



Clase 39. Programación Backend

Arquitectura del servidor: Diseño



OBJETIVOS DE LA CLASE

- Conocer acerca de las buenas prácticas al desarrollar proyectos de Node.
- Conocer acerca del patrón MVC y cómo funciona.
- Comprender las diferencias entre patrón MVC: HTML on wire y Data on wire.
- Conocer acerca de los patrones de diseño.

CRONOGRAMA DEL CURSO

Clase 38



Arquitectura de capas

Clase 39



**Arquitectura del
servidor: Diseño**

Clase 40



**Arquitectura del
servidor: Persistencia**

BUENAS PRÁCTICAS EN EL DESARROLLO DE SERVIDORES NODE



Arquitectura del proyecto



- Tener un buen punto de partida cuando se trata de la arquitectura de nuestro proyecto es vital para la vida del proyecto en sí y cómo podremos abordar las necesidades cambiantes en el futuro. Una arquitectura de proyecto mala y desordenada suele llevar a:
 - Código ilegible y desordenado, que prolonga el proceso de desarrollo y hace que el producto sea más difícil de probar.
 - Repetición inútil, que hace que el código sea más difícil de mantener y administrar.
 - Dificultad para implementar nuevas funciones sin estropear el código existente.



Arquitectura del proyecto



- El principal objetivo de cualquier estructura de proyecto de Node es ayudarnos a:
 - Escribir código limpio y legible.
 - Escribir fragmentos de código reutilizables en nuestra aplicación.
 - Evitar la repetición.
 - Agregar nuevas funciones sin interrumpir el código existente.

PROYECTOS DE NODE



¿De qué se trata?



- Internet está lleno de documentación que cubre los conceptos básicos del desarrollo web. Pero por lo general, la información sobre las mejores prácticas es algo que aprendemos a lo largo del camino, a medida que creamos más aplicaciones, fallamos y tenemos éxito en el camino.
- Vamos a ver los aspectos más importantes del desarrollo web en un conjunto de puntos a tener en cuenta, al crear aplicaciones web con Node.
- Estos puntos aportan información sobre cómo determinadas decisiones de diseño pueden generar grandes dividendos en el transcurso del ciclo de vida de su desarrollo web.

ENFOQUE POR CAPAS



Adoptar un enfoque por capas



- Los frameworks como **Express**, nos permiten definir controladores de ruta como funciones de devolución de llamada, que se ejecutan cuando recibimos una solicitud de cliente. Con la cantidad de flexibilidad que brindan estos frameworks ¿Podríamos definir toda la lógica de negocio directamente dentro de esas funciones?
- Si comenzamos en este camino, el archivo de rutas de nuestro pequeño servidor puede convertirse en un código largo, desordenado, difícil de manejar, de leer, mantener y administrar, dificultando también, realizar una prueba unitaria.



Adoptar un enfoque por capas



- Por lo tanto, este sería un buen lugar para implementar el principio de programación de "separación de responsabilidades". De acuerdo con esto, deberíamos tener diferentes módulos para abordar diferentes inquietudes pertinentes a nuestra aplicación.
- Del lado del servidor, las diferentes capas (o módulos) asumen diferentes responsabilidades al procesar las solicitudes de cliente. Como ya vimos, las capas son:
 - **Controlador:** Rutas de API y endpoints.
 - **Capa de servicios:** para la lógica de negocio.
 - **Capa de acceso de datos:** para trabajar con la base de datos.



Crear una estructura de carpetas



- Todo tiene que tener su lugar en nuestra aplicación, y una carpeta es el lugar perfecto para agrupar elementos comunes.
- Esto proporciona claridad sobre qué funcionalidad se administra y dónde, permitiéndonos organizar nuestras clases y métodos en contenedores separados que son más fáciles de administrar. A continuación se muestra una estructura de carpetas común que podemos usar como plantilla al configurar un nuevo proyecto de Node:

```
src
├── app.js                app entry point
├── /api                  controller layer: api routes
├── /config                config settings, env variables
├── /services              service layer: business logic
├── /models                data access layer: database models
├── /scripts                miscellaneous NPM scripts
├── /subscribers            async event handlers
└── /test                  test suites
```



Modelos de editor/suscriptor



- Es un patrón de intercambio de datos popular en el que hay dos entidades comunicantes: editores y suscriptores.
- Los **editores** envían mensajes a través de canales específicos sin ningún conocimiento de las entidades receptoras.
- Los **suscriptores** (receptores de mensajes), por otro lado, expresan interés en uno o más de estos canales sin ningún conocimiento sobre las entidades editoriales.
- Es una buena idea incorporar un modelo de este tipo en nuestro proyecto para administrar varias operaciones secundarias correspondientes a una sola acción.

IDE/ EDITORES DE CÓDIGO



Código limpio y fácil de leer



- La mayoría de los flujos de trabajo de configuración de código contienen **un formateador y un linter de código**.
 - Un **linter** busca y advierte sobre código sintáctico erróneo
 - Un **formateador de código** trabaja con los aspectos más estilísticos del código para garantizar un conjunto de pautas de formato y estilo coherentes en todo nuestro proyecto.
- Lo bueno es que la mayoría de los IDE/editores de código como **Visual Studio Code**, comprenden la importancia de escribir código de calidad y proporcionan complementos de linting y formateo que son súper intuitivos y extremadamente fáciles de configurar.



Código limpio y fácil de leer



- También podemos consultar las **guías de estilo** de Javascript utilizados por gigantes como Google. Estas guías cubren todo, desde convenciones de nomenclatura (para archivos, variables, clases, etc.) hasta especificaciones de formato, codificaciones de archivos y mucho más.
- Al escribir código, es importante agregar **comentarios útiles** de los que otros desarrolladores de nuestro equipo puedan beneficiarse. Todo lo que se necesita es una oración de pocas palabras para ayudar a los demás en la comprensión del propósito de los fragmentos de código más complejos.



Escribir código asíncrono



- Javascript es bastante conocido por sus funciones **callback**. También nos permiten definir el comportamiento asíncrono en Javascript. El problema con los callback es que, a medida que aumenta el número de operaciones encadenadas, el código se vuelve más difícil de manejar.
- Para resolver esto, ahora tenemos las **Promises**, que facilitan mucho la escritura de código asíncrono. Además, tenemos **async/await** que la simplifica aún más, haciendo que la API sea más intuitiva y natural.
- Por lo tanto, se recomienda utilizar estas últimas en nuestras aplicaciones de Node. Ésto permite un código más limpio, mejor legibilidad, manejo de errores y pruebas más fáciles. Se mantiene un flujo de control claro y una configuración de programación funcional más coherente.



Archivos de configuración y variables de entorno



- A medida que nuestra aplicación escale, necesitaremos que ciertas opciones de configuración global sean accesibles en todos los módulos.
- Siempre es una buena práctica almacenar estas opciones juntas en un archivo separado dentro de una **carpeta de configuración** en nuestro proyecto. Esta carpeta puede contener todas sus diferentes opciones de configuración agrupadas en archivos según su uso.

```
/config
├── index.js
├── module1.js
└── module2.js
```

- Las URLs de conexión a la base de datos se almacenan en archivos **.env** como variables de entorno. Así es como un archivo .env almacena datos en forma de pares clave-valor. Es un archivo secreto que no se agrega a Git.

```
DB_HOST=localhost
DB_USER=root
DB_PASS=my_password_123
```



Archivos de configuración y variables de entorno



- Una práctica de desarrollo muy común es importar todas las variables del `.env` (junto con otras opciones y configuraciones predefinidas) en los archivos de configuración y exponerlos como un objeto al resto de la aplicación.
- De esta manera, si es necesario hacer cambios, solo los realizamos en una configuración común en un lugar y eso se refleja en toda su aplicación.

```
// config/database.js

require('dotenv').config()

export default {
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  pass: process.env.DB_PASS,
}
```

TESTING, LOGGING Y MANEJO DE ERRORES



Testing, Logging y manejo de errores



- El **testing** es parte integral de cualquier aplicación de software. Nos permite probar la validez, precisión y solidez de nuestro código al sacar a la luz incluso las inexactitudes más pequeñas, no solo en el sistema en conjunto, sino también en sus componentes de forma aislada. Lo hacen de una forma automatizada.
- Las **pruebas unitarias** forman la base de la mayoría de las configuraciones de prueba. Aquí, las unidades/componentes individuales se prueban de forma aislada del resto del código para verificar su exactitud.



Testing, Logging y manejo de errores



- El **Logging** juega un papel importante en todo el proceso de cualquier aplicación de software.
- Desde el desarrollo hasta las pruebas y el lanzamiento para la producción, un sistema de **logging** bien implementado nos permite registrar información importante y comprender los diversos aspectos de la precisión y las métricas de rendimiento de nuestra aplicación. También facilita mucho la depuración.



Testing, Logging y manejo de errores



- La verdad contradictoria es que los errores son buenos para los desarrolladores. Nos permiten comprender las inexactitudes y vulnerabilidades en nuestro código al alertarnos cuando el código se rompe. También brindan información relevante sobre lo que salió mal, dónde y qué se debe hacer para reparar el problema.
- Pero en lugar de permitir que Node arroje errores, interrumpa la ejecución del código e incluso falle a veces, preferimos hacernos cargo del flujo de control de nuestra aplicación manejando estas condiciones de error. Esto es lo que podemos lograr mediante el **manejo de excepciones** usando bloques try/catch.
- Al permitir a los desarrolladores administrar de manera programática tales excepciones, mantiene las cosas estables, facilita la depuración y también evita una mala experiencia para el usuario final.



Compresión de código y tamaño de archivo



- **Gzip** es un formato de archivo sin pérdidas que se utiliza para comprimir (y descomprimir) archivos para una transferencia de red más rápida. Por lo tanto, puede ser extremadamente beneficioso para comprimir las páginas web que sirven nuestros servidores Node, como vimos hace algunas clases.
- También es importante controlar el código de la interfaz para estar al tanto de los tamaños de las páginas web que se sirven. Por lo tanto, debemos asegurarnos de minimizar el código HTML, CSS y Javascript de la interfaz utilizando herramientas como HTMLMinifier, CSSNano y UglifyJS antes de servir.
- Estos **minificadores** eliminan los espacios en blanco innecesarios y los comentarios de los archivos y realizan algunas optimizaciones triviales del compilador, lo que en general resulta en un tamaño de archivo reducido.



Inyección de dependencia



- La inyección de dependencias es un patrón de diseño de software que pasa (o inyecta) dependencias (o servicios) como parámetros a nuestros módulos en lugar de requerir o crear específicos dentro de ellos.
- Es un concepto muy básico que mantiene nuestros módulos más flexibles, independientes, reutilizables, escalables y fácilmente probables en toda nuestra aplicación.
- Con esto, nuestro servicio es una interfaz genérica que no solo es fácil de reutilizar sino también más fácil de testear con pruebas unitarias.



Soluciones de terceros



- Node tiene una gran comunidad de desarrolladores en todo el mundo. En lo que respecta al soporte de terceros, el administrador de paquetes de Node, **NPM** está lleno de frameworks, bibliotecas y herramientas con muchas funciones, bien mantenidos y bien documentados para cualquier caso de uso que podamos imaginar. Por lo tanto, es muy conveniente para los desarrolladores conectar estas soluciones existentes en su código y aprovechar al máximo sus API.
- Si bien estas bibliotecas y herramientas alivian gran parte de la carga, es importante ser inteligente y responsable con cada paquete que importamos. Debemos conocer el propósito, las fortalezas y las debilidades de cada uno y asegurarnos de no depender demasiado de ellos.



Usar herramientas de monitoreo de aplicaciones



- Para aplicaciones a gran escala en producción, uno de los principales objetivos es comprender mejor cómo los usuarios interactúan con la aplicación: sobre qué rutas o características se utilizan con más frecuencia, sobre las operaciones que se realizan con más frecuencia, etc. evaluar métricas de desempeño, problemas de calidad, cuellos de botella, errores comunes, etc. y usar esa información para realizar los cambios y mejoras necesarios.
- Acá es donde entran en escena las herramientas de monitoreo de aplicaciones (**APM**) como ScoutAPM. Este nos permite analizar y optimizar de manera constructiva el rendimiento de nuestra aplicación web.
- Nos brinda información en tiempo real para que podamos identificar y resolver rápidamente los problemas antes de que el cliente los vea.

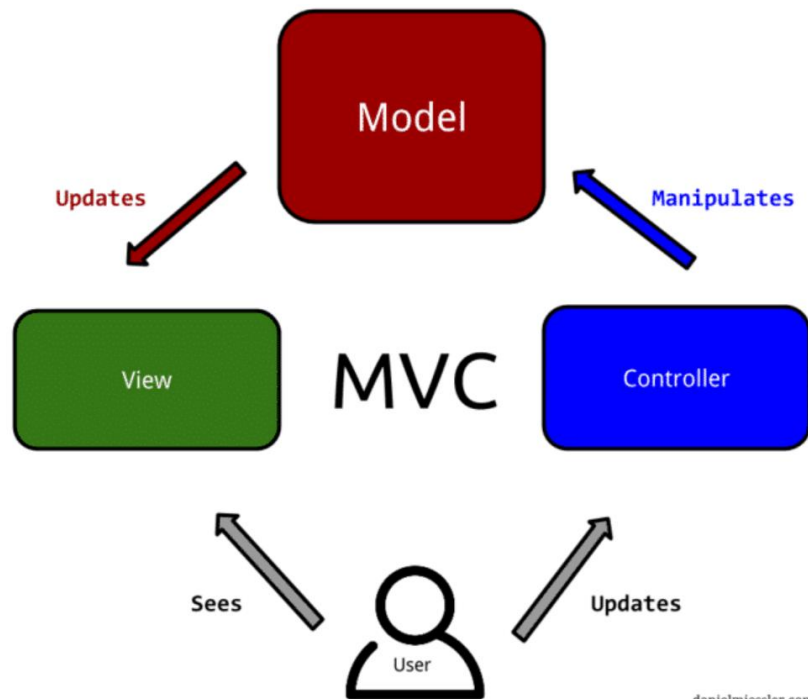


BREAK

¡5/10 MINUTOS Y VOLVEMOS!

PATRÓN MVC

Model, View, Controller.



danielmiessler.com

- **MVC**, es un patrón arquitectónico que separa una aplicación en tres componentes lógicos principales:
 - **Modelo.**
 - **Vista.**
 - **Controlador.**
- Cada uno de estos componentes está diseñado para manejar aspectos de desarrollo específicos de una aplicación.

Patrón MVC



- Al usar la arquitectura MVC en la construcción de servidores web, logramos que el proceso de desarrollo sea más fácil por la división del servidor en los 3 componentes, separando responsabilidades.
 - **Controlador:** Es la parte que se encarga del procesamiento de la solicitud del cliente que maneja esta solicitud y devuelve una respuesta.
 - **Modelo:** Es responsable del dominio de datos de la aplicación. Los objetos de modelo son responsables de almacenar, recuperar y actualizar datos de la base de datos.
 - **Vista:** es el que compila y renderiza en HTML simple. Es la interfaz de usuario de nuestra aplicación. Es la forma en que el usuario obtiene la respuesta de lo que solicitó.

HTML ON WIRE



HTML ON WIRE



- Hay diferentes formas de generar las vistas en un patrón MVC.
- **HTML on wire** genera las vistas en el backend, por ejemplo, con un motor de plantillas con Pug.
- De esta forma, no se tiene una API REST por un lado y un frontend por el otro, sino que dentro de un mismo proyecto tenemos toda la aplicación, solo en backend, incluidas las vistas.
- Estas vistas, son renderizadas en el controlador, como respuesta a las solicitudes que realiza el usuario de la aplicación.



Creando una app con MVC y Express



- Para crear una aplicación usando MVC con HTML on wire, y Express, empezamos con la siguiente estructura principal de carpetas en nuestro proyecto:

```
src/  
--controllers/  
--models/  
--views/  
--routes/
```

- Tenemos las 3 carpetas de los componentes de MVC en los archivos de rutas, en la carpeta homónima. Desde dentro, ejecutan al método correspondiente del controlador. De esta forma queda mejor separada la responsabilidad de cada uno.



Creando una app con MVC y Express



- Vemos en el siguiente código un ejemplo de un **modelo** para una colección de comidas en una base de datos de MongoDB.

```
const mongoose = require('mongoose');
const mealModel = mongoose.Schema({
  _id: mongoose.Schema.Types.ObjectId,
  name: { type: String, required: true },
  type: { type: String, required: false },
  price: { type: Number, required: false }
});
module.exports = mongoose.model('Meal', mealModel);
```

- Usamos **Mongoose** para definir el modelo y poder usar nuestra base de datos en nuestro proyecto de Node.



Creando una app con MVC y Express



- Te presentamos un **controlador**, asociado al modelo de comidas.
- El ejemplo muestra al método para traer todas las comidas que estén en la base de datos. También podríamos tener métodos para crear, modificar o eliminar una comida.
- Como retorno del método, usamos `res.render()` para renderizar la vista.

```
const mealModel = require("../models/mealModel");

module.exports = {
  getMenuController: (req, res, next) => {
    const meals = mealModel.getMeals();

    res.render("menu", { meals });
  }
}
```



Creando una app con MVC y Express



- En el archivo de **rutas**, encontramos las mismas, con el método de controlador que corresponda ejecutar en cada una.
- En este caso, vemos que la ruta por GET que corresponde al método de controlador que vimos antes, es *“/menu”*.

```
const express = require('express');
const router = express.Router();
mealsController = require("../controllers/mealsController");

router.get("/menu", mealsController.getMenuController);

module.exports = router;
```



Creando una app con MVC y Express



- Finalmente, vamos a la carpeta de **vistas** y creamos el archivo de *menu.pug* que es el que renderizamos en el controlador.
- Escribimos, en este caso, un código simple, con el motor de vistas Pug, para que muestre un listado de las comidas que trae de la base de datos con su nombre y su precio.

```
html
  body
    ul
      each meal in meals
        li #{meal.name} price: #{meal.price}
```



ARQUITECTURA MVC CON HTML ON WIRE

Tiempo: 10 minutos

ARQUITECTURA MVC CON HTML ON WIRE

Desafío
generico



Tiempo: 10 minutos

- Crear una arquitectura de servidor node.js MVC que permita ingresar personas por nombre, apellido y DNI mediante un formulario ofrecido a través de sus rutas.
- Por debajo del formulario se representarán en forma dinámica, los datos almacenados en memoria, en forma de tabla.

👉 Realizar este proceso utilizando HTML on wire, creando las rutas necesarias:

ARQUITECTURA MVC CON HTML ON WIRE

Desafío
generico



Tiempo: 10 minutos

- **Ruta get '/html-onwire'**: devolverá una vista dinámica (hbs) que construya el servidor con el formulario y los datos.
- **Ruta post '/html-onwire'** usada por el formulario html-onwire
- Utilizar Handlebars del lado servidor (HTML on wire) para generar la vista dinámica.
- Separar el desarrollo del lado servidor en capas: rutas, controlador, negocio y modelo (MVC)

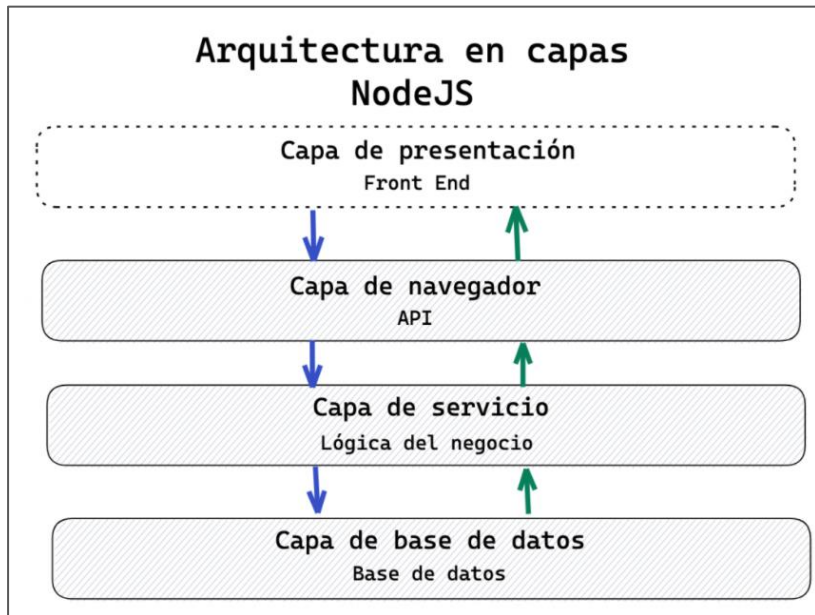
PATRÓN MVC - DATA ON WIRE

¿De qué se trata?



- La diferencia con HTML on wire, es que en este caso, las vistas se realizan por separado, en un frontend, que puede ser realizado por ejemplo con React.
- Entonces, lo que devuelve el backend, desde el controlador, es un json, en lugar de un HTML.
- En el backend, seguimos teniendo las rutas, modelos y controladores. Lo único que cambia es la forma en que llega a los usuarios la respuesta de sus solicitudes.

Arquitectura de capas



- Como ya vimos en clases anteriores, la arquitectura de capas en **Node** se puede estructurar como vemos en la imagen.
- En el *backend* tenemos la capa de servicio (controlador), la capa de base de datos (modelo) y la capa de API (rutas).
- En el *frontend* tenemos la capa de presentación.

COMUNICACIÓN ENTRE FRONT Y BACK



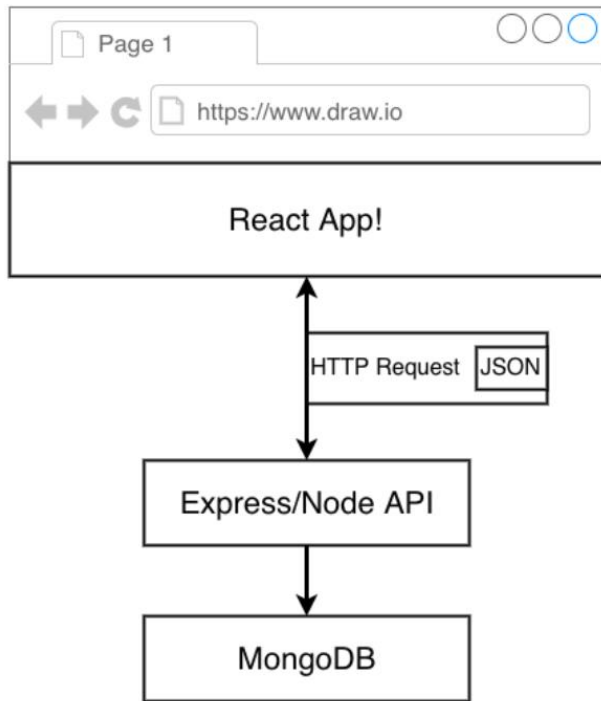
Comunicación entre front y back



- Si nuestra aplicación es dinámica, cuando un usuario entre desde su navegador, realizará una solicitud HTTP, por ejemplo, para que se muestre el listado de productos que vendemos.
- En este caso, la solicitud se realiza desde el front, pero los datos de productos están almacenados en la base de datos. Entonces, tenemos la **API REST** para comunicar ambas partes.
- Como ya vimos, la API REST tiene toda la lógica para manejar las peticiones que provienen del front, pedir la información necesaria a la base de datos y enviar esta información de nuevo al front, donde es utilizada para mostrar al usuario el contenido que solicitó.



Comunicación entre front y back



- En el diagrama vemos que primero, el usuario ingresa a nuestra aplicación desde su navegador.
- Lo que se muestra es el frontend, que en este caso está desarrollado con React. Este, envía la solicitud del usuario al backend.
- El backend es la API REST, desarrollado en Node. Éste, pide a la base de datos la información requerida por el usuario y la envía en formato json al frontend como respuesta.



Estructura en el backend



- La estructura de carpetas en el backend, es similar al que teníamos con HTML on wire, con la diferencia que no tenemos la carpeta de vistas.
- Los archivos de rutas y los de modelos son iguales a los que vimos antes.
- Sin embargo, el controlador cambia en su retorno. En lugar de renderizar una vista, devuelve un json con la información de la respuesta de la solicitud HTTP que realizó el usuario. El mismo del ejemplo anterior queda:

```
const mealModel = require("../models/mealModel");

module.exports = {
  getMenuController: (req, res, next) => {
    const meals = mealModel.getMeals();

    res.status(201).json(meals)
  }
}
```




¿Cómo utiliza el front la respuesta?



- En el frontend, debemos consumir el json que nos envía la API REST para poder mostrar los datos al usuario.
- Para eso, podemos usar un cliente HTTP, como vimos la clase pasada. Por ejemplo, usamos Axios para consumir los datos del ejemplo de comidas.
- En result.data tenemos el array con los objetos de cada una de las comidas. Podemos con Javascript entonces, pasar esa información a un HTML para mostrarla finalmente al usuario.

```
import axios from "axios";
export default {
  allMeals() {
    axios.get("https://localhost:8080/menu").then((result) => {
      console.log(result.data);
    })
  }
};
```



ARQUITECTURA MVC CON DATA ON WIRE

Tiempo: 10 minutos

ARQUITECTURA MVC CON DATA ON WIRE

Desafío
generico



Tiempo: 10 minutos

- Utilizando el servidor del desafío anterior, agregar al proceso HTML on wire, las rutas que permitan mostrar el concepto DATA on wire. Para eso, deberás generar estas nuevas rutas:
 - **Ruta get '/data-onwire'**: devolverá una vista estática (HTML) con el formulario y el código necesario para representar los datos que le lleguen mediante una petición del lado cliente a la ruta '/json'.
 - **Ruta post '/data-onwire'** usada por el formulario data-onwire
 - **Ruta get '/data-json'** : Esta ruta devolverá un objeto JSON con los datos almacenados, utilizada por la vista 'data-onwire'.
- Utilizar Handlebars del lado cliente (DATA on wire) para generar la vista dinámica.
- Seguir utilizando MVC en el desarrollo.

PATRONES DE DISEÑO

¿De qué se trata?

- Los patrones de diseño son una forma de estructurar el código de nuestra solución, de manera que nos permita obtener algún tipo de beneficio, como velocidad de desarrollo más rápida, reutilización de código, etc.
- Son una solución general y reutilizable para un problema común.
- No es obligatorio utilizar los patrones de diseño. Solo es aconsejable en el caso de tener el mismo problema o similar, siempre teniendo en cuenta que en un caso particular puede no ser aplicable.

PATRÓN IIFE

CODER HOUSE



¿De qué se trata?



- IIFE significa Expresiones de función inmediatamente invocadas. Nos permite definir y llamar a una función al mismo tiempo.
- Debido a la forma en que funcionan los ámbitos de JavaScript, el uso de IIFE puede ser excelente para simular cosas como propiedades privadas en clases. De hecho, este patrón en particular se usa a veces como parte de los requisitos de otros más complejos.

Implementación

- La plantilla para un IIFE consiste en una declaración de función anónima, dentro de un conjunto de paréntesis (que convierte la definición en una expresión de función, también conocida como una asignación) y luego un conjunto de paréntesis de llamada al final de la misma.

```
(function(/*received parameters*/) {  
  //your code here  
})(/*parameters/>)
```

- Un ejemplo puede ser:

```
(function() {  
  var x = 20;  
  var y = 20;  
  var answer = x + y;  
  console.log(answer);  
})();
```


PATRÓN SINGLETON



¿De qué se trata?



- Es un patrón bastante simple pero nos ayuda a realizar un seguimiento de cuántas instancias de una clase estamos instanciando. De hecho, nos ayuda a mantener ese número en uno solo, todo el tiempo.
- Básicamente, el patrón Singleton nos permite crear una instancia de un objeto una vez y luego usarlo cada vez que lo necesite, en lugar de crear uno nuevo sin tener que realizar un seguimiento de una referencia a este, ya sea globalmente o simplemente pasándolo como un dependencia en todas partes.



¿Cómo lo implementamos?



- Normalmente, otros lenguajes implementan este patrón utilizando una única propiedad estática donde almacenan la instancia una vez que existe. El problema con Node es que no tenemos acceso a variables estáticas en JS. Entonces, podríamos implementar esto de dos maneras:
- Usando **IIFE** (Expresiones de función inmediatamente invocadas) en lugar de clases.
- Usando **módulos ES6** y hacer que nuestra clase singleton use una variable global local, en la que almacenamos nuestra instancia. Al hacer esto, la clase en sí se exporta fuera del módulo, pero la variable global permanece local en el módulo.

¿Cómo lo implementamos?



```
let instance = null

class SingletonClass {

  constructor() {
    this.value = Math.random(100)
  }

  printValue() {
    console.log(this.value)
  }

  static getInstance() {
    if(!instance) {
      instance = new SingletonClass()
    }

    return instance
  }
}

module.exports = SingletonClass
```

- Para implementarlo, vemos que creamos una clase de ES6.
- En este caso, definimos una variable en el constructor, y luego un método para obtener el valor de esa variable.
- Vemos que dentro de la clase, tenemos el método **`getInstance()`**, que es **estático**, y que crea una instancia de la clase, si esta no existe. Entonces, el código queda reutilizable sin volver a instanciar en cada archivo donde necesitemos usarlo.



¿Cómo lo usamos?



```
const obj = Singleton.getInstance()
const obj2 = Singleton.getInstance()

obj.printValue()
obj2.printValue()

console.log("Equals:: ", obj === obj2)
```

- En el archivo que queramos usarlo, primero debemos importar el archivo donde implementamos la clase, por ejemplo como:

```
const Singleton = require("./singleton")
```

- Luego, lo usamos como vemos en el código. En este caso, ejecutamos dos veces el método `getInstance()` y luego imprimimos en consola lo que devuelve cada una.



¿Cómo lo usamos?



- Finalmente obtenemos en la salida:

```
0.5035326348000628  
0.5035326348000628  
Equals:: true
```

- Como vemos, el valor de “Equals” nos da true. Con ésto confirmamos que, de hecho, solo estamos creando una instancia del objeto una vez y devolviendo la instancia existente y por eso estos dos valores coinciden.

Casos de uso



- Al intentar decidir si necesitamos una implementación tipo Singleton o no, debemos considerar algo:

🤖 ¿Cuántas instancias de nuestras clases necesitaremos realmente? Si la respuesta es 2 o más, este no es el patrón a elegir.
- Sin embargo, por ejemplo, puede haber ocasiones en las que tengamos que lidiar con conexiones de bases de datos que deseemos considerar. En este caso, una vez que nos hayamos conectado a la base de datos, podría ser una buena idea mantener viva esa conexión y accesible a través del código. Y esto lo podemos resolver, entre otras opciones, con el patrón Singleton.



PATRÓN SINGLETON

Tiempo: 10 minutos

PATRÓN SINGLETON

Desafío
generico



Tiempo: 10 minutos

Tomando como base el proyecto del desafío anterior, crear un objeto singleton llamado **PrimeraConexion** que me permita obtener la hora de conexión, mediante el método **obtenerHora**, del primer cliente que se conecte a la ruta '/datos'.

El objeto se debe instanciar dentro de dicha ruta y en cada solicitud debe informar esa hora inicial.

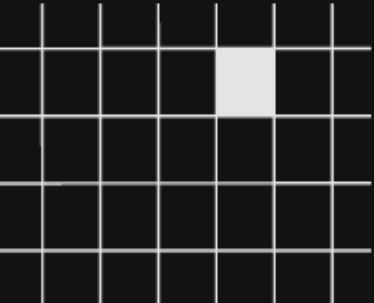
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Buenas prácticas en proyectos de Node
 - Patrón MVC - HTML on wire
 - Patrón MVC - Data on wire
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN