
 }
 onChange(e) {
 this.setState({
 name: e.target.value
 });}
 render() {
 return (
```

```
<div>
 <label for='name-input'>Name: </label>
 <input
 id='name-input'
 onChange={this.onChange}
 defaultValue={this.state.name} />
 </div>
 )} }
```

- Here, the component's state is updated via the onChange event handler, just as for controlled components.
- However, instead of a value property, a defaultValue property is supplied.
- This determines the initial value of the input during the first render.

- Any subsequent changes to the component's state are not automatically reflected by the input value; If this is required, a controlled component should be used instead.

Repeating elements

- We've already seen this before where we've iterated over a list of objects and render multiple components on screen.
- Before we add too much complexity in our app with loading external data, today we'll take a quick peek at how to repeat components/elements in our app.
- Since JSX is seen as plain JavaScript by the browser, we can use any ole' JavaScript inside the template tags in JSX. We've already seen this in action.

As a quick demo:

```
const a = 10;
const ShowA = () => <div>{a}</div>;
const MultipleA = () => <div>{a * a}</div>;
const App = props => {
  return (
    <div className="app">
      <ShowA />
      <MultipleA />
    </div>
  );
};
```

- Notice the things inside of the template tags {} look like simple JavaScript.
- That's because it is just JavaScript.

- This feature allows us to use (most) native features of JavaScript inside our template tags including native iterators, such as map and forEach.
- Let's see what we mean here.
- Let's convert the previous example's a value from a single integer to a list of integers:

```
const a = [1, 10, 100, 1000, 10000];
```

- We can map over the a variable here inside our components and return a list of React components that will build the virtual DOM for us.

```
const a = [1, 10, 100, 1000, 10000];
const Repeater = () => {
  return (
    <ul>
      {a.map(i => {
```

```
return <li>{i}</li>;
      })}
    </ul>
  );
};
```

What is the map() function?

- The map function is a native JavaScript built-in function on the array.
- It accepts a function to be run on each element of the array, so the function above will be run four times with the value of i starting as 1 and then it will run it again for the second value where i will be set as 10 and so on and so forth.

- Let's open up our src/App.js file and replace the content of the App component with this source.
- Cleaning up a few unused variables and your src/App.js should look similar to this:

```
import React from "react";
const a = [1, 10, 100, 1000, 10000];
const App = props => {
  return (
    <ul>
      {a.map(i => {
        return <li>{i}</li>;
      })}
    </ul>
  );
};
```

export default App;

- Starting the app again with the command generated by the create-react-app command: npm start, we can see the app is working in the browser!
- However, if we open the developer console, we'll see we have an error printed out.
- This error is caused by the fact that React doesn't know how to keep track of the individual components in our list as each one just looks like a component.
- For performance reasons, React uses the virtual DOM to attempt to limit the number of DOM elements that need to be updated when it rerenders the view.

- That is if nothing has changed, React won't make the browser update anything to save on work.
- This feature is really fantastic for building web applications, but sometimes we have to help React out by providing unique identifiers for nodes.
- Mapping over a list and rendering components in the map is one of those times.
- React expects us to uniquely identify components by using a special prop:
- the key prop for each element of the list. The key prop can be anything we want, but it must be unique for that element.
- In our example, we can use the i variable in the map as no other element in the array has the same value.
- Let's update our mapping to set the key:

```
const App = props => {
  return (
    <ul>
      {a.map(i => {
        return <li key={i}>{i}</li>;
      })}
    </ul>
  ); };
```

Children

- We talked about building a parent-child relationship a bit earlier this week, but let's dive a bit more into detail about how we get access to the children inside a parent component and how we can render them.

- we built a <Formatter /> component to handle date formatting within the Clock component to give our users flexibility with their own custom clock rendering.
- Recall that the implementation we created is actually pretty ugly and relatively complex.

```
const Formatter = props => {
  let children = props.format.split("").map((e, idx) => {
    if (e === "h") {
      return <Hour key={idx} {...props} />;
    } else if (e === "m") {
      return <Minute key={idx} {...props} />;
    } else if (e === "s") {
      return <Second key={idx} {...props} />;
    }
```

```
else if (e === "p") {
      return <Ampm key={idx} {...props} />;
    } else if (e === " ") {
      return <span key={idx}> </span>;
    } else {
      return <Separator key={idx} {...props} />;
    }
  });return <span>{children}</span>;
};
```

- We can use the **React.Children** object to map over a list of React objects and let React do this heavy-lifting.
- The result of this is a much cleaner Formatter component (not perfect, but functional):

```
const Formatter = props => {
  let children = props.format.split("").map(e => {
    if (e == "h") {
      return <Hour />;
    } else if (e == "m") {
      return <Minute />;
    } else if (e == "s") {
      return <Second />;
    } else if (e == "p") {
      return <Ampm />;
    } else if (e == " ") {
      return <span> </span>;
    } else {
      return <Separator />;}  });
```

```
return (
    <span>
      {React.Children.map(children, c =>
React.cloneElement(c, props))}
    </span>
  );
};
```

React.cloneElement

- We have yet to talk about the React.cloneElement() function, so let's look at it briefly here.
- Remember WWWWWAAAAAYYYYY back on day 2 we looked at how the browser sees JSX? It turns it into JavaScript that looks similar to:

```
React.createElement("div", null,
 React.createElement("img", {src: "profile.jpg", alt:
"Profile photo"}),
 React.createElement("h1", null, "Welcome back Ari")
);
```

- Rather than creating a new component instance (if we already have one), sometimes we'll want to copy it or add custom props/children to the component so we can retain the same props it was created with.
- We can use React.cloneElement() to handle this for us.
- The React.cloneElement() has the same API as the React.createElement() function where the arguments are:

- The ReactElement we want to clone
- Any props we want to add to the instance
- Any children we want it to have.
- In our Formatter example, we're creating a copy of all the children in the list (the <Hour />, <Minute />, etc.
- components) and passing them the props object as their props.
- The React.Children object provides some nice utility functions for dealing with children.
- Our Formatter example above uses the map function to iterate through the children and clone each one in the list.
- It creates a key (if necessary) for each one, freeing us from having to manage the uniqueness ourselves.

- Let's use the **React.Children.map**() function to update our App component:

```
const App = props => {
  return (
    <ul>
      {React.Children.map(a, i => (
        <li>{i}</li>
      ))}
    </ul>
  );
};
```

- There are several other really useful methods in the React.Children object available to us.

- We'll mostly use the React.Children.map() function, but it's good to know about the other ones available to us.
- Up through this point, we've only dealt with local data, not really focusing on remote data (although we did briefly mention it when building our activity feed component).

React Boilerplate [React +Babel + Webpack]:

react-starter project:

About this Project

- This is simple boilerplate project. This post will guide you to set up the environment for ReactJs + Webpack + Bable.

Lets get Started

- we will need node package manager for fire up express server and manage dependencies throughout the project.
- If you are new to node package manager, you can check here.

Note : Installing node package manager is require here.

- Create a folder with suitable name and navigate into it from terminal or by GUI.
- Then go to terminal and type npm init this will create a package.json file, Nothing scary , it will ask you few question like name of your project ,version, description, entry point, git repository, author, license etc.
- Here entry point is important because node will
- initially look for it when you run the project.

- At the end it will ask you to verify the information you provide.
- You can type yes or modify it. Well that's it , our **package.json** file is ready.
- Express server setup run **npm install express@4 --save.** This is all the dependencies we needed for this project.
- Here save flag is important, without it package.js file will not be updated.
- Main task of package.json is to store list of
- dependencies.
- It will add express version 4.
- Your package.json will look like "dependencies": { "express":
- "^4.13.4", ... },

- After complete download you can see there is node_modules folder and sub folder of our dependencies.
- Now on the root of project create new file server.js file. Now we are setting express server.
- I am going to past all the code and explain it later.

```
var express = require('express');
// Create our app
var app = express();
app.use(express.static('public'));
app.listen(3000, function () {
 console.log('Express server is using port:3000');
});
```

var express = require('express'); this will gave you the access of entire express api.

var app = express(); will call express library as function.
app.use(); let the add the functionality to your expressapplication.

app.use(express.static('public')); will specify the folder name that will be expose in our web server.

app.listen(port, function(){}) will here our port will be 3000 and function we are calling will verify that out web server

is running properly.

That's it express server is set up.

- Now go to our project and create a new folder public and create index.html file.
- index.html is the default file for you application and Express server will look for this file.

The index.html is simple html file which looks like

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="UTF-8"/>
</head>
<body>
 <h1>hello World</h1>
</body>
</html>
```

- And go to the project path through the terminal and type node server.js.
- Then you will see * console.log('Express server is using port:3000');*.

- Go to the browser and type http://localhost:3000 in nav bar you will see hello World.
- Now go inside the public folder and create a new file app.jsx. JSX is a preprocessor step that adds XML syntax to your JavaScript.
- You can definitely use React without JSX but JSX makes React a lot more elegant.

Here is the sample

code for app.jsx

```
ReactDOM.render(
 <h1>Hello World!!!</h1>,
 document.getElementById('app')
);
```

- Now go to index.html and modify the code , it should looks like this

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="UTF-8"/>
 <script
src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23
 /browser.min.js"></script>
 <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.
7/react.js">
 </script>
 <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.
7/react-dom.js"> </script>
```

```
</head>
<body>
 <div id="app"></div>
 <script type="text/babel" src="app.jsx"></script>
</body>
</html>
```

- With this in place you are all done, I hope you find it simple.

Setting up the project

- You need Node Package Manager to install the project dependencies.
- Download node for your operating system

from Nodejs.org. Node Package Manager comes with node.

- You can also use Node Version Manager to better manage your node and npm versions.
- It is great for testing your project on different node versions.
- However, it is not recommended for production environment.
- Once you have installed node on your system, go ahead and install some essential packages to blast off your first React project using Babel and Webpack
- Before we actually start hitting commands in the terminal.
- Take a look at what Babel and Webpack are used for.
- You can start your project by running npm init in your terminal.

Follow the initial setup.

After that, run following commands in your terminalDependencies:

npm install react react-dom --save

Dev Dependecies:

npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0

webpack webpack-dev-server react-hot-loader --save-dev

Optional Dev Dependencies:

npm install eslint eslint-plugin-react babel-eslint --save-dev

You may refer to this sample package.json

Create .babelrc in your project root with following

contents:

```
{
 "presets": ["es2015", "stage-0", "react"]
}
```

Optionally create .eslintrc in your project root with following contents:

```
{
 "ecmaFeatures": {
 "jsx": true,
 "modules": true
 },
 "env": {
 "browser": true,
 "node": true
 },
```

```
"parser": "babel-eslint",
 "rules": {
 "quotes": [2, "single"],
 "strict": [2, "never"],
 },
 "plugins": [
 "react"
 ]
}
```

- Create a .gitignore file to prevent uploading generated files to your git repo.

```
node_modules
npm-debug.log
.DS_Store
dist
```

- Create webpack.config.js file with following minimum contents.

```
var path = require('path');
var webpack = require('webpack');
module.exports = {
devtool: 'eval',
entry: [
'webpack-dev-server/client?http://localhost:3000',
'webpack/hot/only-dev-server',
'./src/index'
],
output: {
path: path.join(__dirname, 'dist'),
filename: 'bundle.js',
```

```
publicPath: '/static/'
 },
 plugins: [
 new webpack.HotModuleReplacementPlugin()
 ],
 module: {
 loaders: [{
 test: /\.js$/,
 loaders: ['react-hot', 'babel'],
 include: path.join(__dirname, 'src')
 }]
 }};
```

- And finally, create a sever.js file to be able to run npm start, with following contents:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');
new WebpackDevServer(webpack(config), {
 publicPath: config.output.publicPath,
 hot: true,
 historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
 if (err) {
 return console.log(err);
 }
 console.log('Serving your awesome project at http://localhost:3000/');
});
```

- Create src/app.js file to see your React project do something.

```
import React, { Component } from 'react';
export default class App extends Component {
 render() {
 return (
 <h1>Hello, world.</h1>
 );
 }
}
```

- Run node server.js or npm start in the terminal, if you have defined what start stands for in your package.json

Fetching Remote Data

Querying for remote data

- The normal browser workflow is actually a synchronous one.
- When the browser receives html, it parses the string of html content and converts it into a tree object (this is what we often refer to as the DOM object/DOM tree).
- When the browser parses the DOM tree, as it encounters remote files (such as <link /> and <script /> tags), the browser will request these files (in parallel), but will execute them synchronously (so as to maintain their order they are listed in the source).

- What if we want to get some data from off-site? We'll make requests for data that's not available at launch time to populate data in our app.
- However, it's not necessarily that easy to do because of the asynchronous nature of external API requests.
- Essentially, what this means is that we'll have to handle with JavaScript code after an unknown period of time as well actually make an HTTP request.
- Luckily for us, other people have dealt with this problem for a long time and we now have some pretty nice ways of handling it.
- Starting with handling how we'll be making an HTTP request, we'll use a library (called fetch, which is also a web standard, hopefully) to make the http requesting easier to deal with.

Fetch

- In order to use fetch, we'll need to install the library in our app we previously created.
- Let's open up a terminal window again and use npm to install the whatwg-fetch library (an implementation of fetch). In the same directory where we created our application, let's call:

npm install --save whatwg-fetch

- With the library installed, we can make a request to an off-site server.
- In order to get access to the fetch library, we'll need to import the package in our script.
- Let's update the top few lines of our src/App.js file adding the second line:

import React, { Component } from "react";

```
import "whatwg-fetch";
// ...
```

- The whatwg-fetch object is unique in that it is one of the few libraries that we'll use which attaches it's export on the global object (in the case of the browser, this object is window).
- Unlike the react library, we don't need to get a handle on it's export as the library makes it available on the global object.
- With the whatwg-fetch library included in our project, we can make a request using the fetch() api.
- However, before we can actually start using the fetch() api, we'll need to understand what Promises are and how they work to deal with the asynchronous we discussed in the introduction

Data-Driven

- We broke down our demo into components and ended up building three separate components with static JSX templates.
- It's not very convenient to have to update our component's template everytime we have a change in our website's data.
- Instead, let's give the components data to use to display.
- Let's start with the <**Header** /> component.
- As it stands right now, the <**Header** /> component only shows the title of the element as Timeline.
- It's a nice element and it would be nice to be able to reuse it in other parts of our page, but the title of Timeline doesn't make sense for every use.

- Let's tell React that we want to be able to set the title to something else.

Introducing props

- React allows us to send data to a component in the same syntax as HTML, using attributes or properties on a component.
- This is akin to passing the src attribute to an image tag. We can think about the property of the <img /> tag as a prop we're setting on a component called img.
- We can access these properties inside a component as this.props. Let's see props in action.

- Recall, we defined the <Header /> component as:

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

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

- We can pass in our title as a prop as an attribute on the <Header /> by updating the usage of the component setting the attribute called title to some string, like so:

```
<Header title="Timeline" />
```

- Inside of our component, we can access this title prop from the this.props property in the Header class.
- Instead of setting the title statically as Timeline in the template, we can replace it with the property passed in.

```
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've also updated the code slightly to get closer to what our final <Header /> code will look like, including adding a searchIcon and a few elements to style the menuIcon.

- Now our <Header /> component will display the string we pass in as the title when we call the component.
- For instance, calling our <Header /> component four times like so:

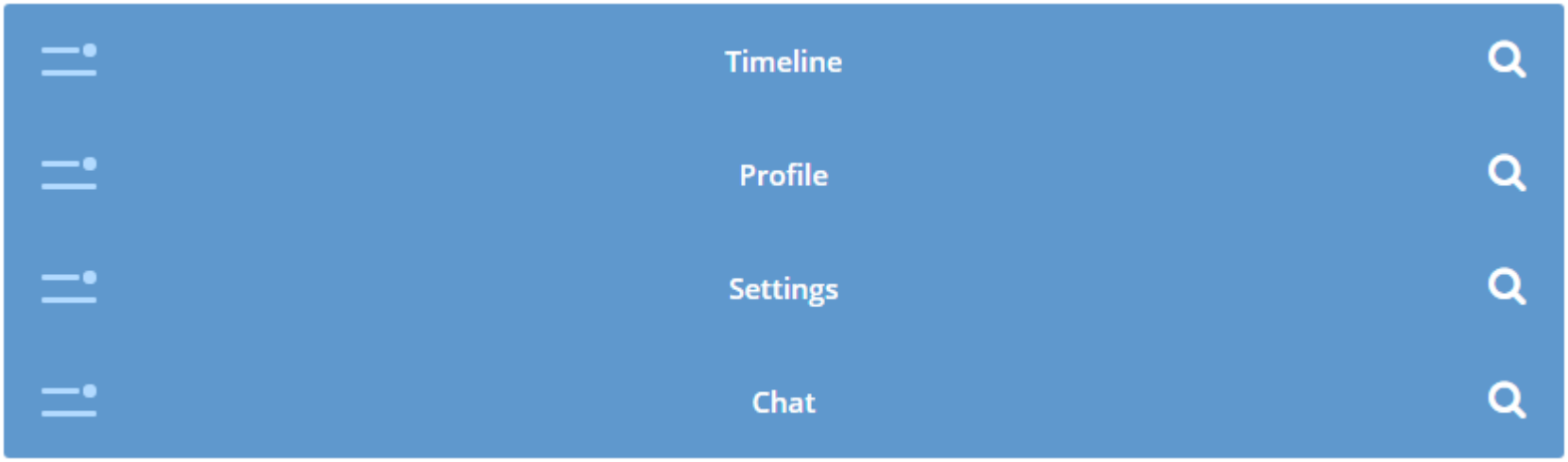
<Header title="Timeline" />

<Header title="Profile" />

<Header title="Settings" />

<Header title="Chat" />

Results in four <Header /> components to mount like so:



We can pass in more than just strings in a component.

- We can pass in numbers, strings, all sorts of objects, and even functions! We'll talk more about how to define these different properties so we can build a component api later.
- Instead of statically setting the content and date let's take the Content component and set the timeline content by a data variable instead of by text.

- Just like we can do with HTML components, we can pass multiple props into a component.

```
class Content extends React.Component {
  render() {
    return (
      <div className="content">
        <div className="line"></div>
{/* Timeline item */}
        <div className="item">
          <div className="avatar">
            <img
src="http://www.croop.cl/UI/twitter/images/doug.jpg"
/> Doug
        </div>
```

```
 <span className="time">
           An hour ago
         </span>
         <p>Ate lunch</p>
         <div className="commentCount">
           2
         </div>
       </div>
{/* ... */}

     </div>
   )
  }
}
```

- As we did with title, let's look at what props our Content component needs:

A user's avatar image

- A timestamp of the activity
- Text of the activity item
- Number of comments
- Let's say that we have a JavaScript object that represents an activity item.
- We will have a few fields, such as a string field (text) and a date object.
- We might have some nested objects, like a user and comments. For instance:

```
{
  timestamp: new Date().getTime(),
  text: "Ate lunch",
```

```
user: {
    id: 1,
    name: 'Nate',
    avatar:
"http://www.croop.cl/UI/twitter/images/doug.jpg"
  },
  comments: [
    { from: 'Ari', text: 'Me too!' }
  ]
}
```

- Just like we passed in a string title to the <Header /> component, we can take this activity object and pass it right into the Content component.
- Let's convert our component to display the details from this activity inside it's template.

- In order to pass a dynamic variable's value into a template, we have to use the template syntax to render it in our template. For instance:

```
{
  timestamp: new Date().getTime(),
  text: "Ate lunch",
  user: {
    id: 1,
    name: 'Nate',
    avatar:
"http://www.croop.cl/UI/twitter/images/doug.jpg"
  },
  comments: [
    { from: 'Ari', text: 'Me too!' }
  ] }
```

- Just like we passed in a string title to the <Header /> component, we can take this activity object and pass it right into the Content component.
- Let's convert our component to display the details from this activity inside it's template.
- In order to pass a dynamic variable's value into a template, we have to use the template syntax to render it in our template. For instance:

```
class Content extends React.Component {
  render() {
    const {activity} = this.props; // ES6 destructuring
return (
      <div className="content">
        <div className="line"></div>
```

```
{/* Timeline item */}
        <div className="item">
          <div className="avatar">
            <img
              alt={activity.text}
              src={activity.user.avatar} />
           {activity.user.name}
          </div>
<span className="time">
           {activity.timestamp}
          </span>
          <p>{activity.text}</p>
          <div className="commentCount">
           {activity.comments.length}
```

```
</div>
        </div>
      </div>
    )
}
```

- We've use a little bit of ES6 in our class definition on the first line of the render() function called destructuring. The two following lines are functionally equivalent:

```
// these lines do the same thing
const activity = this.props.activity;
const {activity} = this.props;
```

- Destructuring allows us to save on typing and define variables in a shorter, more compact way.

- We can then use this new content by passing in an object as a prop instead of a hard-coded string. For instance:

<Content activity={moment1} />

- However, you might have noticed that we would have to implement this multiple times with different comments.
- Instead, we could pass an array of objects into a component.
- Let's say we have an object that contains multiple activity items

```
const activities = [
  {
    timestamp: new Date().getTime(),
    text: "Ate lunch",
```

```
user: {
      id: 1, name: 'Nate',
      avatar:
"http://www.croop.cl/UI/twitter/images/doug.jpg"
    },
    comments: [{ from: 'Ari', text: 'Me too!' }]
  }, {
    timestamp: new Date().getTime(),
    text: "Woke up early for a beautiful run",
    user: {
      id: 2, name: 'Ari',
      avatar:
"http://www.croop.cl/UI/twitter/images/doug.jpg"
    },
```

- We can rearticulate our usage of <Content /> by passing in multiple activities instead of just one:

<Content activities={activities} />

- However, if we refresh the view nothing will show up! We need to first update our Content component to accept multiple activities.
- As we learned about previously, JSX is really just JavaScript executed by the browser.
- We can execute JavaScript functions inside the JSX content as it will just get run by the browser like the rest of our JavaScript.
- Let's move our activity item JSX inside of the function of the map function that we'll run over for every item.

```
class Content extends React.Component {
  render() {
    const {activities} = this.props; // ES6 destructuring
return (
      <div className="content">
        <div className="line"></div>
 {/* Timeline item */}
        {activities.map((activity) => {
          return (
            <div className="item">
              <div className="avatar">
                <img
                  alt={activity.text}
                  src={activity.user.avatar} />
```

```
{activity.user.name}
            </div>
<span className="time">
              {activity.timestamp}
            </span>
            <p>{activity.text}</p>
            <div className="commentCount">
              {activity.comments.length}
            </div>
          </div>
        );
      })</div>
    )
  }
```

- Now we can pass any number of activities to our array and the Content component will handle it, however if we leave the component right now, then we'll have a relatively complex component handling both containing and displaying a list of activities

ActivityItem

- Here is where it makes sense to write one more component to contain displaying a single activity item and then rather than building a complex Content component, we can move the responsibility.
- This will also make it easier to test, add functionality, etc.
- Let's update our Content component to display a list of ActivityItem components (we'll create this next).

```
class Content extends React.Component {
  render() {
    const {activities} = this.props; // ES6 destructuring
return (
      <div className="content">
        <div className="line"></div>

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

- Not only is this much simpler and easier to understand, but it makes testing both components easier.
- With our freshly-minted Content component, let's create the ActivityItem component.
- Since we already have the view created for the ActivityItem, all we need to do is copy it from what was our Content component's template as it's own module.

```
class ActivityItem extends React.Component {
  render() {
    const {activity} = this.props; // ES6 destructuring
 return (
      <div className="item">
        <div className="avatar">
```

```
<img
          alt={activity.text}
          src={activity.user.avatar} />
        {activity.user.name}
      </div>
<span className="time">
        {activity.timestamp}
      </span>
      <p>{activity.text}</p>
      <div className="commentCount">
        {activity.comments.length}
      </div>
    </div>
  )}
}
```

Summary:

- Controlled and UnControlled Components
- Repeating Elements
- react-starter project
- Fetching Remote Data
- Data-Driven

Thank You.....

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

Uplatz