**M2S12: Modularización de aplicaciones con NodeJS**

**Hola, ¿cómo estás?, nos encontramos nuevamente en la semana 12 de tu Bootcamp; en esta oportunidad aprenderás sobre Patrones de Diseño; recuerda que debes estar en un lugar lo más tranquilo posible e iluminado para que puedas estudiar con calma y en caso de que tengas alguna duda, recuerda que tus coaches están allí para ayudarte.**

**Sigamos con el contenido de esta semana. Que tengas mucho éxito.**

**ÍNDICE**

* [Patrones de Diseño](https://github.com/U-Camp/BOOT-M2-SEM12#patrones-de-dise%C3%B1o)
* [Patrón Modular](https://github.com/U-Camp/BOOT-M2-SEM12#patr%C3%B3n-modular)
  + [require](https://github.com/U-Camp/BOOT-M2-SEM12#require)
  + [Exportación](https://github.com/U-Camp/BOOT-M2-SEM12#exportaci%C3%B3n)
  + [Importación](https://github.com/U-Camp/BOOT-M2-SEM12#importaci%C3%B3n)

**Patrones de Diseño**

Un patrón de diseño se define como una solución reusable a un problema en común dentro de un contexto específico, esto quiere decir que no hay que reinventar la rueda, en la mayoría de los casos ya hay soluciones probadas a determinados problemas y una arquitectura preestablecida que la implementa, por lo que no es necesario reimplementar, sólo conocer esas soluciones a problemas comunes y saber dónde implementarlas.

Los patrones de diseño, como ya se dijo, son soluciones ya probadas a problemas que son comunes a una arquitectura, en el desarrollo de software se aplican para solucionar problemas en la estructura de los objetos, por ejemplo si se ocupa de la funcionalidad de un objeto particular y éste objeto se va a usar en diferentes partes de la aplicación, lo ideal sería no instanciar el objeto en todas y cada una de las partes en las que se va a usar ya que cada vez que se instancia un objeto este ocupa espacio en memoria, por lo que si se aplica un patrón Singleton25 , sólo se instancia un sólo objeto y este mismo objeto es accesible y reusable en todos los lugares en donde se necesite.

Además de singleton, hay varios patrones de diseño que se pueden aplicar a problemas comunes de la arquitectura de software y son los siguientes:

* ***MVC. Model-View-Controller***: hace referencia a dividir las responsabilidades de cada objeto (Single Responsibility Principle ) en un modelo que se encargará de obtener los datos necesarios de la aplicación, un controlador que se encargará de procesarlos y una vista que se encargará de mostrar los resultados del proceso, por lo que la separación de concerns queda definida y hace más mantenible el código.

Este no es realmente un patrón de diseño, ya que se le considera más como una estructura base de arquitectura de software, la cual es de las más usadas.

La arquitectura Frontend <-> Backend <-> Base de Datos trata de replicar este formato a gran escala (siendo el frontend la vista, el backend el controlador y la base de datos el modelo), así mismo dentro de cada uno de ellos se aplica el mismo formato para organizar las clases y los archivos que las definen.

* ***Lazy Loading***: No todos los recursos u objetos se ocupan al momento de cargar una aplicación, por ejemplo, una imagen que se encuentre hasta el pie de una página web no es necesaria para iniciar la aplicación en sí, por lo que se puede cargar de manera perezosa, esto quiere decir que primero se carga lo importante o se cargan los recursos de manera priorizada, de tal manera que la aplicación sea funcional y luego se pueden ir cargando más recursos no tan importantes diferidamente (deferred).
* ***Prototype***: Al momento de instanciar un objeto, normalmente lo que se hace es usar la palabra reservada new para crear una nueva instancia de la clase, pero eso ocasiona que las propiedades de esa instancia en particular no pueda cambiar nunca, está atada a la estructura de la clase que se definió, el patrón Prototype hace más flexible la estructura de las clases, ya que establece que en lugar de instanciar por medio de new se instancie clonando, lo que la convierte en un prototipo al cual se le pueden agregar o eliminar propiedades o métodos en tiempo de ejecución el código lo vaya necesitando. Javascript es un lenguaje orientado a prototipos, esto quiere decir que cuando instanciamos un objeto en JS, realmente lo que se está haciendo es clonar la clase e instanciar ese clon, eso ya lo hace internamente Javascript al momento de usar la palabra reservada new.
* ***Decorator***: Cuando los objetos que se crean son estrictamente estáticos, lo que quiere decir que éstos siempre mantendrán la funcionalidad definida en su clase y no permiten nuevas funcionalidades una vez ya creados, se usan decoradores para extender la funcionalidad de un objeto en tiempo de ejecución, por lo que este patrón otorga flexibilidad al objeto en sí, al permitir agregar propiedades o comportamientos nuevos conforme se vayan usando. Al sere Javascript un lenguaje orientado a prototipos, cuando a un objeto que se haya creado se le agregan o modifican propiedades o métodos que no tienen definidos en su clase base, se dice que se está decorando ese objeto.
* ***Iterator***: Muchas estructuras de datos, como los arreglos, son iterativas, esto quiere decir que se puede recorrer cada elemento dentro de la estructura de manera cíclica, ya sea para buscar un elemento en particular o para modificar la misma estructura del elemento y generar nuevas estructuras a partir de ésta (inmutabilidad), un iterador hace precisamente esto, provee métodos que recorren una estructura de datos de tal manera que se puede acceder a sus elementos de una manera estándar. En Javascript el prototipo Array ya implementa el patrón y provee métodos como map, filter o reduce, los cuales se usarán durante el proceso de escritura de código de los proyectos y actividades.
* ***Composite***: Cuando se requiere que un objeto tenga objetos hijos y éstos a su vez tengan otros objetos hijos, puede que el árbol de jerarquías sea muy grande y difícil de seguir en el código, por lo que el patrón Composite provee la funcionalidad de crear componentes a partir de varios objetos relacionados entre sí, de tal manera que queden agrupados en un componente padre, el cual se encargará de manejar y pasarle propiedades a todos los objetos hijos. React hace uso extensivo de este patrón, lo cual se verá durante el curso.
* ***State***: Como ya se mencionó, un objeto tiene estado y comportamiento definidos, el patrón state permite que el comportamiento dependa del estado actual del objeto, esto quiere decir que con cada cambio que el estado tenga, se generen comportamientos distintos, por ejemplo cuando el usuario introduce datos a una caja de texto, su estado puede cambiar y se puede validar la entrada al momento de que el usuario escriba, cambiando su comportamiento para dibujar un borde rojo y un mensaje de error si la entrada del usuario no es correcta o dibujando un borde verde si pasa el proceso de validación.
* ***Observer***: En ocasiones un objeto cambia su estado y probablemente muchos otros objetos dependen de ese estado en particular, por lo que es necesario que los objetos intercomunicados sepan en el momento en que uno de ellos cambió su estado, para que éstos se adapten al mismo y así mismo cambien su estado si es necesario, por lo que el patrón observer provee el mecanismo para estar observando cambios de estado sobre objetos en tiempo de ejecución y comunicar a objetos dependientes del cambio de estado en el momento en el que éste ocurra. Los exploradores web usan este patrón para manejar parte del sistema interno de eventos, así mismo internamente React hace uso del patrón state y el patrón observer para comunicar cambios del estado entre los componentes y realizar cambios en las propiedades y comportamiento de los hijos, ajustando el estado de cada componente de esa manera, a este proceso React le llama reconciliación.
* ***Adapter***: Muchas veces una aplicación necesita consumir un servicio o una librería que tiene otra forma diferente de comunicarse, esto quiere decir que usa otro formato de comunicación (por ejemplo un servicio que responda en XML 37 y la aplicación sólo usa JSON), como en la vida real que muchas veces necesitamos adaptadores para conectar ciertos electrónicos a la red eléctrica de nuestras casa, así mismo muchas veces ocupamos adaptadores que comunican dos aplicaciones que tienen distinto protocolo de transmitir información, por lo que el patrón adapter abstrae la forma en la que se comunican los dos componentes y es el encargado de adaptar los datos de tal manera que sean entendibles para ambos componentes (por ejemplo, convirtiendo los datos en XML a un formato JSON o al revés); éste tipo de patrón se aplica normalmente cuando se consumen APIs REST o servicios web de terceros.
* ***Proxy***: Una de las formas de consumir una API REST o servicios web de terceros es realizando peticiones directamente a los endpoints de cada una de ellas, esperando el resultado del proceso y pasando los datos del resultado a un adapter, hay recursos públicos que pueden consumirse directamente desde el frontend sin ningún problema, pero muchas veces el recurso está protegido contra accesos indebidos y no permite el consumo directo, sino que necesita una forma de saber que el frontend que lo consume está autorizado a hacer ésta petición, por lo que se usa un proxy para realizar este tipo de peticiones seguras, ya que normalmente requieren del uso de tokens de autorización que no deben ser vistos o compartidos directamente en el frontend, por lo que se crea un objeto del lado del servidor con los permisos necesarios para consumir los recursos y a su vez escucha las peticiones realizadas desde el frontend, consume los recursos protegidos y devuelve los datos del resultado al frontend. La librería axios que se usará durante el curso para consumir la API REST del backend puede implementar este patrón en las peticiones que se le indique.
* ***Publisher-Subscriber***: El patrón publisher-subscriber es parecido al patrón observer, pero normalmente se implementa para suscribirse a un servicio de stream de datos, esto quiere decir un servicio web o API que siempre esté mandando información o que comunique cambios internos, se puede ver como un mecanismo de escucha que siempre está conectado al flujo de datos, por ejemplo las notificaciones en una app web (como gmail), son escuchadas por medio de un subscriber el cual está observando cambios en el flujo de datos (publisher) y pasa esos cambios a la aplicación para que maneje los datos que cambiaron de forma conveniente.
* ***Front Controller***: Este patrón de diseño se usa principalmente en aplicaciones frontend o móviles, ya que es usado para centralizar y manejar cualquier tipo de petición a la API REST del backend o a servicios de terceros, esto quiere decir que un sólo objeto se tiene que encargar de manejar y abstraer ésta funcionalidad, construyendo los modelos de datos que la aplicación va a usar. Se usa en conjunto con el patrón proxy, observer y adapter; envolviéndolos en un objeto que aplique estos patrones. Para poder tener una buena organización del frontend, React requiere este tipo de patrón para poder componentizar correctamente y separar la lógica de obtención de datos (peticiones directas por medio del proxy o aplicando subscribers) de la lógica de procesamiento de estos (adapter) y la presentación del resultado (MVC).
* ***Façade***: Los sistemas grandes están formados por módulos o subsistemas, façade provee el estándar en la que cada módulo se va a comunicar con cualquier otro módulo, abstrae la mayor parte de los métodos de un sistema y provee una interfaz de métodos públicos de ese módulo en particular (fachada del módulo) que cualquier otro módulo puede usar. En sistemas grandes y complejos, aplicar este patrón se hace necesario, ya que en equipos de desarrollo grandes donde cada equipo se encargue de desarrollar un módulo del sistema en particular, debe proveer una interfaz fachada, que dé acceso al mismo desde otros módulos, normalmente se aplica para sistemas muy grandes en el backend.
* ***Dependency Injection***: Normalmente un objeto depende de otros objetos para realizar su labor, por ejemplo, un objeto que se comunica con un servicio, puede depender de un adapter al cual pasarle los datos una vez que éstos se obtienen, por lo que para que el objeto pueda hacer eso, es necesario que se inyecte una instancia de éste adapter de tal manera que cuando el objeto lo necesite, éste no tenga que preocuparse por instanciarlo, sino simplemente usarlo. El proceso de inyección lo realiza un proveedor de dependencias o provider, el cual es el encargado de instanciar los objetos necesarios e inyectarlos dentro del objeto que los necesita. El uso de éste patrón habilita que el objeto dependiente se ocupe de usar la interfaz de su dependencia no importando que tipo sea ésta, lo que facilita la implementación y las pruebas unitarias, ya que de esta manera podemos inyectar un objeto falso (mock50 ) que implemente una interfaz falsa del objeto dependiente, por ejemplo en un objeto que use axios, podemos inyectar un objeto simulado de axios para que las llamadas a los servicios no se hagan realmente y para probar sólo la funcionalidad en sí.
* ***Abstract Factory***: Algunas aplicaciones necesitan dependencias de la plataforma donde se estén ejecutando, por ejemplo el acceso al sistema de archivos o a distintas bases de datos, pero normalmente éste tipo de accesos la plataforma no las encapsula y su manejo y acceso es diferente entre sí (por ejemplo el sistema de archivos de Linux es EXT3 y el de Windows es NTFS y los dos guardan archivos de manera distinta), lo que quiere decir que al estar desarrollando este tipo de aplicaciones que requieren esos accesos, se tienen que validar sobre que plataforma se está ejecutando la aplicación y ejecutar las instrucciones necesarias para realizar las acciones que requiera la aplicación específicas para esa plataforma por lo que el código puede finalizar teniendo muchas validaciones (ifs anidados) y siendo menos mantenible, Abstract Factory resuelve ese problema ya que al implementarlo provee una fábrica abstracta de objetos de la misma familia que realizan una acción determinada sobre una condición específica, por lo que al final la aplicación en sí no sabe qué clase concreta está usando, sino simplemente accede a la interfaz y se comunica a través de ella directamente, delegando a la fábrica la responsabilidad de proveer el objeto de acceso que mejor convenga para esa acción. La librería de manejo de archivos File de NodeJS 52 implementa este patrón al igual que cualquier ORM 53 de acceso a base de datos.
* ***Object Pool***: Cuando el performance de una aplicación es muy bajo, el costo en memoria que requiere instanciar objetos es alto y se requiere acceder a esos objetos muy seguido, se puede introducir una pool de recursos en la cual todos los objetos vivirán durante su ciclo de vida, esa pool puede ser accedida desde varios puntos de la aplicación y acceder a los objetos ya creados sin necesidad de volver a instanciarlos, usando la misma instancia siempre, eso es una object pool y se puede ver como un conjunto de singletons para muchos objetos. Las conexiones de acceso a base de datos normalmente son un pool de objetos de conexión aplicando este patrón, por lo que se aplica entre Mongoose y MongoDB.

Se puede ver que todos estos puntos forman un proceso definido y relacionados entre sí: solucionar un problema por medio de un algoritmo (en un lenguaje de programación) estructurar como se implementará esa solución (paradigma de programación) y establecer cuál va a ser la arquitectura dentro del sistema reusando soluciones ya probadas (patrón de diseño) es un proceso interactivo e iterativo, esto quiere decir que se debe repetir tantas veces como sea necesario por cada problema que el sistema requiera solucionar, a éste proceso de repeticiones se le llama ciclo de vida de desarrollo de software, el cual se compone de los siguientes pasos:

1. **Análisis**: Establecer el problema a solucionar y a manera general el algoritmo que solucionará ese problema en particular.
2. **Diseño**: Plasmar de forma visual o escrita la solución, especificando patrones de diseño si se necesitan y que se usarán para llegar a la solución.
3. **Implementación**: Escribir el código en un lenguaje de programación, siguiendo los principios y estándares definidos en el diseño.
4. **Pruebas**: Realizar las pruebas necesarias que verifiquen que la solución fue implementada correctamente y que realmente resuelve el problema y no introduce nuevos.
5. **Despliegue**: Subir la solución implementada y sin errores a la plataforma donde se vaya a ejecutar.

Los pasos de éste proceso se pueden repetir tantas veces sea necesario dependiendo de la cantidad de problemas o requerimientos que un sistema de software en particular tenga que solucionar, hay distintos marcos de trabajo que implementan éste proceso, el más usado actualmente es SCRUM, en el cual todos los problemas que tiene que solucionar un sistema, se especifican desde el principio (backlog) y se van armando grupos de tareas (epics) las cuales se van solucionando poco a poco en cada iteración (sprint), por lo que en cada iteración se va construyendo el sistema incrementalmente, desplegando las soluciones a éstos pequeños problemas de manera ágil y rápida, por lo que al ir armando el sistema de manera incremental, se va entregando valor de forma expedita.

Como se puede observar, todos estos conceptos son universales al desarrollo de software, por lo que se aplican y adaptan a cualquier lenguaje de programación y se usan para construir software de calidad, mantenible y ordenado. Durante el desarrollo del curso se irá ahondando más en estos conceptos teóricos a través de la práctica, para ello se usará el lenguaje de programación JavaScript, los paradigmas de programación antes descritos, aplicando los patrones de diseño que se requieran en cada actividad a realizar.

**Hola, espero te encuentres bien y estes aprendiendo mucho. Si deseas consultar y descargar una infografía que resume un poco lo visto sobre patrones de diseño [presiona aquí](https://github.com/U-Camp/BOOT-M2-SEM12/blob/main/infografias/Patrones%20de%20dise%C3%B1o.pdf)**

**Sigamos adelante con Patrón modular.**

**Patrón modular**

El patrón modular permite mantener unidades de código para tener una mayor organización y limpieza.

Se promueve el concepto de encapsulamiento, el cual significa que las variables y funciones se mantienen privadas dentro de cada módulo y no pueden ser sobreescritas.

let module1 = (function () {

// ÁREA PRIVADA

const edad = 4

const nombre = 'Pulgas'

let raza

const establecerRaza = function (razaEstablecida) {

raza = razaEstablecida

}

const generarRaza = function (razaGenerada) {

establecerRaza(razaGenerada)

return raza

}

// ÁREA PÚBLICA

return {

raza: raza,

edad: generarRaza

}

}())

console.log(module1.edad("Husky")) // => "Husky"

Observa que, bajo este patrón, estamos permitiendo disponer variables públicas y privadas.

Ahora bien, esto puede estar sucediendo en un mismo archivo, pero, ¿qué pasa si quisiéramos utilizar múltiples archivos donde pudiéramos hacer una división mucho más clara en público y privado?

Utilizaremos el patrón modular dentro de ES6+, en las versiones modernas de los estándares de JavaScript.

Para poder aplicarlo, es necesario tener instalado Node.js

Vamos a aplicarlo.

**require**

require es una palabra reservada que permite requerir un módulo de un archivo específico.

Crea un archivo index.js y otro llamado sum.js

index.js

const sum = require("./sum")

const result = sum(20, 30); // 50

console.log(result)

sum.js

const name = "Mike"

const sum = (num1, num2) => num1 + num2;

module.exports = sum

En el momento que ejecutamos index.js, estamos utilizando require para conectar un archivo externo y extraer una variable o una función.

Estamos aplicando la privacidad de un módulo porque nosotros no podemos acceder a name de sum.js a menos que nosotros lo permitamos a través del module.exports

Hablemos a continuación de este último concepto.

**Exportación**

Para realizar una exportación de variables y funciones a través de un modulo, utilizamos la palabra reservada module.

Dentro, agregaremos datos a través de la propiedad exports, el cual incluirá un objeto con todo lo que queramos dejar disponible al público.

Cada vez que liberes un módulo, esté disponible a otros archivos JS, recuerda compartir las variables y funciones específicas.

**Importación**

Para poder conectarse a un módulo y extraer los datos, utilizaremos require para conectarnos al archivo específico y lo asignaremos a una variable.

Si nos encontramos en un ambiente de desarrollo con ES6+, podríamos sustituir el require por la palabra import.

Para realizar múltiples importaciones de diferentes archivos, recuerda utilizar require con los módulos que necesites.

const sum = require('./sum.js')

**Hasta aquí el contenido de esta semana, recuerda que debes tener instalado NodeJs para que puedas ejecutar lo que se enseño en este módulo.**

**Nos vemos la próxima semana.**