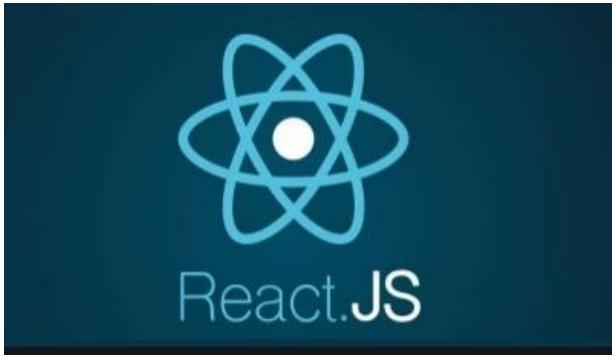

Presentation by Uplatz

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

Email: info@uplatz.com

Phone:+44 7836 212635

Uplatz

Table Of Contents:

Ajax File Uploads

A Simple Complete Example

- We could use this sample code to upload the files selected by the user every time a new file selection is made.

```
<input type="file" id="file-input" multiple>
var files;
var fdata = new FormData();
$("#file-input").on("change", function (e) {
 files = this.files;
 $.each(files, function (i, file) {
 fdata.append("file" + i, file);
 });
```

```
fdata.append("FullName", "John Doe");
 fdata.append("Gender", "Male");
 fdata.append("Age", "24");
 $.ajax({
 url: "/Test/Url",
 type: "post",
 data: fdata, //add the FormData object to the data parameter
 processData: false, //tell jquery not to process data
contentType: false, //tell jquery not to set content-type
 success: function (response, status, jqxhr) {
 //handle success
 },
 error: function (jqxhr, status, errorMessage) {
```

```
//handle error
 }
 });});
```

Now let's break this down and inspect it part by part.

Working With File Inputs

- This MDN Document (Using files from web applications) is a good read about various methods on how to handle file inputs.
- Some of these methods will also be used in this example.
- Before we get to uploading files, we first need to give the user a way to select the files they want to upload. For this purpose we will use a file input.
- The multiple property allows for selecting more than one files, you can remove it if you want the user to select one file at a time.

<input type="file" id="file-input" multiple>

- We will be using input's change event to capture the files.

```
var files;
$("#file-input").on("change", function(e){
 files = this.files;
});
```

- Inside the handler function, we access the files through the files property of our input.
- This gives us a FileList, which is an array like object.

Creating and Filling the FormData

- In order to upload files with Ajax we are going to use FormData.

```
var fdata = new FormData();
```

- FileList we have obtained in the previous step is an array like object and can be iterated using various methods including for loop, for...of loop and jQuery.each.
- We will be sticking with the jQuery in this example.

```
$.each(files, function(i, file) {
 //...
});
```

- We will be using the append method of FormData to add the files into our formdata object.

```
$.each(files, function(i, file) {
 fdata.append("file" + i, file);
});
```

- We can also add other data we want to send the same way.
- Let's say we want to send some personal information
- we have received from the user along with the files.
- We could add this this information into our formdata object.

```
fdata.append("FullName", "John Doe");
fdata.append("Gender", "Male");
fdata.append("Age", "24");
//...
```

Sending the Files With Ajax

```
$.ajax({
 url: "/Test/Url",
 type: "post",
```

```
data: fdata, //add the FormData object to the data parameter
 processData: false, //tell jquery not to process data
 contentType: false, //tell jquery not to set content-type
 success: function (response, status, jqxhr) {
 //handle success
 },
 error: function (jqxhr, status, errorMessage) {
 //handle error
 }
});
```

- We set processData and contentType properties to false.
- This is done so that the files can be send to the server and be processed by the server correctly

Checkbox Select all with automatic check/uncheck on othercheckbox change

- I've used various Stackoverflow examples and answers to come to this really simple example on how to manage
- "select all" checkbox coupled with an automatic check/uncheck if any of the group checkbox status changes.
- Constraint: The "select all" id must match the input names to create the select all group. In the example, the input
- select all ID is cbGroup1. The input names are also cbGroup1 Code is very short, not plenty of if statement (time and resource consuming).

select all checkboxes with corresponding group checkboxes

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.
1/jquery.min.js"></script>
<p>
<input id="cbGroup1" type="checkbox">Select all
<input name="cbGroup1" type="checkbox"
value="value1_1">Group1 value 1
<input name="cbGroup1" type="checkbox"
value="value1_2">Group1 value 2
<input name="cbGroup1" type="checkbox"
value="value1_3">Group1 value 3
</p><p>
<input id="cbGroup2" type="checkbox">Select all
<input name="cbGroup2" type="checkbox"
value="value2_1">Group2 value 1
```

```
<input name="cbGroup2" type="checkbox"
value="value2_2">Group2 value 2
<input name="cbGroup2" type="checkbox"
value="value2_3">Group2 value 3
</p>
<script type="text/javascript" language="javascript">
 $("input").change(function() {

$('input[name=\"+this.id+'\']').not(this).prop('checked',
this.checked);
 $('#'+this.name).prop('checked',
$('input[name=\"+this.name+'\']').length ===
$('input[name=\"+this.name+'\']').filter(':checked').len
gth);
 }); </script>
```

Plugins

Plugins - Getting Started

- The jQuery API may be extended by adding to its prototype.
- For example, the existing API already has many
- functions available such as .hide(), .fadeIn(), .hasClass(), etc.
- The jQuery prototype is exposed through $.fn, the source code contains the line

jQuery.fn = jQuery.prototype

- Adding functions to this prototype will allow those functions to be available to be called from any constructed
- jQuery object (which is done implicitly with each call to jQuery, or each call to $ if you prefer).

- A constructed jQuery object will hold an internal array of elements based on the selector passed to it. For example,
- $('.active') will construct a jQuery object that holds elements with the active class, at the time of calling (as in, this is not a live set of elements).
- The this value inside of the plugin function will refer to the constructed jQuery object.
- As a result, this is used to represent the matched set.

Basic Plugin:

```
$.fn.highlight = function() {
 this.css({ background: "yellow" });
};
// Use example:
$("span").highlight();
```

Chainability & Reusability

- Unlike the example above, jQuery Plugins are expected to be Chainable.
- What this means is the possibility to chain multiple Methods to a same Collection of Elements like

$(".warn").append("WARNING! ").css({color:"red"}) (see how we used the .css() method after the .append(), both methods apply on the same .warn Collection)

- Allowing one to use the same plugin on different Collections passing different customization options plays an important role in Customization / Reusability

```
(function($) {
 $.fn.highlight = function( custom ) {
 // Default settings
```

```
var settings = $.extend({
 color : "", // Default to current text color
 background : "yellow" // Default to yellow background
 }, custom);
 return this.css({ // `return this` maintains method chainability
 color : settings.color,
backgroundColor : settings.background
 });
 };
}( jQuery ));
// Use Default settings
```

```
$("span").highlight(); // you can chain other methods
// Use Custom settings
$("span").highlight({
 background: "#f00",
 color: "white"
});
```

Freedom

- The above examples are in the scope of understanding basic Plugin creation.
- Keep in mind to not restrict a user to a limited set of customization options.
- Say for example you want to build a .highlight() Plugin where you can pass a desired text String that will be highlighted and allow maximal freedom regarding styles:

```
//...
// Default settings
var settings = $.extend({
 text : "", // text to highlight
 class : "highlight" // reference to CSS class
}, custom);
return this.each(function() {
 // your word highlighting logic here
});
//...
```

- the user can now pass a desired text and have complete control over the added styles by using a custom CSS class:

```
$("#content").highlight({
 text : "hello",
 class : "makeYellowBig"
});
```

Using ReactJS with jQuery

ReactJS with jQuery

- Firstly, you have to import jquery library .
- We also need to import findDOmNode as we're going to manipulate the dom.
- And obviously we are importing React as well.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

- We are setting an arrow function ‘handleToggle’ that will fire when an icon will be clicked.
- We’re just showing and hiding a div with a reference naming ‘toggle’ onClick over an icon.

```
handleToggle = () => {
 const el = findDOMNode(this.refs.toggle);
 $(el).slideToggle();
};
```

- Let’s now set the reference naming ‘toggle’

```
<ul className=”profile-info additional-profile-info-list” ref=”toggle”>
 <li>
 <span className=”info-email”>Office Email</span> me@shuvohabib.com
 </li>
```

```
</ul>
```

- The div element where we will fire the 'handleToggle' on onClick.

```
<div className="ellipsis-click"
onClick={this.handleToggle}>
 <i className="fa-ellipsis-h"/>
</div>
```

- Let review the full code below, how it looks like .

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
export default class FullDesc extends
React.Component {
 constructor() {
```

```
super();
 }
 handleToggle = () => {
 const el = findDOMNode(this.refs.toggle);
 $(el).slideToggle();
 };
 render() {
 return (
 <div className="long-desc">
 <ul className="profile-info">
 <li>
 <span className="info-title">User Name : </span>
Shuvo Habib
```

```
</li>
 </ul>
 <ul className="profile-info additional-profile-info-list" ref="toggle">
 <li>
 <span className="info-email">Office Email</span> me@shuvohabib.com
 </li>
 </ul> <div className="ellipsis-click" onClick={this.handleToggle}>
 <i className="fa-ellipsis-h"/>
 </div>
 </div>
 );
 }}
```

Introduction to Promises

What is a promise

- As defined by the Mozilla, a Promise object is used for handling asynchronous computations which has some important guarantees
- that are difficult to handle with the callback method (the more old-school method of handling asynchronous code).
- A Promise object is simply a wrapper around a value that may or may not be known when the object is instantiated
- and provides a method for handling the value after it is known (also known as resolved) or is unavailable for a failure reason (we'll refer to this as rejected).

- Using a Promise object gives us the opportunity to associate functionality for an asynchronous operation's eventual success or failure (for whatever reason).
- It also allows us to treat these complex scenarios by using synchronous-like code.
- For instance, consider the following synchronous code where we print out the current time in the JavaScript console:

```
var currentTime = new Date();
console.log('The current time is: ' + currentTime);
```

- This is pretty straight-forward and works as the new Date() object represents the time the browser knows about.

- Now consider that we're using a different clock on some other remote machine.
- For instance, if we're making a Happy New Years clock, it would be great to be able to synchronize the user's browser with everyone elses using a single time value for everyone so no-one misses the ball dropping ceremony.
- Suppose we have a method that handles getting the current time for the clock called **getCurrentTime**() that fetches the current time from a remote server.
- We'll represent this now with a **setTimeout**() that returns the time (like it's making a request to a slow API):

```
function getCurrentTime() {
  // Get the current 'global' time from an API
```

```
 return setTimeout(function() {
    return new Date();
  }, 2000);
}
var currentTime = getCurrentTime()
console.log('The current time is: ' + currentTime);
```

- Our console.log() log value will return the timeout handler id, which is definitely not the current time.
- Traditionally, we can update the code using a callback to get called when the time is available:

```
function getCurrentTime(callback) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    var currentTime = new Date();
    callback(currentTime);
```

```
  }, 2000);
}
getCurrentTime(function(currentTime) {
  console.log('The current time is: ' + currentTime);
});
```

- What if there is an error with the rest? How do we catch the error and define a retry or error state?

```
function getCurrentTime(onSuccess, onFail) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    // randomly decide if the date is retrieved or not
    var didSucceed = Math.random() >= 0.5;
    if (didSucceed) {
      var currentTime = new Date();
      onSuccess(currentTime);
```

```
 } else {
      onFail('Unknown error');
    }
  }, 2000);
}
getCurrentTime(function(currentTime) {
  console.log('The current time is: ' + currentTime);
}, function(error) {
  console.log('There was an error fetching the time');
});
```

- Now, what if we want to make a request based upon the first request's value?
- As a short example, let's reuse the getCurrentTime()

- function inside again (as though it were a second method, but allows us to avoid adding another complex-looking function):

```
function getCurrentTime(onSuccess, onFail) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    // randomly decide if the date is retrieved or not
    var didSucceed = Math.random() >= 0.5;
    console.log(didSucceed);
    if (didSucceed) {
      var currentTime = new Date();
      onSuccess(currentTime);
    } else {
      onFail('Unknown error');
    }
```

```
}, 2000);
}
getCurrentTime(function(currentTime) {
  getCurrentTime(function(newCurrentTime) {
    console.log('The real current time is: ' + currentTime);
  }, function(nestedError) {
    console.log('There was an error fetching the second
time');
  })
}, function(error) {
  console.log('There was an error fetching the time');
});
```

Dealing with asynchronousity in this way can get complex quickly.

- In addition, we could be fetching values from a previous function call, what if we only want to get one... there are a lot of tricky cases to deal with when dealing with values that are not yet available when our app starts.

Enter Promises

- Using promises, on the other hand helps us avoid a lot of this complexity (although is not a silver bullet solution).
- The previous code, which could be called spaghetti code can be turned into a neater, more synchronous-looking version:

```
function getCurrentTime() {
  // Get the current 'global' time from an API using
Promise
```

```
return new Promise((resolve, reject) => {
    setTimeout(function() {
      var didSucceed = Math.random() >= 0.5;
      didSucceed ? resolve(new Date()) : reject('Error');
    }, 2000);
  })
}
getCurrentTime()
  .then(currentTime => getCurrentTime())
  .then(currentTime => {
    console.log('The current time is: ' + currentTime);
    return true;
  })
```

.catch(err => console.log('There was an error:' + err))

- The current time is: Sun Apr 19 2020 11:46:34 GMT+0530 (India Standard Time)
- This previous source example is a bit cleaner and clear as to what's going on and avoids a lot of tricky error handling/catching.
- To catch the value on success, we'll use the then() function available on the Promise instance object.
- The then() function is called with whatever the return value is of the promise itself.
- For instance, in the example above, the **getCurrentTime**() function resolves with the **currentTime**() value (on successful completion) and calls the then() function on the return value (which is another promise) and so on and so forth.

- To catch an error that occurs anywhere in the promise chain, we can use the catch() method.
- We're using a promise chain in the above example to create a chain of actions to be called one after another.
- A promise chain sounds complex, but it's fundamentally simple.
- Essentially, we can "synchronize" a call to multiple asynchronous operations in succession.
- Each call to then() is called with the previous then() function's return value.
- For instance, if we wanted to manipulate the value of the getCurrentTime() call, we can add a link in the chain, like so:

```
getCurrentTime()
  .then(currentTime => getCurrentTime())
  .then(currentTime => {
    return 'It is now: ' + currentTime;
  })
  // this logs: "It is now: [current time]"
  .then(currentTimeMessage =>
console.log(currentTimeMessage))
  .catch(err => console.log('There was an error:' + err))
```

Single-use guarantee

- A promise only ever has one of three states at any given time:

pending

fulfilled (resolved)

- rejected (error)
- A pending promise can only ever lead to either a fulfilled state or a rejected state once and only once, which can avoid some pretty complex error scenarios.
- This means that we can only ever return a promise once. If we want to rerun a function that uses promises, we need to create a new one.

Creating a promise

- We can create new promises (as the example shows above) using the Promise constructor.
- It accepts a function that will get run with two parameters:
- The onSuccess (or resolve) function to be called on success resolution

- The onFail (or reject) function to be called on failure rejection
- Recalling our function from above, we can see that we call the resolve() function if the request
- succeeded and call the reject() function if the method returns an error condition.

```
var promise = new Promise(function(resolve, reject) {
  // call resolve if the method succeeds
  resolve(true);
})
promise.then(bool => console.log('Bool is true'))
```

Displaying Remote Data

Fetching data

- First, the basis of the wrapper component which will show and fetch the current time looks like the following.
- Let's copy and paste this code into our app at src/App.js:

```
import React from 'react';
import 'whatwg-fetch';
import './App.css';
import TimeForm from './TimeForm';
class App extends React.Component {
  constructor(props) {
    super(props);
this.fetchCurrentTime = this.fetchCurrentTime.bind(this);
    this.handleFormSubmit =
this.handleFormSubmit.bind(this);
```

```
 this.state = {
      currentTime: null, msg: 'now'
    }
  }
// methods we'll fill in shortly
  fetchCurrentTime() {}
  getApiUrl() {}
  handleFormSubmit(evt) {}
  handleChange(newState) {}
render() {
    const {currentTime, tz} = this.state;
    const apiUrl = this.getApiUrl();
return (
      <div>
```

```
 {!currentTime &&
        <button onClick={this.fetchCurrentTime}>
          Get the current time
        </button>}
       {currentTime && <div>The current time is:
{currentTime}</div>}
       <TimeForm
         onFormSubmit={this.handleFormSubmit}
         onFormChange={this.handleChange}
         tz={tz}
         msg={'now'}
       />
       <p>We'll be making a request from:
<code>{apiUrl}</code></p>
     </div>
```

)}}

export default App;

- The previous component is a basic stateful React component as we've created.
- Since we'll want to show a form, we've included the intended usage of the TimeForm let's create next.
- Let's create this component in our react app using create-react-app. Add the file src/TimeForm.js into our project:

touch src/TimeForm.js

- Now let's add content. We'll want our TimeForm to take the role of allowing the user to switch between timezones in their browser.
- We can handle this by creating a stateful component we'll call the TimeForm.

- Our TimeForm component might look like the following:

```
import React from 'react'
const timezones = ['PST', 'MST', 'MDT', 'EST', 'UTC']
export class TimeForm extends React.Component {
constructor(props) {
    super(props);
 this._changeTimezone =
this._changeTimezone.bind(this);
    this._handleFormSubmit =
this._handleFormSubmit.bind(this);
    this._handleChange =
this._handleChange.bind(this);
    this._changeMsg = this._changeMsg.bind(this);
```

```
 const {tz, msg} = this.props;
    this.state = {tz, msg};
  }
_handleChange(evt) {
    typeof this.props.onFormChange === 'function' &&
      this.props.onFormChange(this.state);
  }
_changeTimezone(evt) {
    const tz = evt.target.value;
    this.setState({tz}, this._handleChange);
  }
_changeMsg(evt) {
    const msg =
```

```
encodeURIComponent(evt.target.value).replace(/%20
/g, '+');
    this.setState({msg}, this._handleChange);
  }
_handleFormSubmit(evt) {
    evt.preventDefault();
    typeof this.props.onFormSubmit === 'function' &&
      this.props.onFormSubmit(this.state);
  }
render() {
    const {tz} = this.state;
 return (
      <form onSubmit={this._handleFormSubmit}>
        <select
```

```
onChange={this._changeTimezone}
          defaultValue={tz}>
          {timezones.map(t => {
            return (<option key={t} value={t}>{t}</option>)
          })}
        </select>
        <input
          type="text"
          placeholder="A chronic string message (such as
7 hours from now)"
          onChange={this._changeMsg}
        />
        <input
          type="submit"
```

```
value="Update request"
        />
      </form>
    )
  }
}
export default TimeForm;
```

- With these Components created, let's load up our app in the browser after running it with npm start and we'll see our form (albeit not incredibly beautiful yet).
- Of course, at this point, we won't have a running component as we haven't implemented our data fetching

Fetching data

- As we said yesterday, we'll use the fetch() API with promise support.
- When we call the fetch() method, it will return us a promise, where we can handle the request however we want.
- We're going to make a request to our now-based API server (so start-up might be slow if it hasn't been run in a while).
- We're going to be building up the URL we'll request as it represents the time query we'll request on the server.
- I've already defined the method getApiUrl() in the App component, so let's fill that function in.

- The chronic api server accepts a few variables that we'll customize in the form.
- It will take the timezone to along with a chronic message.
- We'll start simply and ask the chronic library for the pst timezone and the current time (now):

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      currentTime: null, msg: 'now', tz: 'PST'
    }
  }
  // ...
  getApiUrl() {
```

```
const {tz, msg} = this.state;
    const host = 'https://andthetimeis.com';
    return host + '/' + tz + '/' + msg + '.json';
  }
  // ...
export default App;
```

- Now, when we call getApiUrl(), the URL of the next request will be returned for us.
- Now, finally, let's implement our fetch() function. The fetch() function accepts a few arguments that can help us customize our requests.
- The most basic GET request can just take a single URL endpoint.
- The return value on fetch() is a promise object

- Let's update our fetchCurrentTime() method to fetch the current time from the remote server.
- We'll use the .json() method on the response object to turn the body of the response from a JSON object into JavaScript object and then update our component by setting the response value of the dateString as the currentTime in the component state:

```
class App extends React.Component {
  // ...
  fetchCurrentTime() {
    fetch(this.getApiUrl())
      .then(resp => resp.json())
      .then(resp => {
        const currentTime = resp.dateString;
```

```
this.setState({currentTime})
    })
  }
  // ...}
```

- The final piece of our project today is getting the data back from the form to update the parent component.
- That is, when the user updates the values from the TimeForm component, we'll want to be able to access the data in the App component.
- The TimeForm component already handles this process for us, so we just need to implement our form functions.

- When a piece of state changes on the form component, it will call a prop called onFormChange.
- By defining this method in our App component, we can get access to the latest version of the form.
- In fact, we'll just call setState() to keep track of the options the form allows the user to manipulate:

```
class App extends React.Component {
// ...
  handleChange(newState) {
    this.setState(newState);
  }
// ...
}
```

- Finally, when the user submits the form (clicks on the button or presses enter in the input field), we'll want to make another request for the time.
- This means we can define our handleFormSubmit prop to just call the fetchCurrentTime() method:

```
class App extends React.Component {
// ...
  handleFormSubmit(evt) {
    this.fetchCurrentTime();
  }
// ...
}
```

Summary:

- Ajax File Uploads
- Creating and Filling the FormData
- Introduction to Promises
- Fetching data

Thank You.....

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

Uplatz