HenryLogo

Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

Módulos y Bundlers

Cuando desarrollamos, queremos que la estructura de nuestro programa/codigo sea lo mas transparente posible, facil de explicar y que cada parte cumpla una tarea definida.

Un programa típico crece de manera organica con el tiempo, nuevas piezas de funcionalidad van siendo agregadas a medida que surgen nuevas necesidades. Esto hace que dar una Estructura -y mantenerla mientras crece- sea fundamental. El problema que mantener esa estructura es un trabajo *extra*, el cual solo veremos los frutos en el futuro, cuando alguien nuevo trabaje en el proyecto.Por lo tanto, lo que puede terminar sucediendo, es que no se haga el trabajo extra y se deja que las partes del programa queden muy entreveradas entre sí.

Finalmente, aparecen dos problemas: el primero, es que entender un programa o sistema sin estructura clara es díficil. Si está tan entreverado que tocar una cosa puede impactar en el todo, al introducir cambios seguramente vas a crear muchos bugs que trendas que corregir. O sea que no podés trabajar de manera aislada una sóla parte del código. Finalmente, te ves obligado a construir un entendimiento del código como un todo. Segundo, si quisieras **reutilizar** una parte de código en otro proyecto, es muy probable que *reescribir* esa funcionalidad sea más fácil que lograr extraerla de tu programa complejo.

Modules

Los **módulos** son un intento de evitar estos problemas. Un **Módulo** es un pedazo de código que cumple una tarea específica y que indica sobre qué piezas de código depende (*dependencias*).

> **Interfaz** es lo que conocemos en inglés como interface (“superficie de contacto”). En informática, se utiliza para nombrar a la conexión funcional entre dos sistemas, programas, dispositivos o componentes de cualquier tipo, que proporciona una comunicación de distintos niveles permitiendo el intercambio de información. Su plural es interfaces.

Estos módulos proveen una interfaz de contacto hacia afuera, es decir que todo el funcionamiento del mismo está encapsulado del mundo externo, y sólo se permite interactuar con el módulo a través de puntos de contactos bien definidos y documentos (en el mejor de los casos).

Es muy similar a cuando interactuamos con un objeto, como un `Array` y usamos sus métodos:

```
var arreglo = [];
arreglo.push(1);
```

En el ejemplo, nosotros usamos la función `push` pero no tenemos idea como está implementada adentro, gracias a la documentación sabemos cómo funciona y cómo usarla, pero su funcionamiento está encapsulado dentro de la función.

Dependencias

Las relaciones entre módulos se llaman **dependencias**. Cuando un módulo necesita una parte de código de otro módulo, vamos a decir que ese módulo *depende* del segundo. En general los módulo van a especificar qué otros módulos son sus dependencias, de tal forma que para usarlo podemos cargar todas esas dependencias.

Encapsulando Código

En `js`, para lograr esta encapsulación vamos a necesitar que cada módulo tenga su propio **scope**.

Poner nuestro código en diferentes archivos no es suficiente, ya que si cargamos varios archivos, todos comparten el mismo contexto global. Por lo tanto podría haber colisiones entre módulos e interferir entre ellos, rompiendo el encapsulamiento.

Paquetes

Bien, ahora imaginemos que intenamos encapsular el código para poder usarlo como una pieza en nuestro proyecto (más abajo veremos cómo se hace). Cuando llegue el momento de usar ese código en otro proyecto, lo que hagamos, probablemente, es copiar ese código y *reutilizarlo*. Imaginemos ahora, que en el nuevo proyecto detecto un bug en el código y decido corregirlo. Ahora tambien debería ir al proyecto viejo y corregirlo tambien. Como se pueden imaginar, esto no se puede escalar. Cuando el número de proyectos en el que usamos ese código crezca, va a ser inmanejable la tarea de updatear cada pedazo de código en cada proyecto manualmente.

La solución a este problema son los **paquetes**. Un **paquete** es un pedazo de código que puede ser distribuido (copiado e instalado). Cada paquete puede contener uno o más módulos y a su vez tiene información sobre las dependencias que tiene con otros paquetes. Generalmente, estos paquetes viene acompañado de documentación que indican al usuario cómo usarlos y qué hacen.

Cuando un error es encontrado en algún paquete, o se le agrega funcionalidad nueva. Es corregido y updateado. Ahora los proyectos que dependen de ese paquete pueden actualizar esos paquetes a la nueva versión.

Para lograr distribuir estos paquetes y mantenerlos correctamente actualizados, vamos a necesitas la ayuda de un **gestor de paquetes**. El gestor de paquetes es un pedazo de software que se encarga de manejar esto de manera automática. En el mundo de JavaScript, el gestor de paquetes más usados es NPM (https://npmjs.org).

NPM es un servicio online en donde estan hosteados los paquetes que los usuarios comparten, y a su vez un programa que se puede instalar en cualquier SO, que te ayuda a descargarlos, instalarlos y mantenerlos actualizados.

> Veremos NPM más en detalle en el módulo de Back-End.

Creando Módulos

Hasta 2015, JavaScript no tenía una forma *nativa* de construir módulos. Pero de todos modos, las personas lo usasron para construir grandes proyectos a lo largo de diez años. Por lo tanto, los desarrolladores crearon su propia forma de crear módulos en JavaScript. Lo lograron usando funciones para crear scopes isolados, y usaron objetos para crear las interfaces de los módulos.

Vamos a crear un módulo que nos ayude a trabajar con fechas, va a tener dós métodos que nos permiten pasar un Integer y recibir el nombre del día, y al revés.

```
const weekDay = function() {
  const names = ["Domingo", "Lunes", "Martes", "Miercoles",
                 "Jueves", "Viernes", "Sabado"];
  return {
    name: function name(number) { return names[number]; },
    number: function number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Domingo")));
// → Sunday
```

La interfaz está creada en el objeto que retornamos. Que tiene los dos métodos antes mencionados. Lo interesante es notar, que se logro encapsular el código a traves de un **IIFE** (Inmidiatly invoked function expression), y creando un closure con el arreglo `names`. De esta manera, en el scope global, si hubiera una variable `names` no interfiere con la que declaramos en nuestro módulo.

> Pensá que ocurriría si no hubiesemos creado el close con al IIFE. ¿Qué pasaría si alguien usa nuestro código y en su proyecto ya tenía declarada una variable `names`?

Este tipo de solución solo ofrece isolación, pero no habla de dependecias, solamente pone su interfaz en el contexto global (el objeto `weekDay`). Por mucho tiempo, esta fue la forma de programar módulos en la web.

Mejorando los módulos

Una siguiente mejora lógica para nuestros módulos, sería poder tenerlos en archivos separados, por ejemplo, tener nuestro módulo de los días en el archivo: `weekDay.js`, y tener alguna manera de *importarlo*.

Para hacer eso deberíamos tener la capacidad de leer el contenido de un archivo ('strings') y poder pasarla al interprete para que la ejecute. Hay varias formas de lograr esto en JS.

La primera es usando una función especial de JS llamada `eval`. Básicamente esta función recibe un `string` como parámetro y va a ejecutar el código en el scope actual (como si lo estuvieras copiando y pegando ahí).

```
const x = 1;
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
```

```
// → 2
console.log(x);
// → 1
```

Este método no es muy efectivo, como vemos esta función puede romper el sistema de scopes tradicional, por lo tanto no es muy predecible.

Otra forma, más predecible, es usar el constructor de `Function` (es el constructor que JS utiliza internamente para crear funciones). Este recibe dos argumentos: una string que contiene una lista separada por comas de argumentos, y una string que contiene el cuerpo de la función.

```
// estas dos formas producen la misma funcion
function plusOne(n) {
    return n +1;
}

let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

Utilizando esto, vamos a poder encapsular un módulo dentro de una función y usar el scope de esa función como el scope del módulo.

CommonJS

El sistema que finalmente elegió NodeJS (y por lo tanto casi haciendo que sea standart) es conocido como **CommonJS**. Es el sistema de módulos utilizado por la mayoría de paquetes de `npm`.

El concepto más importante de CommonJS es una función llamada `require`, que recibe una string que indica el nombre de una dependencia. Cuando es invocada, esta función busca el módulo, lo carga y retorna la interfaz de ese módulo. Esta función, además, envuelve todo el módulo en una función, por lo tanto cada módulo automaticamente tiene su propio scope.

Para pasar el módulo que hicimos antes a CommonJS, básicamente vamos a necesitar utilizar un objeto llamado `exports`. En CommonJS este objeto es donde debemos poner todo lo que queremos que esté en la interfaz de nuestro módulo, es decir, todo lo que querramos que el mundo exterior pueda ver.

Siguiendo con el ejemplo del módulo de las fechas, veamos como podríamos hacer un verdadero módulo usando CommonJS: Primero creamos un archivo con nombre `weekDays.js` con el siguiente contenido:

```
var names = ["Domingo", "Lunes", "Martes", "Miercoles",
                "Jueves", "Viernes", "Sabado"];

exports.name = function name (number) { return names[number]; };
exports.number = function number(name) { return names.indexOf(name); };
```

Para usarlo, tenemos que usar `require` en el archivo que quisieramos utilizar nuestro módulo:

```
var weekDays = require('./WeekDays.js');

console.log(weekDay.name(weekDay.number("Domingo")));
```

Viendo esto, podríamos imaginar cómo funciona `require` por adentro:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name); // funcion que lee un archivo de texto
    let module = {exports: {}};
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module); // pasa la funcion require por si es
necesario usarla adentro (otras dependencias)
  }
  return require.cache[name].exports;
}
```

> En este código `readFile` es una función inventada que lee un archivo de texto y retorna su contenido como una string. JS no tiene una función tal, pero el ambiente donde se ejecuta el motor (Node o el browser), puede proveer la forma de leer archivos a JS.

Para evitar tener que cargar el mismo módulo muchas veces, `require` tiene un *cache* de módulos que ya fueron cargados. Si el módulo ya fue invocado, estará en el objeto `cache`, si no, leerá el código del módulo, lo envolverá en una función e lo invocará.

Que son los Bundlers?

Bien, ahora que sabemos algo sobre módulos, veamos cómo estos revolucionaron la forma de escribir código para el front-end con la introducción de los bundlers.

Como sabemos, la forma de importar librerías (que a su vez son módulos) en HTML es la siguiente:

 scripts

Si pensamos en detalle que sucede cuando importamos cada uno de esos scripts, veremos que básicamente todos terminan cayendo al mismo contexto, el global. Para salvar esto, las librerías básicamente elegían arbitrariamente un nombre de variable donde exponer su funcionalidad. Por ejemplo, `jQuery` utilizaba el signo `$`. Ahora bien, si otra librería decidía utilizar el mismo nombre de variable para su interfaz, tendríamos un conflicto, y ambas librerías no podrían ser usadas en el mismo HTML.

Al principio, sólo se importaba una cantidad pequeñas de librerías para el front, por lo tanto esto no era un problema tan grande. pero a medida que la complejidad del front fue aumentando, y la cantidad de librerías tambien, se tuvo que pensar una nueva forma para resolver esto.

Acá aparecieron los **Module Bundlers**. Como por ejemplo: `Browserify`, `Webpack`, `Rollup`, etc... Básicamente lo que hacen es ejecutar un proceso que lee todas las dependencias de nuestro proyecto, y luego genera un archivo JS que contiene todos los módulos necesarios que podemos incluir en nuestro HTML.

Hay dos cosas etapas en tarea de un Bundler:

- Resolución de dependecias
- Empaquetamiento

Entrando desde un entry point (nuestro archivo `.js` principal), el objetivo de la resolución de dependencias es buscar todas las dependencias del código y construir un grafo (llamado grafo de dependencias).

Una vez hecho esto, podés empaquetar o convertir todo tu grafo de dependencias en un sólo archivo que tu aplicación va a usar. Finalmente, obtenemos un archivo único (el **bundle**) que vamos a importar en nuestro HTML. De esta forma, resolvemos los problemas de encapsulamiento que mencionamos anteriormente.

> Cuando veamos *React*, vamos a aprender a usar el bundeler *webpack*.

Homework

Completa la tarea descrita en el archivo README