Azúcar Sintáctico

Sugar Syntax refiere a la sintaxis agregada a un lenguaje de programación con el objetivo de hacer más fácil y eficiente su utilización. Favorece su escritura, lectura y comprensión, existe porque los lenguajes tienden a evolucionar.

Al implementarlos de manera adecuada podemos mejorar la legibilidad y pragmatismo de nuestro codigo

Ej:

i = 1+1 -> i++

ES5 función normal

```
var miFuncion = function(num) {
 returnnum + num;
}
```

ES6 función arrow más corta y fácil de leer

```
var miFuncion = (num) => num + num;
```

Clases

Ahora JavaScript tendrá clases, muy parecidas las funciones constructoras de objectos que realizabamos en el estándar, pero ahora bajo el paradigma de clases, con todo lo que eso conlleva, como por ejemplo, herencia. Aunque no deja de ser un azúcar sintáctico (Sugar Syntax) porque en JavaScript no tenemos clases, tenemos prototipos.

```
classLibro{
  constructor(tematica, paginas, autor) {
    this.tematica=tematica;
    this.paginas=paginas;
    this.autor=autor;
  }
  metodo() {
    // ...
```

```
    }
}
```

Let y Const

Ahora podemos declarar variables con let en lugar de var si no queremos que sean accesibles más allá de un á

ES5

```
(function() {
    console.log(x); // x no está definida aún.
    if(true) {
        var x = "hola mundo";
    }
    console.log(x);
    // Imprime "hola mundo", porque "var" hace que sea global
    // a la función;
})();
```

ES6

```
(function() {
    if(true) {
        let x = "hola mundo";
    }
    console.log(x);
    //Da error, porque "x" ha sido definida dentro del "if"
})();
```

Template Strings

Ahora con EcmaScript6 se puede concatenar de una simple manera las cadenas de string,para ello en vez de utilizar el signo de suma solamente se coloca el símbolo de dolar con llaves entre comillas invertida, ejemplo:

ES6

```
let nombre1 = "JavaScript";
let nombre2 = "awesome";
console.log(`Sólo quiero decir que ${nombre1} is ${nombre2`);
```

Valores por defecto

Los valores por defecto se puede pasar como valores de las variables en los parámetros de una función:

ES5

```
function(valor) {   valor = valor || "foo";
}
```

ES6

```
function(valor = "foo") {...};
```

Elevación de potencia:

Ahora en ES7 tenemos una nueva para calcular una potencia.

```
let base = 3;
let exponente = 2;
let potencia = base ** exponente;
console.log(potencia);
```

Webpack

Webpack nació a finales de 2012 de la mano de un solo desarrollador (Tobias Koppers, alemán), y en la actualidad es utilizado por miles de proyectos de desarrollo web Front-End: desde frameworks como React o Angular hasta en el desarrollo de aplicaciones tan conocidas como Twitter, Instagram, PayPal o la versión web de Whatsapp.

Webpack se define como un empaquetador de módulos (un bundler en la jerga habitual) pero que hace muchísimas cosas más:

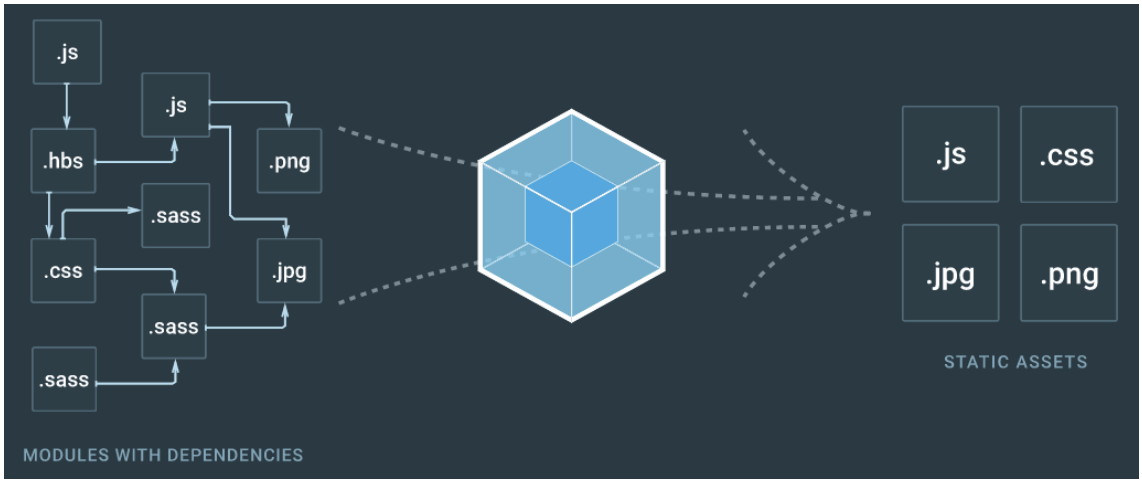
- Gestión de dependencias
- Ejecución de tareas
- Conversión de formatos
- Servidor de desarrollo
- Carga y uso de módulos de todo tipo (AMD, CommonJS o ES2015)

Y esto último lo que hace que destaque en especial. Es una herramienta extremadamente útil cuando desarrollas aplicaciones web diseñadas con filosofía modular, es decir, separando el código en módulos que luego se utilizan como dependencias en otros módulos. Una de las cosas que hace realmente bien Webpack es la gestión de esos módulos y de sus dependencias, pero también puede usarse para cuestiones como concatenación de código, minimización y ofuscación, verificación de buenas prácticas (linting), carga bajo demanda de módulos, etc...

Una de las muchas cosas interesantes de Webpack es que no solo el código JavaScript se considera un módulo. Las hojas de estilo, las páginas HTML e incluso las imágenes se pueden utilizar también como tales, lo cual da un extra de potencia muy interesante.

Webpack es una herramienta de compilación (una build tool) que coloca en un grafo de dependencias a todos los elementos que forman parte de tu proyecto de desarrollo: código JavaScript, HTML, CSS, plantillas, imágenes, fuentes... Esta idea central es la que lo convierte en una herramienta tan poderosa.

Webpack se puede considerar como un Task Runner muy especializado en el procesamiento de unos archivos de entrada para convertirlos en otros archivos de salida, para lo cual utiliza unos componentes que se denominan loaders.



Lo habitual es utilizar un archivo de código especial llamado webpack.config.js que está en la raíz de tu proyecto de desarrollo y que define mediante código JavaScript las operaciones que quieres realizar. En este archivo defines toda la información necesaria para poder utilizar Webpack para tus propósitos.

Generalmente las herramientas como Webpack (la más conocida: Browserify) se han utilizado en combinación con gestores de dependencias y task runners. Pero Webpack es tan potente que ya puede efectuar de serie, sin necesidad de nada más, casi cualquier tarea que puedas necesitar para tu desarrollo. En la actualidad está desplazando poco a poco el uso de otras herramientas.

Por ejemplo, no solo tiene soporte de serie para minimizar o combinar archivos, generar mapas de depuración o copiar recursos, sino que incluso ofrece un servidor web integrado con la capacidad de recarga automática cuando algo cambia o reemplazo en caliente de módulos. Las pocas cosas para las que no tiene soporte directo o a través de loaders o plugins se pueden conseguir con el uso de scripts npm, al fin y al cabo la herramienta está basada en Node.js y necesita npm para instalarse.

El mayor problema que ha tenido Webpack desde siempre es que la configuración es un tanto complicada, al menos al principio, lo que hace que tardes un poco en tenerlo funcionando para tu proyecto. Aunque en los últimos tiempos están haciendo lo posible para mejorar esto gracias a añadir funcionalidad en la interfaz de línea de comandos para partir de plantillas ya pre-configuradas.

Ventajas e inconvenientes de usar Webpack

El concepto del grafo de dependencias de Webpack es muy poderoso y ofrece muchísimos beneficios, como, por ejemplo:

Eliminación de recursos no utilizados: permite de manera sencilla obtener para producción únicamente aquellos recursos que son necesarios, dejando de lado los que no se utilizan. Esto incluye, por ejemplo, las reglas CSS que de verdad se están utilizando, dejando fuera las demás, lo cual es una utilidad muy potente.

Control absoluto sobre cómo procesas los recursos: para tomar decisiones avanzadas como que lleven un hash al final de su nombre en cada despliegue para que no queden

"atrapados" en ninguna caché o que las imágenes pequeñas se incluyan codificadas en Base64 en la página en lugar de descargarse (ahorrando peticiones al servidor). Cosas por el estilo.

Te permite utilizar cualquier tipo de sistema de modularización para JavaScript, sea AMD, CommonJS, ES2015 o Angular, sin preocuparte de que el navegador tenga que soportarlo (Browserify, por ejemplo, solo soporta CommonJS, que es el de Node.js).

Tienes casi cualquier cosa que puedas necesitar sin recurrir a otras herramientas, por lo que no necesitarás un task runner como Gulp o un gestor de dependencias como Bower.

Es muy rápido.

Despliegues confiables: utilizando correctamente los conceptos de Webpack no puede darse el caso de que te dejes cosas fuera a la hora de desplegar. Esto en aplicaciones grandes es mucho decir.

Aunque al principio te puede costar arrancar con Webpack en el proyecto, una vez que lo hagas ganarás mucha velocidad de desarrollo.

Pero Webpack también tiene inconvenientes:

La configuración es bastante compleja y cuesta, sobre todo al principio, aunque están intentando mejorarlo ofreciendo plantillas y validadores intergados.

Todo lo que utilices debe ser modular. No solo deberás escribir los archivos JavaScript como módulos, sino que las dependencias de otras bibliotecas de JavaScript de terceros que utilices deben ser modulares también (aunque hoy en día eso no es muy problemático).

En proyectos grandes tocar la configuración puede ser complejo.

Algunas sintaxis pueden ser algo confusas.

La documentación ha sido tradicionalmente mala, difícil de entender y poco clara en algunas cosas, aunque la van mejorando.

Si algo falla es difícil de saber qué está pasando, ya que no da mucha información sobre el evento.

El código fuente es muy complejo. Esto está relacionado con lo anterior. No es que tengas que ir mucho a ver el código fuente, pero si alguna vez lo necesitas para ver por qué algo falla o cómo funciona una determinada característica, no es lo más fácil de seguir del mundo.

El uso de Webpack puede llegar a ser complejo y, al igual que otras herramientas típicas de desarrollo web Front-End,como gestores de dependencias o task runners, para un proyecto pequeño quizá no tenga mucho sentido usarlo o quizá no aporte grandes beneficios de entrada. Sin embargo, en proyectos de tamaño mediano o grande los beneficios se ven inmediatamente.

Si vas a programar con React es casi obligatorio que utilices Webpack, pues es la herramienta que han adoptado para su desarrollo y la que usa todo el mundo en su comunidad.

¿Qué es Babel?

Babel es un "compilador" (o transpilador) para JavaScript. Básicamente permite transformar código escrito con las últimas y novedosas características de JavaScript y transformarlo en un código que sea entendido por navegadores más antiguos.

¿Por qué es necesario transformar el código JavaScript?

JavaScript es un lenguaje que no para de evolucionar y que cada año agrega nuevas características a su estándar. El estándar de JavaScript, conocido como ECMAScript.

ECMAScript (o abreviado ES) es la especificación en la que se basa JavaScript. ES nace de la organización ECMA International, cuyo objetivo es desarrollar estándares y reportes técnicos para facilitar el uso de tecnologías de la información

Esta organización se constituye de varios equipos siendo el TC39 el equipo encargado de revisar, proponer y definir las características que el estándar tendrá cada año.

Lamentablemente la velocidad de actualización del estándar no es la misma que la velocidad de adopción de los navegadores, es decir, los navegadores no siempre pueden implementar las nuevas características en sus engine, ni tampoco hacerlo retrocompatible con versiones más antiguas. Además, tampoco es posible asegurar que todos los usuarios estén utilizando navegadores modernos o de última generación lo que imposibilita tener la seguridad de que podamos hacer uso de las nuevas características del lenguaje en todas partes.

¿Qué ventajas tiene el utilizar las nuevas características propuestas por ECMA?

Las nuevas características y "habilidades" que JavaScript provee cada año son sobre todo ventajas para ti como desarrollador, ya que te proporciona de herramientas más poderosas y expresivas para implementar soluciones a problemas complejos. Algunas de las características se han ido agregando con los años son:

- Funciones flechas (arrow functions)
- Plantillas Literales (Template Literals)
- Map y Set
- Let y Const
- Clases

- Encadenamiento Opcional (Optional Chaining)
- Operador Fusión Nula (Nullish Coalescing)
- Métodos Privados (ES2021)
- Operador de Asignación Local (ES2021)

¿Cómo puedo usar las nuevas versiones de JavaScript?

Ya que no todos los navegadores pueden interpretar o entender las nuevas características del lenguaje que estás o quieres usar es necesario buscar algunas soluciones, una de ellas es el uso de polyfills o librerías que implementan con código "antiguo" el mismo comportamiento que lo que intentas expresar con características nuevas, un ejemplo de esto es el polyfill para usar Object.keys

Pero si necesitas usar algo más que un par de operadores entonces entra en juego más requerimientos y aquí es donde Babel hace su aparición

Babel permite transformar código de versiones nuevas de ES (ECMA2015+) a versiones retrocompatibles de JavaScript, esto lo realiza mediante:

- Transformaciones de sintaxis
- Polyfills
- Transformaciones de código fuente (codemods)

Babel es entonces una herramienta que lee tu código (archivo por archivo) y genera nuevo código que puedes usar en navegadores antiguos (o como entrada para algún bundler como webpack, rollup, etc)

Babel es casi omnipresente, utilizado por el 99.9% de los equipos que desarrolla software utilizando JavaScript, además Babel se alinea directamente a las discusiones del TC39 proveyendo así implementaciones de propuestas que aún ni siquiera son aceptadas por el comité, lo que te permite tener dichas funcionalidades mucho antes incluso que los navegadores.

¿Cómo funciona Babel?

Revisaremos de forma breve como babel funciona convirtiendo tu código de una versión a otra.

Considera este pequeño trozo de código JavaScript

```
const sum = (a, b) => a + b
```

Ahora considera que por alguna razón necesitas que este se ejecute en un navegador antiguo, entonces deberás transformarlo a

```
'use strict';

function sum(a,b) {
        return a + b;
```

}

¿Cómo se logra esto?

Es un proceso que se ejecuta en al menos 3 procesos

Parsing: Babel toma el código fuente, lo lee y lo convierte en una representación abstracta conocida como AST (Abstract Syntax Tree). Esta es una representación donde cada nodo del árbol representa una estructura del código fuente. Babel utiliza Babylon para este trabajo.

Transformación: En esta etapa Babel trabaja sobre el AST generado en el proceso anterior y lo manipula para generar un nuevo AST que represente el código necesario para el soporte seleccionado.

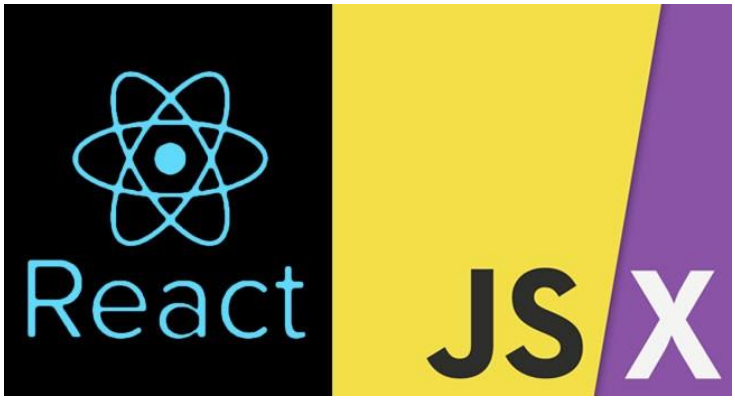
Este paso es realizado por una serie de "plugins" que permiten ejecutar varias transformaciones de forma sencilla y atómica. Aquí cada plugin toma el AST generado por el previo para aplicar una pequeña transformación.

Generación de código: En esta etapa, babel toma el AST generado y crea/escribe el código compatible, es decir, crea ese trozo de código soportado por navegadores más antiguos.

```
'use strict';

function sum(a,b) {
        return a + b;
}
```

JSX



Considera la declaración de esta variable:

const element = <h1>Hello, world!</h1>;

Esta curiosa sintaxis de etiquetas no es ni un string ni HTML.

Se llama JSX, y es una extensión de la sintaxis de JavaScript. Recomendamos usarlo con React para describir cómo debería ser la interfaz de usuario. JSX puede recordarte a un lenguaje de plantillas, pero viene con todo el poder de JavaScript.

JSX produce “elementos” de React.

¿Por qué JSX?

React acepta el hecho de que la lógica de renderizado está intrínsecamente unida a la lógica de la interfaz de usuario: cómo se manejan los eventos, cómo cambia el estado con el tiempo y cómo se preparan los datos para su visualización.

En lugar de separar artificialmente tecnologías poniendo el maquetado y la lógica en archivos separados, React separa intereses con unidades ligeramente acopladas llamadas “componentes” que contienen ambas.

React no requiere usar JSX, pero la mayoría de la gente lo encuentra útil como ayuda visual cuando trabajan con interfaz de usuario dentro del código Javascript. Esto también permite que React muestre mensajes de error o advertencia más útiles.

Insertando expresiones en JSX

En el ejemplo a continuación, declaramos una variable llamada name y luego la usamos dentro del JSX envolviéndola dentro de llaves:

```
const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>;


ReactDOM.render(
  element,
```

```
  document.getElementById('root')
);
```

Puedes poner cualquier expresión de JavaScript dentro de llaves en JSX. Por ejemplo, 2 + 2, user.firstName, o formatName(user) son todas expresiones válidas de Javascript.

En el ejemplo a continuación, insertamos el resultado de llamar la función de JavaScript, formatName(user), dentro de un elemento <h1>.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Try it on CodePen

Dividimos JSX en varias líneas para facilitar la lectura. Aunque no es necesario, cuando hagas esto también te recomendamos envolverlo entre paréntesis para evitar errores por la inserción automática del punto y coma.

JSX también es una expresión

Después de compilarse, las expresiones JSX se convierten en llamadas a funciones JavaScript regulares y se evalúan en objetos JavaScript.

Esto significa que puedes usar JSX dentro de declaraciones if y bucles for, asignarlo a variables, aceptarlo como argumento, y retornarlo desde dentro de funciones:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

Especificando atributos con JSX

Puedes utilizar comillas para especificar strings literales como atributos:

```
const element = <a href="https://www.codoacodo.com.ar"> link </a>;
```

También puedes usar llaves para insertar una expresión JavaScript en un atributo:

```
const element = <img src={user.avatarUrl}></img>;
```

No pongas comillas rodeando llaves cuando insertes una expresión JavaScript en un atributo. Debes utilizar comillas (para los valores de los strings) o llaves (para las expresiones), pero no ambas en el mismo atributo.

Cuidado:

Dado que JSX es más cercano a JavaScript que a HTML, React DOM usa la convención de nomenclatura camelCase en vez de nombres de atributos HTML.

Por ejemplo, class se vuelve className en JSX, y tabindex se vuelve tabIndex.

Especificando hijos con JSX

Si una etiqueta está vacía, puedes cerrarla inmediatamente con />, como en XML:

```
const element = <img src={user.avatarUrl} />;
```

Las etiquetas de Javascript pueden contener hijos:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX previene ataques de inyección

Es seguro insertar datos ingresados por el usuario en JSX:

```
const title = response.potentiallyMaliciousInput;
const element = <h1>{title}</h1>;
```

Por defecto, React DOM escapa cualquier valor insertado en JSX antes de renderizarlo. De este modo, se asegura de que nunca se pueda insertar nada que no esté explícitamente escrito en tú aplicación. Todo es convertido en un string antes de ser renderizado. Esto ayuda a prevenir vulnerabilidades XSS (cross-site-scripting).

JSX representa objetos

Babel compila JSX a llamadas de React.createElement().

Estos dos ejemplos son idénticos:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

React.createElement() realiza algunas comprobaciones para ayudarte a escribir código libre de errores, pero, en esencia crea un objeto como este:

```
// Nota: Esta estructura está simplificada
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

Estos objetos son llamados “Elementos de React”. Puedes pensar en ellos como descripciones de lo que quieres ver en pantalla. React lee estos objetos y los usa para construir el DOM y mantenerlo actualizado.