DESARROLLO WEB FULLSTACK

con Python y JavaScript

polotic misiones

polotic misiones

Funciones Anónimas

Una de las cosas mas significativas que se pueden hacer con JavaScript es la posibilidad de declarar una función sin nombre y asignarla a una constante o variable, como por ejemplo de la siguiente manera:

```
const hola = function(){
     console.log(“Hola mundo”);
}
```

Y después puedes pasarla como parámetro a otra funcionalidad como por ejemplo setTimeout que sirve para implementar un delay de tiempo en la ejecución del codigo (setTimeout la uso como ejemplo nada más, pero puedes usarla con las que quieras)

```
setTimeout(hola, 1000);
```

Funciones Anónimas

Puedo incluso, obviar la declaración de la función y pasarla directamente como parámetro al setTimeout como puedes ver a continuación:

```
setTimeout( function(){
    console.log(“Hola mundo”);
} , 1000);
```

Éste tipo de escritura de código es muy común en JavaScript y lo verás constantemente.

Funciones Flecha

Además de la notación de función tradicional que ya hemos visto, JavaScript ahora nos da la capacidad de usar funciones de flecha donde tenemos una entrada (o paréntesis cuando no hay entrada) seguida de => seguido de algún código para ejecutar.
Por ejemplo, podemos modificar nuestro script anterior para usar una función de flecha anónima:

```
document.addEventListener('DOMContentLoaded', () => {
    document.querySelectorAll('button').forEach(button => {
        button.onclick = () => {
            document.querySelector("#hola").style.color = button.dataset.color;
        }
    });
});
```

Funciones Flecha

También podemos tener funciones nombradas que usan flechas, como en esta reescritura de la función de conteo:

```
contar = () => {
    contador++;
    document.querySelector('h1').innerHTML = contador;

    if (contador % 10 === 0) {
        alert(`Contar es ahora ${contador}`)
    }
}
```

Funciones Flecha

Para tener una idea sobre otros eventos que podemos usar, veamos cómo podemos implementar nuestro selector de color usando un menú desplegable en lugar de tres botones separados.

Podemos detectar cambios en un elemento seleccionado usando el atributo `onchange`.

En JavaScript, esta es una palabra clave que cambia según el contexto en el que se usa.

En el caso de un controlador de eventos, se refiere al objeto que desencadenó el evento.

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Colores</title>
        <script>
            document.addEventListener('DOMContentLoaded', function() {
                document.querySelector('select').onchange = function() {
                    document.querySelector('#hola').style.color = this.value;
                }
            });
        </script>
    </head>
    <body>
        <h1 id="hola">Hola</h1>
        <select>
            <option value="black">Negro</option>
            <option value="red">Rojo</option>
            <option value="blue">Azul</option>
            <option value="green">Verde</option>
        </select>

    </body>
</html>
```

Local Storage

Una cosa a tener en cuenta sobre todos nuestros sitios hasta ahora es que cada vez que recargamos la página, se pierde toda nuestra información. El color del título vuelve a ser negro, el contador vuelve a 0 y todas las tareas se borran. A veces, esto es lo que pretendemos, pero en otras ocasiones querremos poder almacenar información que podamos usar cuando un usuario regrese al sitio.

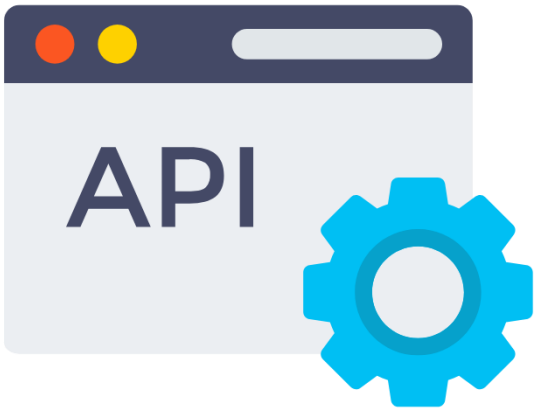
Una forma de hacerlo es mediante el almacenamiento local (local storage) que refiere a almacenar información en el navegador web del usuario al que podemos acceder más tarde. Esta información se almacena como un conjunto de pares clave-valor, casi como un diccionario de Python. Para utilizar el almacenamiento local, emplearemos dos funciones clave:

- `localStorage.getItem(clave):` esta función busca una entrada en el almacenamiento local con una clave determinada y devuelve el valor asociado con esa clave.
- `localStorage.setItem(clave, valor):` esta función establece una entrada en el almacenamiento local, asociando la clave con un nuevo valor.

Local Storage

Veamos cómo podemos usar estas nuevas funciones para actualizar nuestro contador. En el siguiente código:

```
// Verifica si ya existe un valor en el almacenamiento local
if (!localStorage.getItem('contador')) {

    // Si no, configura el contador a 0 en el almacenamiento local
    localStorage.setItem('contador', 0);
}

function conteo() {
    // Recuperar el valor del contador del almacenamiento local
    let counter = localStorage.getItem('contador');

    // actualizar contador
    contador++;
    document.querySelector('h1').innerHTML = contador;

    // Almacenar el contador en el almacenamiento local
    localStorage.setItem('contador', contador);
}

document.addEventListener('DOMContentLoaded', function() {
    // Establecer el encabezado al valor actual dentro del almacenamiento local
    document.querySelector('h1').innerHTML = localStorage.getItem('counter');
    document.querySelector('button').onclick = contador;
});
```


API

Application Programming Interface

En informática, una **API** o *application programming interface* (o en español: interfaz de programación de aplicaciones) es una interfaz que define las interacciones entre múltiples aplicaciones de software o intermediarios mixtos de hardware y software.

Define los *tipos* de llamadas o solicitudes que se pueden realizar, *cómo* realizarlas, los *formatos* de datos que se deben utilizar, las *convenciones* a seguir, etc.

Una API puede ser completamente personalizada, específica de un componente o diseñada según un estándar de la industria para garantizar la interoperabilidad. A través de la ocultación de información, las API permiten la **programación modular**, lo que permite a los usuarios utilizar la interfaz **independientemente** de la implementación.

Application Programming Interface

Application Programming Interface

Un objeto de JavaScript es muy similar a un diccionario de Python, ya que nos permite almacenar pares clave-valor. Por ejemplo, podría crear un objeto JavaScript que represente a Harry Potter:

```
let persona = {
    nombre: 'Harry',
    apellido: 'Potter'
};
```

Luego puedo acceder o cambiar partes de este objeto usando la notación entre corchetes o puntos:

{JSON}

JavaScript Object Notation

polotic misiones

Una forma en que los objetos JavaScript son realmente útiles es en la transferencia de datos de un sitio a otro, especialmente cuando se utilizan APIs.

Es posible que deseemos que nuestra aplicación obtenga información de Google Maps, Amazon o algún servicio meteorológico. Podemos hacer esto haciendo llamadas a la API de un servicio, que nos devolverá datos estructurados, a menudo en formato JSON (JavaScript Object Notation). Por ejemplo, un vuelo en formato JSON podría verse así:

Los valores dentro de un JSON no tienen que ser solo cadenas y números como en el ejemplo anterior. También podemos almacenar listas, o incluso otros objetos JavaScript:

```
{
   "origen": {"ciudad": "Buenos Aires", "codigo": "EZE"},
   "destino": {"ciudad": "Londres", "codigo": "LHR"}, "duracion": 14
}
```

JavaScript Object Notation

Para mostrar cómo podemos usar las API en nuestras aplicaciones, trabajemos en la creación de una aplicación en la que podamos encontrar tipos de cambio entre dos monedas. A lo largo del ejercicio, utilizaremos la API de tipos de cambio la pagina DolarSI.

`https://www.dolarsi.com/api/api.php?type=valoresprincipales`.

Cuando visitas esta página, verás el tipo de cambio entre el peso argentino y el dólar estadounidense escrito en formato JSON. También puede cambiar el parámetro GET en la URL de `type` de `valoresprincipales` a `cotizador` para cambiar el tipo de información a obtener.

Veamos cómo implementar esta API en una aplicación mediante la creación de un nuevo archivo HTML llamado `cotización.html` y vincularlo a un archivo JavaScript.

Ejemplo

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Cotizador de Moneda</title>
        <script src="cotizador.js"></script>
    </head>
    <body></body>
</html>
```

Ahora, usaremos algo llamado **`AJAX`** (Asynchronous JavaScript and XML), que nos permite acceder a información de páginas externas incluso después de que nuestra página se haya cargado.

Para hacer esto, usaremos la función de recuperación `fetch` que nos permitirá enviar una solicitud `HTTP request`.

La función `fetch` devuelve una `promise`. No hablaremos de los detalles de lo que es una `promise` aquí, pero podemos pensar en ella como un valor que se manifestará en algún momento, pero no necesariamente de inmediato (recuerda que una solicitud fuera de la computadora local si o si demora un tiempo x).

Nos ocupamos de las **`promise`** dándoles un atributo **`.then`** que describe lo que se debe hacer cuando recibimos una **`response`**.

AJAX

polotic misiones

```
document.addEventListener('DOMContentLoaded', function() {
    // Enviar una GET request a la URL
    fetch(' https://www.dolarsi.com/api/api.php?type=valoresprincipales')
    // Colocar la respuesta en una forma JSON
    .then(response => response.json())
    .then(datos => {
        // Registrar info en la consola
        console.log(datos);
    });
});
```

Un punto importante sobre el código anterior es que el argumento de `.then` **siempre** es una función que se ejecuta una vez que los datos han sido recibidos.

Aunque parece que estamos creando las variables `response` y `datos`, esas variables son solo los parámetros de dos funciones anónimas.

AJAX

polotic misiones

En lugar de simplemente registrar estos datos, podemos usar JavaScript para mostrar un mensaje en la pantalla, como en el siguiente código:

```
document.addEventListener('DOMContentLoaded', function() {

    // Enviar una GET request a la URL
    fetch(' https://www.dolarsi.com/api/api.php?type=valoresprincipales ‘)

    // Colocar la respuesta en una forma JSON
    .then(response => response.json())
    .then(datos => {
        // Obtener el valor de un elemento del JSON
        const dolar_compra = datos.casa.compra;
        // Mostrar el valor en la pantalla
        document.querySelector('body').innerHTML = `1 USD es igual a ${rate.toFixed(3)} ARS en tipo de cambio oficial (compra).`;
    });

});
```

Interfaces de Usuario

- Hasta ahora hemos aprendido a trabajar con **Python** en lo que sería el backend, aprendimos también a construir el frontend usando muchas herramientas como **HTML**, **CSS**, **Bootstrap**, **diseño responsivo** y demás. Luego introdujimos **JavaScript** y aprendimos cómo usarlo para hacer que las páginas web fueran más interactivas.

- Ahora, analizaremos paradigmas comunes en el diseño de la **interfaz de usuario**, utilizando JavaScript y CSS para hacer que nuestros sitios sean aún más fáciles de usar.

- Una interfaz de usuario es la forma en que los visitantes de una página web interactúan con esa página. Nuestro objetivo como **desarrolladores web** es hacer que estas interacciones sean lo más agradables posible para el usuario, y hay muchos métodos que podemos utilizar para hacerlo.

SinglePage Application

- Anteriormente, si queríamos un sitio web con varias páginas, lo lograríamos usando diferentes paginas (archivos) HTML. Ahora, tenemos la capacidad de cargar una sola página y luego usar JavaScript para manipular el DOM.

- Una ventaja importante de hacer esto es que solo necesitamos modificar la parte de la página que realmente está cambiando. Por ejemplo, si tenemos una barra de navegación que no cambia en función de su página actual, no querríamos tener que volver a renderizar esa barra de navegación cada vez que cambiamos a una nueva parte de la página.

SinglePage Application

Observa en el HTML de la derecha que tenemos tres botones y tres divs.

Por el momento, los divs contienen solo una pequeña parte de texto, pero podríamos imaginar que cada div contiene el contenido de una página de nuestro sitio.

Ahora, agregaremos algo de JavaScript que nos permite usar los botones para alternar entre páginas.

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Una sola página</title>
        <style>
            div {
                display: none;
            }
        </style>
        <script src="unapagina.js"></script>
    </head>
    <body>
        <button data-page="pagina1">Pagina 1</button>
        <button data-page="pagina2">Pagina 2</button>
        <button data-page="pagina3">Pagina 3</button>
        <div id="pagina1">
            <h1>Esta es la Pagina 1</h1>
        </div>
        <div id="pagina2">
            <h1>Esta es la Pagina 2</h1>
        </div>
        <div id="pagina3">
            <h1>Esta es la Pagina 3</h1>
        </div>
    </body>
</html>
```

SinglePage Application

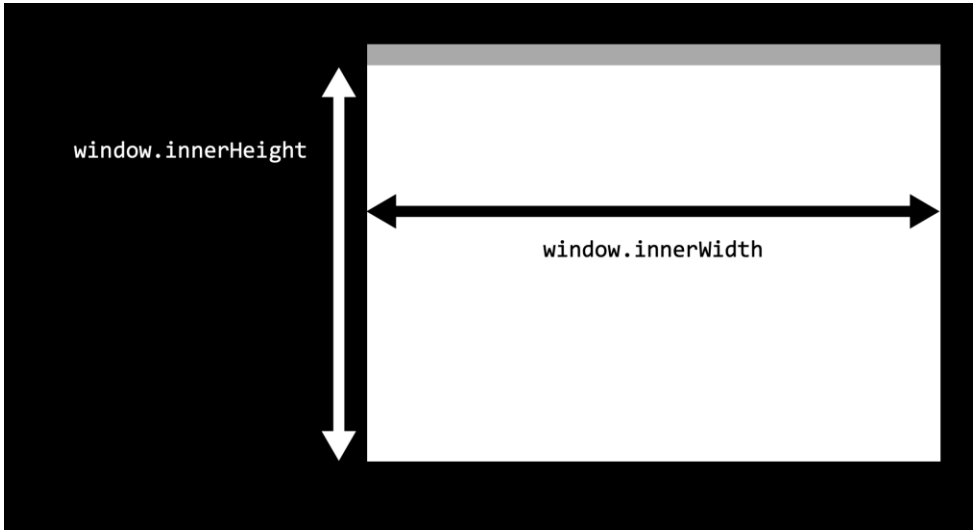
```html
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Una sola página</title>
        <style>
            div {
                display: none;
            }
        </style>
        <script src="unapagina.js"></script>
    </head>
    <body>
        <button data-page="pagina1">Pagina 1</button>
        <button data-page="pagina2">Pagina 2</button>
        <button data-page="pagina3">Pagina 3</button>
        <div id="pagina1">
            <h1>Esta es la Pagina 1</h1>
        </div>
        <div id="pagina2">
            <h1>Esta es la Pagina 2</h1>
        </div>
        <div id="pagina3">
            <h1>Esta es la Pagina 3</h1>
        </div>
    </body>
</html>
```

```javascript
// Muestra una página y oculta las otras dos.
function mostrarPagina(pagina) {

    // Oculta todos los divs:
    document.querySelectorAll('div').forEach(div => {
        div.style.display = 'none';
    });

    // Muestra el div provisto en el argumento
    document.querySelector(`#${pagina}`).style.display = 'block';
}

// Espera que la pagina cargue:
document.addEventListener('DOMContentLoaded', function() {

    // Selecciona todos los botones
    document.querySelectorAll('button').forEach(button => {

        // Cuando se clickea un boton muestra la pagina
        button.onclick = function() {
            mostrarPagina(this.dataset.pagina);
        }
    })
});
```

Scroll

Para actualizar y acceder al historial del navegador, utilizamos un objeto JavaScript importante conocido como **`window`** (ventana). Hay algunas otras propiedades de **`window`** que podemos usar para hacer que nuestros sitios se vean mejor:

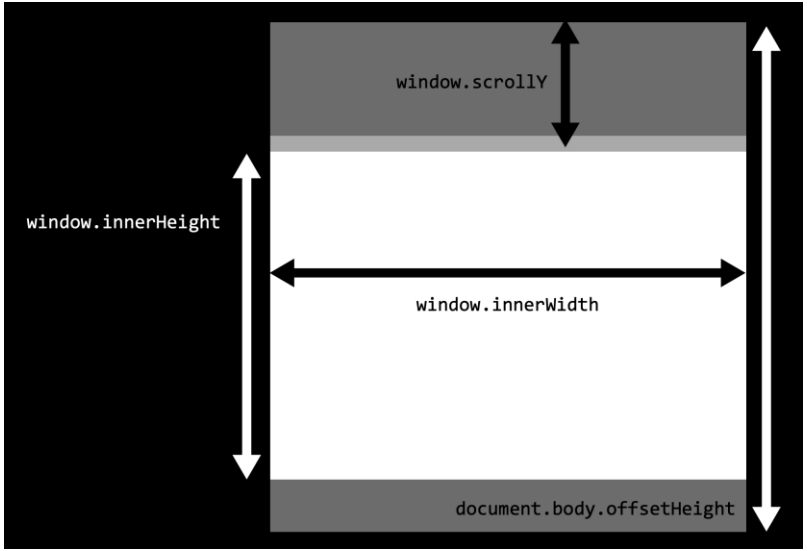
- `window.innerWidth`: Ancho de la ventana en píxeles
- `window.innerHeight`: Altura de la ventana en píxeles



Scroll

Si bien **`window`** representa lo que actualmente es visible para el usuario, el documento se refiere a la página web completa, que a menudo es mucho más grande que la ventana, lo que obliga al usuario a desplazarse hacia arriba y hacia abajo para ver el contenido de la página. Para trabajar con nuestro desplazamiento, tenemos acceso a otras variables:

- `window.scrollY`: cuántos píxeles hemos desplazado desde la parte superior de la página
- `document.body.offsetHeight`: la altura en píxeles de todo el documento.



Scroll

Podemos usar estas medidas para determinar si el usuario se ha desplazado o no al final de una página usando la comparación:

```
window.scrollY + window.innerHeight =
document.body.offsetHeight
```

La siguiente página, por ejemplo, cambiará el color del fondo a verde cuando lleguemos al final de una página:

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Mi Scroll</title>
        <script>

            // Event listener para scrollear
            window.onscroll = () => {

                // Verificar si estamos al final
                if (window.innerHeight + window.scrollY >= document.body.offsetHeight) {

                    // Cambiar el color a verde
                    document.querySelector('body').style.background = 'green';
                } else {

                    // Cambiar el color a blanco
                    document.querySelector('body').style.background = 'white';
                }

            };

        </script>
    </head>
    <body>
        <p>1</p>
        <p>2</p>
        <!- No pongo mas para no llenar la diapositiva -->
        <p>99</p>
        <p>100</p>
    </body>
</html>
```

polotic misiones

¿Qué es React.js?

- Desarrollado por facebook
- React es una libería de capas de vista, no un framework como Backbone, Angular, etc. (*React Native* si es un framework, pero eso es otra historia…)
- No puedes usar React para crear una aplicación web completamente funcional
- Es utilizada para facilitar el desarrollo de aplicaciones de una sola página

¿Por qué se desarrolló React.js?

- Complejidad del enlace de datos bidireccional
- Mala UX debido al uso de "actualizaciones en cascada" del árbol DOM
- Una gran cantidad de datos en una página cambian con el tiempo
- Complejidad de la arquitectura de la UI de Facebook
- Cambio de la mentalidad MVC

¿Quién usa React.js?

polotic misiones

facebook

TERADEK

Instagram

NETFLIX

KHANACADEMY

reddit

asana:

airbnb

- https://github.com/facebook/react/wiki/Sites-Using-React

React.js: Lo bueno

- Es fácil entender que es lo que renderizará un componente
- Código declarativo → código predecible
- Realmente no necesitas analizar el JS en el archivo de vista para comprender lo que hace el archivo
- Fácil de mezclar HTML y JS
- Utiliza toda la potencia de JS
- El desacoplamiento de las plantillas de la lógica no depende de las abstracciones primitivas de las plantillas, sino que utiliza todo el poder de JavaScript para mostrar vistas.
- Sin flujo de datos bidireccional complejo
- Usa menos código

React: ¡es rápido!

- El DOM real es lento
- JavaScript es rápido
- El uso de objetos Virtual DOM permite actualizaciones por lotes rápidas del DOM real, con grandes ganancias de productividad sobre las frecuentes actualizaciones en cascada del árbol DOM
- La extensión React Chrome hace que la depuración sea mucho más fácil
- Renderización del lado del servidor: React.renderToString() retorna puro HTML

React: Qué no es

- No hay eventos
- Sin XHR
- Sin datos / modelos
- Sin promesas / aplazamientos

React: ¿Para que es ideal?

- Vistas fáciles de leer y comprender
- El concepto de componentes es el futuro del desarrollo web
- Si su página usa muchos datos de actualización rápida o datos en tiempo real, React es el camino a seguir
- Una vez que vos y tu equipo hayan superado la curva de aprendizaje de React, el desarrollo de tu aplicación será mucho más rápido.

¿De qué trata React?

Hay varias formas de usar React (incluido el popular comando create-react-app publicado por Facebook), pero hoy nos centraremos en comenzar directamente en un archivo HTML.

Para hacer esto, tendremos que importar tres paquetes de JavaScript:

- `React`: define componentes y su comportamiento.
- `ReactDOM`: toma los componentes de `React` y los inserta en el `DOM`
- `Babel`: traduce desde JSX, el lenguaje en el que escribiremos en `React`, hacia `JavaScript` plano que nuestros navegadores pueden interpretar. `JSX` es muy similar a `JavaScript`, pero con algunas características adicionales, incluida la capacidad de representar `HTML` dentro de nuestro código.

polotic misiones

Elementos Fundamentales

React: Component

- Los **component** son bloques de construcción autónomos y reutilizables de aplicaciones web.
- Los components de React son básicamente funciones idempotentes (la misma entrada produce la misma salida).
- Describen tu UI en cualquier momento, como una aplicación renderizada por un servidor.

React: Component

- Creada usando React.createClass()
- El único método requerido es render()
- Insertado en el DOM usando React.renderComponent()

```
var React = require('react'),
    SimpleView = React.createClass({
        render: function () {
            return <h1><strong>Ejemplo 1:</strong> Un simple componente</h1>;
        }
    });

React.renderComponent(SimpleView(), document.getElementById('ejemplo'));
```

React: Props

- Se transmite al componente desde el componente principal y representa los datos del componente.
- acceso via **this.props**

```
render: function() {
    var someProp = 'bar';

    console.log('component render()', this.props);

    return <div>
        <AnotherComponent foo={someProp} model={this.props.model} />
    </div>;
}
```

React: State

- Representa el estado interno del componente.
- Se accede a través de **this.state**
- Cuando los datos de estado de un componente cambian, el renderizado se actualizará volviendo a invocar el método render()

```
render: function() {
    return <h3>Conteo de Clicks:
        <span className='label label-default'>{this.state.clicks}</span>
    </h3>;
}
```

React: JSX

- Podría decirse que una de las cosas más geniales de React
- Sintaxis similar a XML para generar HTML de componentes
- Más fácil de leer y comprender grandes árboles DOM
- Se traduce a JavaScript simple usando **react-tools**

```
/** @jsx React.DOM */

render: function () {
    return <div>
        <h2>
            <strong>Example 4:</strong>  React App
        </h2>
    </div>;
}

/** regular DOM */

render: function () {
    return React.DOM.div(null,
        React.DOM.h2(
              null,
              React.DOM.strong(null, "Example 4:"),
              " React App")
    );
}
```

¡Construyamos nuestra primera aplicación REACT!

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
        <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
        <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
        <title>Hola</title>
    </head>
    <body>
        <div id="app" />
        <script type="text/babel">

            class App extends React.Component {

                render() {
                    return (
                        <div>
                            <h1>Bienvenido</h1>
                            ¡Hola!
                        </div>
                    );
                }
            }

            ReactDOM.render(<App />, document.querySelector("#app"));
        </script>
    </body>
</html>
```

React – Analicémos el código

Dado que esta es nuestra primera aplicación React, echemos un vistazo detallado a lo que hace cada parte del código anterior:

- En las tres líneas sobre el título, importamos las últimas versiones de React, ReactDom y Babel.
- En el cuerpo, incluimos un solo `div` con un `id` de `app`. Casi siempre queremos dejar esto vacío y completarlo con nuestro código de reacción a continuación.
- Incluimos una etiqueta de script donde especificamos ese `type="text / babel"`. Esto indica al navegador que el siguiente script debe traducirse con Babel.
- A continuación, creamos un componente llamado `App` que extiende `React.Component`. Los componentes en React se representan como clases de JavaScript, que son similares a las clases de Python que aprendimos anteriormente. Esto nos permite comenzar a crear un componente sin tener que volver a escribir mucho código incluido en la definición de la clase `React.Component`.
- Dentro de nuestro componente, incluimos una función de `render`. Se requiere que todos los componentes tengan esta función, y lo que se devuelva dentro de la función se agregará al DOM, en este caso, simplemente estamos agregando `<div>Hola</div>`.
- La última línea de nuestro script emplea la función `ReactDOM.render`, que toma dos argumentos:
    1. Un componente para renderizar
    2. Un elemento en el DOM dentro del cual se debe representar el componente.

React – Paso a Paso

Una característica útil de `React` es la capacidad de renderizar componentes dentro de otros componentes (**nesting**).
Para demostrar esto, creemos otro componente llamado `Hola`:

```
class Hola extends React.Component {
    render() {
        return (
            <h1>Hola</h1>
        );
    }
}
```

Y ahora, rendericemos tres componentes de Hola dentro de nuestro componente de App:

```
class App extends React.Component {
    render() {
        return (
            <div>
                <Hola />
                <Hola />
                <Hola />
            </div>
        );
    }
}
```

React – Paso a Paso

Hasta ahora, los componentes no han sido tan interesantes, ya que todos son exactamente iguales.

Agreguemos algunas propiedades adicionales (los **props** que mencioné anteriormente) a ellos.

Por ejemplo, digamos que deseamos saludar a tres personas diferentes. Podemos proporcionar los nombres de esas personas en un método similar a las etiquetas HTML:

```
class App extends React.Component {
    render() {
        return (
            <div>
                <Hola nombre="Harry" />
                <Hola nombre="Ron" />
                <Hola nombre="Hermione" />
            </div>
        );
    }
}
```

React – Paso a Paso

Luego podemos acceder a esas propiedades usando `this.props.PROP_NOMBRE`, donde representa el objeto actual. Luego podemos insertar esto en nuestro HTML usando llaves:

```
class Hola extends React.Component {
    render() {
        return (
            <h1>Hola, {this.props.nombre}!</h1>
        );
    }
}
```

React – Ejemplo del “Contador”

Ahora, veamos cómo podemos usar React para volver a implementar la página de contador que creamos cuando trabajamos por primera vez con JavaScript.
Nuestra estructura general seguirá siendo la misma, pero dentro de nuestra clase App, incluiremos un método **`constructor`**, un método llamado cuando se crea el componente por primera vez.
Este constructor siempre tomará **props** como argumento, y la primera línea siempre será **`super(props);`** que configura el objeto basado en la clase `React.Component`.
A continuación, inicializamos el **`state`** del componente, que es un objeto `JavaScript` que almacena información sobre el componente. Por el momento, solo estableceremos el `count` en `0`.

```
constructor(props) {
    super(props);
    this.state = {
        count: 0
    };
}
```

React – Ejemplo del “Contador”

Ahora, podemos trabajar en la función de renderizado, donde especificaremos un encabezado y un botón. También agregaremos un detector de eventos para cuando se haga clic en el botón, lo que React hace usando el atributo `onClick`:

```
render() {
    return (
        <div>
            <h1>{this.state.count}</h1>
            <button onClick={this.count}>Contar</button>
        </div>
    );
}
```

React – Ejemplo del “Contador”

Finalmente, definamos la función de conteo. Para hacer esto, usaremos la función `this.setState`, que puede tomar como argumento una función del estado anterior al nuevo.

```
count = () => {
    this.setState(state => ({
        count: state.count + 1
    }))
}
```

Trabajemos Juntos

Llegó la hora del desafío:

- Intenta construir una aplicación JavaScript que te permita obtener el clima de una ciudad (temperatura, humedad, etc…) desde una API.
- Intenta construir la misma calculadora de la clase anterior pero con React.

¡HASTA LA *próxima!*

www.polotic.misiones.gob.ar

/poloticmisiones

polotic misiones

Gobierno de Misiones