CLASE 4.1 NODE.JS GRABADA (20/4/2023)

Hacemos un pedido de paquetes a travez de internet. Nuestra proveedora nos facilita llevar ese pedido a los servidores de destino. Esos servidores, si funciona todo bien, nos devuelve el html,css y mas archivos para que podamos utilizar la pagina web seleccionada.

Toda la informacion de la pagina/proyecto, se guardan en los servidores (Computadoras con gran caudal de informacion). Nodejs puede simular ser un servidor.

Servidores web mas usadas: Apache, JBoss, Litespeed, Lighttpd,Glassfish.

Los tipos de pedidos/metodos para hacerle un servidor son GET, POST, PUT,DELETE.

Nodejs nos permite ejecutar codigo javascript del lado del servidor.

Los navegadores, tienen motores para leer codigo JS (ejemplo el motor v8 para chrome).

Node.js fue creado por Ryan Dahl en 2009. Logró que las computadoras del lado del servidor puedan leer codigo JS, cosa que antes solo lo podian hacer los motores de los los navegadores. Esto lo hizo para crear un servidor web que pueda funcionar con javascript.

Node.js es un entorno de ejecucion para Javascript construido con V8.

Este programa va a poder ejecutar archivos js y actuar como servidor, responder peticiones.

Los hosting (alojamiento), por ejemplo utilizan en sus servidores sistemas operativos con el nodejs para poder trabajar.

Modelos de alojamiento de las compañias para guardar la informacion. Hay 4 maneras:

-Alojamiento compartido: Se alojan muchas paginas web, proyectos y archivos.

-Alojamiento con servidores dedicados: Dedcados a una sola pagina.

-VPS(Servidor Virtual Privado): Se utiliza VMware para virtualizar otro sistema operativo y alojar proyectos en cada maquina virtual.

-Cloud: Aquí un proyecto web se encuentra localizado en distintas partes.

CLASE 4.2 GRABADA (25/4/2023)

Ver Practica node-practica-4

Aclaracion. El BOM y DOM no existe en node.js

En node, vamos a utilizar mucho las arrow function y callbacks.

Con las callbacks llamamos a una funcion dentro de otra.

Funcion impura:

let variable1 = 25;

function fun2() {...}

fun2();

function function1(){

    fun2();

    return variable1 + 5;

}

Es impura porque no es la forma “correcta” de llamar a una funcion dentro de otra. Aunque en la practica pueda funcionar. Es impura porque no sigue puramente el concepto de funcion (porcion del codigo separada del codigo principal). La unica forma de pasarle un dato a una funcion es a traves de los parametros. Como aquí no lo estamos pasando por parametro, es impuro.

Funcion pura:

function fun2() {...}

function function1(param){//param = fun2

    param(); //Llamamos fun2

}

function1(fun2);

Recordar foreach

let lista\_fruta=["manzana","banana","naranja"];

lista\_fruta.forEach((element,index) => {

    console.log(element +" "+index);

});

O tambien

lista\_fruta.forEach(funcionParaForeach);

Desestructuracion de arrays.

Forma larga:

let lista\_numeros = [1, 2, 3];

let uno = lista\_numeros[0];

let dos = lista\_numeros[1];

let tres = lista\_numeros[2];

Forma corta:

let lista\_numeros = [1, 2, 3];

let [uno, dos, tres] = lista\_numeros;

De la forma corta, lo que sucede internamente es que saca cada valor de lista\_numeros y los coloca dentro de las variables nombradas en las llaves.

Desestructuracion de objetos.

Forma larga

let objeto\_persona ={nombre:"Agustin", edad:20, tienePerro:false}

let var1= objeto\_persona.nombre;

let var2= objeto\_persona.edad;

let var3= objeto\_persona.tienePerro;

Forma corta

let objeto\_persona ={nombre:"Agustin", edad:20, tienePerro:false}

let {nombre,edad,tienePerro} = objeto\_persona;

Tomar en cuenta que los nombre de las variables deben ser las mismas que las keys.

Template literales

Es una nueva manera de escribir concatenacion de caracteres.

let texto=”Bienvenido”;

let num= 21;

let nombreConTemplateLiterales=`${texto} tu edad es ${num}`

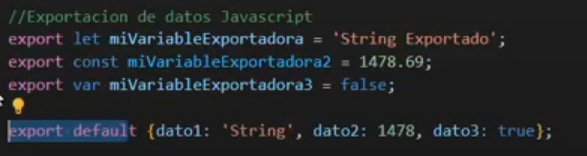
Recordar las comillas invertidas con alt+96.

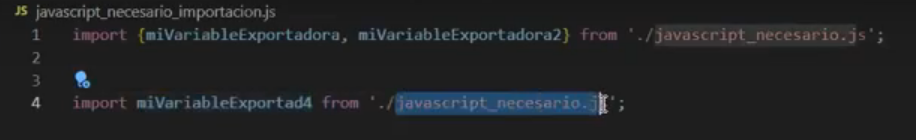
CLASE 4.3 GRABACION (2/5/2023)

Cuando hacemos varios export normal, en la importacion tiene que estar entre llaves por si queremos determinar que de ese archivo vienen varios datos asi utilizarlos.

En cambio, con el export default, como solo se puede hacer uno por archivo, en la importacion no le tenemos que poner entre llaves ya que solo viene uno por archivo. Ademas, con el export default no hace falta aclarar tipo de variable ni nombre, si no tiene nombre, en el import se le da el nombre que queramos.

Por lo general, para exportar varios datos, se utiliza un export default en un objeto en el cual dentro tiene esos datos que queremos exportar. Si queremos podemos darle nombre a ese objeto ahí mismo, sino le podemos dar nombre en el import.





(El primero es import normal, el segundo es un export default).

Iniciamos con node.js.

Vease el archivo introduccion\_node.js en node-practica-4

Para ejecutar un archivo js con node, tengo que poner en consola node nombrearchivo.js

Node.js creado por Ryan dahl, es un entorno de eecucion para javascript creado con el v8, el motor de javascript de chrome.

Un archivo js normal, necesita ser importado si o si a un html a traves de la etiqueta script. Ademas, tenia el DOM y BOM para comunicarse con el navegador.

Node.js es un programa en el cual no necesita todo eso. Solo ejecuta javascript.

En js normal y node.js se pueden usar los mismos tipos de variables.

En js normal, podiamos invocar al objeto padre ‘window’ y tambien ‘document’ . En node.js, en lugar de existir estos, al no haber navegador, esta el objeto padre ‘global’.

‘\_\_filename’ es un atributo que tiene la ruta del archivo.

‘\_\_dirname’ es un atributo que tiene la ruta del archivo pero sin decirnos el nombre del archivo.

‘module’ es un objeto que tiene informacion del modulo actual. Gracias a module podemos almacenar toda la informacion de un modulo. Lo que mas importa de este ultimo, es el atributo exports. Con ‘module.exports = datoCualquiera’ le puedo asignar un valor, funciona como el export default. Se puede importar con require(ubicacionDeExports).

‘require()’ es una funcion que nos permite importar js. Importamos todo lo que este dentro del export del module del archivo seleccionado.

‘process’ Nos da informacion cuando de todos los procesos cuando se ejecuta el archivo. Nos vamos a concentrar en ‘env’ que nos va a dar la informacion de carpetas y archivos que se ejecutan. Dentro de ella vamos a definir archivos publicos.

Para node.js, cada archivo js es un modulo. Los modulos nos van a servir para exportar e importar datos. Hay 3 tipos de modulos, los creados por nosotros, los creados por Ryan Dahl para node.js, y modulos de otros programadores.

CLASE 4.4 GRABACION (9/5/2023)

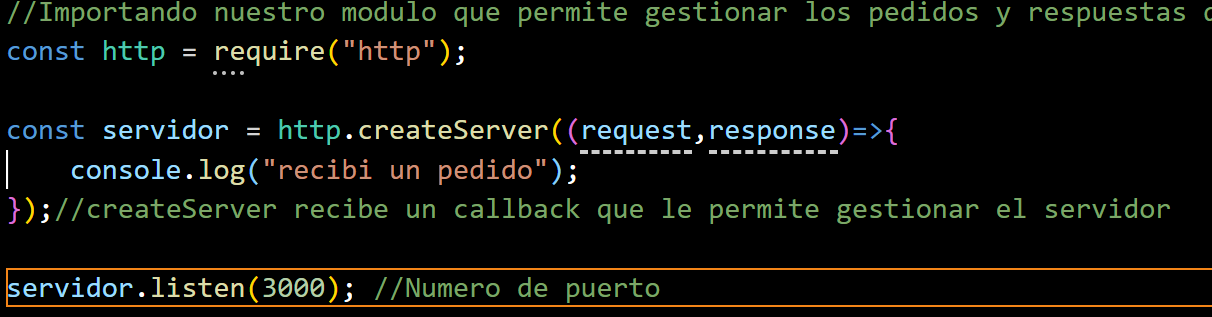
Ver archivo modulos\_node.js en el cual vemos distintos metodos del modulo ‘os’ propio de node.js.

Ver archivo servidor.js

Cuando tenemos la intencion de instalar varios paquetes en nuestro proyecto, vamos a ejecutar ‘npm init’. Y se creara el packge.json para manejar los paquetes.

Al crear un servidor gracias al metodo createServer, necesitamos pasar por su parametro, una funcion que permita gestionar al servidor, esta funcion de adentro tiene que tener como parametro request y response.

Con lo que tenemos aquí



Si ejectamos el codigo, vemos que se queda trabado, está esperando a que ingresemos al servidor (que le pusimos en el puerto 3000). Al ingresar en el navegador localhost:3000 luego de ejecutar el codigo, vemos que muestra en consola “recibi un pedido”, eso significa que se pudo entrar al servidor y ejecutó el codigo que hay en la funcion anonima de la callback. Pero todavia no le declaramos que de una respuesta, queremos que la respuesta sea el servidor web, para eso se utiliza ‘response’. Y para recibir los distintos tipos de pedidos se utiliza ‘request’.

Para cerrar el servidor damos al control+C.

CLASE 4.5 GRABACION (11/5/2023)

Al metodo listen(3000) le podemos agregar el numero de ip de nuestra pc. Luego para entrar en el navegador debemos poner numeroIp:3000.

A evitar problemas de cache. Vamos a instalar el paquete nodemon. Sirve para limpiar la cache en la carga del navegador.

El parametro ‘request’ tiene dentro la informacion del pedido (lo que nos da el usuario). Al hacer request, es decir recibir lo que nos da el usuario, se está haciendo un pedido/metodo GET.

Nuestra respuesta, tiene que finalizar y enviar la informacion al usuario. Eso lo hacemos con response.end(). Dentro del parametro podemos enviar el servidor web.

Antes del response.end() no se recomienda poner otra cosa.

Vamos a usar el modulo express nos permite simular un servidor. Lo tenemos que descargar con npm install express. Se basaron en HTTP pero lo mejoraron para dar mas funcionalidades y mas facilidad.

CLASE 4.6 GRABACION (16/5/2023)

Como se puede ver http es un poco limitado, por eso vamos a usar express que nos permite hacer cosas mas complejas como la interaccion con rutas, cosa que http lo hacia de manera menos eficiente. Ademas se importa en el proyecto simplemente con require, cosa que con http es mas complejo. Tambien separa por un lado los estilos y por el otro las funcionalidades.

Para escribir pedidos con express:

servidor.get('/',(req, res)=>{

Forma 1- En esta seleccionamos el tipo de metodo HTTP como funcion, como primero parametro tenemos la ruta donde el usuario hace el pedido. El segundo es una arrow function que se ejecutará. Dentro de esa funcion vamos a responder con html,css y js.

(Ver archivo servidor\_express.js)

Forma 2- Usando Route (Ver archivo servidor\_express-rutas.js)

Con route:

servidor.route("/contacto")//Forma 2

.get((req,res)=>{})

.post((req,res)=>{})

Con este se puede apreciar que en el parametro solo ponemos la ruta en donde se hace el pedido. El metodo se pone por separado.

Route es utile para dar varias funcionalidades a la misma pagina a traves de rutas. Como vimos recien el ejemplo de contacto.

Como a muchos programadores no les gusto la forma de Route, es decir que primero se ponga la ruta y luego el tipo de pedido. Se decidió combinar lo mejor de la forma de express vanilla (Forma 1) y lo mejor de Route (Forma 2). Y de ahí salió Router que es la mas utilizada y mas compleja

Forma 3- Usando Router (Ver archivo servidor\_express-rutas.js)(recomendable):

Tambien Router es una funcion que nos da express. Primero hay que importarla. Y luego la utilizamos como la forma 1, es decir que primero indicamos el tipo de pedido y luego la ruta. El router lo vamos a utilizar cuando queramos hacer una ruta de funcionalidades o paginas con respecto a la ubicacion que se utilice.

router.get("/contacto/crear",(req,res)=>{

    res.sendFile(\_\_dirname + "/contacto.html")

})

router.get("/contacto/editar",(req,res)=>{

    res.sendFile(\_\_dirname + "/contacto.html")

})

router.get("/contacto/eliminar",(req,res)=>{

    res.sendFile(\_\_dirname + "/contacto.html")

})

Y ademas tenemos que indicar que el servidor utilice esa ruta.servidor.use(router);

Dentro del podemos enviar archivos html con el metodo siguiente:

res.sendFile(\_\_dirname + "/index.html");

El \_\_dirname es una variable global en la cual dentro tiene la ruta del archivo donde estamos trabajando. Podemos ver que si visualizamos el proyecto y si este le damos desde un principio archivos css y js, no se mostraran ni se enviaran desde el servidor.

Si queremos agregar otros archivos js y css a ese archivo html, ademas de poner los href, le tenemos que indicar donde van a estar esos archivos estaticos que vamos a agregar. Esto lo hacemos con:

express.static("public")

Con esto decimos que los archivos estaticos van a estar en la carpeta “public”.

Para enviarlo a nuestro servidor tenemos que usar la funcion ‘use’.

servidor.use(express.static("public"));

NOTA: Recordar que los archivos estaticos agregados al html por el metodo recien nombrado, en el href o src no hay que aclarar que estan dentro de la carpeta public.

Esta bien:<link rel="stylesheet" href="estiloagregado.css">

Esta mal:<link rel="stylesheet" href="/public/estiloagregado.css">

Arquitectura MVC (Modelo Vista Controlador)

Paso 1- El cliente hace un pedido(Request)

Paso 2- Ese pedido se pasa a la ruta (Por ejemplo: servidor.get("/contacto",)

Paso 3- El pedido se pasa al controller, este controla lo que yo quiero hacer con el. Controla la logica del proyecto.

Paso 4- El controller puede mandar la informacion a la base de datos (Model) para luego devolverla

Paso 5- El controller envia la informacion al usuario para que lo vea (View)

Nota: Con respecto al punto 4 y 5. El controller, controla para donde va a ir la informacion dada por el usuario, controla si va hacia otra vista o si va hacia una base de datos.

Usando ejs para trabajar con variables en html:

Primero debemos instalar la libreria ejs en nuestro proyecto con npm i ejs.

El motor de visualizacion es el tipo de archivo que se va a utilizar para visualizar lo que queremos que el cliente vea.

Para que el motor de visualizacion o view no sea a traves de html sino que sea con plantillas de html en la cual le podemos ingresar variables, lo vamos a hacer con ejs.

(Vease el archivo servidor-ejs.js y la carpeta view)

//Al servidor le indicamos la visualizacion

servidor.set("view engine", "ejs");//Que use como motor de plantillas, el ejs

servidor.set("views", \_\_dirname+"/views");//Que donde se muestren las vistas sea en views

//Al servidor le indicamos las rutas disponibles

servidor.route("/")

.get((req,res)=>{

    res.render("index.ejs");

})

Indicamos el motor de vistas y que indiquemos donde esta las plantillas que vamos a usar y modificar, y ademas en el route() indicamos que cuando se entre a la pagina de inicio, el get que se hace al entrar, retorne como respuesta con render.

Render renderiza todo lo que haya dentro de views, sino no funciona.

(Aclaracion: la localizacion en el render se encuentra en la carpeta “views” porque aclaramos mas arriba que ahi es donde vamos a trabajar la visualizacion, asi que cuidado con la ruta dentro del render).

Para recibir y dar variables a traves del render:

.get((req,res)=>{

    res.render("index.ejs", {variable1: "Agustin"});

})

Como se ve, utilizamos un objeto como segundo parametro, donde estaran las variables a usar

Y en el ejs ponemos:

<body>

    <h1>Inicio por ejs</h1>

    <p>Mi dato secreto es que mi nombre es: <%= variable1 %></p>

</body>

Como se ve, para traer la variable del servidor, utilizamos <%= nomvariable %> ,con esa etiqueta ejs evia o recibe informacion .

Dentro de los <% %> (sin el =) podemos escribir codigo js.

En resumen, ejs es HTML + javascript

Hace cosas parecidas a lo que hace React.

Con ejs escribimos html que le podemos insertar codigo js. React esta basado en componentes a los cuales le podemos insertar codigo tambien y su motor de visualizacion es jsx. Los dos tienen el objetivo de combinar js con html pero lo hacen de manera diferente.

Ahora estamos usando ejs para entender conceptos, pero en el futuro utilizaremos jsx.

CLASE 4.7 (GRABACION 23/5/2023)

Layout (es nombre representativo, puede ser cualquiera)

Asi podemos llamar a la carpeta que contenga componentes que se repitan en varias paginas.

Por ejemplo, si tenemos varias paginas como inicio, contactos, productos, y todas tienen el mismo header y footer, entonces esos los modularizamos como haciamos con react.

<!DOCTYPE html>

<html lang="en">

    <%- include("./layout/head")%>

    <body>

        <h1>Inicio por ejs</h1>

        <p>Mi dato secreto es que mi nombre es: <%= variable1 %></p>

    </body>

</html>

Como se puede ver, el componente se importa asi en ejs. Es un proceso similar al de react.

(Recordar que gracias al render, estamos apoyados en la carpeta “views”)

CLASE 4.8 (30/05/2023)

Si le queremos agregar estilos al archivo ejs (ver archivo index.ejs y head.ejs), podemos utilizar la etiqueta link como hacemos normalmente. Pero recordar que ese archivo style tiene que estar dentro de la carpeta “public” que declaramos hace un rato como carpeta de archivos estaticos al servidor.

Incorporacion de bootstrap o tailwind

En esta practica lo voy a integrar de manera online en el head de la pagina gracias a la etiqueta link.

Si queremos integrarla de manera offline (recomendable), descargamos el script de boostrap para tenerlo de manera local, recordar meterlo dentro de la carpeta public.

Practicando esquema MVC: (vease archivo servidor-mvc.js)

Pasar datos del usuario al servidor por url.

Como desde la pagina estamos subiendo algo hacia el servidor, se esta haciendo un pedido POST.

<http://localhost:3000/?dato1=15&dato2=24>

En este ejemplo, los datos van luego del signo de pregunta. Entonces estamos pasando el dato1=15 y el dato2=24.

Para obtener esa informacion enviada por el usuario, lo podemos hacer a traves del ‘request.query’.

Por otrolado. Si creamos una pagina de formulario en el cual rellenamos datos y le damos a submit. Como desde la pagina estamos subiendo algo hacia el servidor, se esta haciendo un pedido POST.

Para obtener esa informacion enviada por el usuario, lo podemos hacer a traves del ’request.body’.

NOTA: Recordar que en el html donde esta la etiqueta form tenemos que declarar el tipo de metodo y la ruta donde se hace la accion.

<form method="post" action="/contacto">

Con esto se envia la informacion del formulario al metodo post de la ruta contacto y ejecuta el codigo de la misma.

En este punto, no podemos ver la informacion enviada por url porque está codificada. Entonces, le indicamos a nuestro servidor que cuando reciba pedidos, use el urlencoded. Eso hace que se decodifique la url.

Ademas, para pasar la informacion al servidor, va a ser necesario declarar al servidor que lo decodifique en formato json.

servidor\_web.use(express.urlencoded({extended: true}));//Le indicamos a nuestro servidor que cuando reciba pedidos, use el urlencoded. Eso hace que la url venga decodificada

servidor\_web.use(express.json()); //Indicamos al servidor que decodifique en formato json

Para pasar un dato de un campo llenado por el usuario hacia al servidor, esos input deben tener el atributo ‘name’ con un valor descriptivo. Vease el archivo contacto.ejs.

Con poner esos atributos, podemos ver en la body de la request que llego al servidor los valores puestos por el usuario.

NOTA: Por lo general lo que viene en el body viene gracias a un post hecho.

CLASE 4.9 GRABACION (6/6/2023)

En la clase pasada vimos que la informacion pasada por un form llega al servidor en el body de la request. (gracias a un POST)

Si yo quiero mandar estos datos del usuario desde una visualizacion a otra para mostrarlo como respuesta, tenemos que mandar los datos en la misma funcion donde se renderiza. Por ej: (vease servidor-mvc.js)

servidor\_web.route('/contacto/formulario1')

.post((req,res)=>{

    let datos\_formulario= req.body;

    res.render("contacto\_respuesta.ejs",datos\_formulario);

})

Aquí vemos que queremos renderizar como respuesta un ejs, al que le estamos pasando el objeto que contiene los datos enviados por el usuario al servidor.

Entonces, si queremos utilizar los datos que contiene el objeto que pasamos, tenemos que usar el ‘<%= %>’ con el key dentro que queramos usar

<div class="container">

        <div class="alert alert-primary" role="alert">

            <%=email %>

            <%=nombre %>

        </div>

</div>

En este ejemplo se mostraran los datos email y nombre

Vamos a utilizar xammp para simular un servidor web, en esta vamos a crear una base de datos.

Luego de instalarlo, vamos a darle start a apache y star a MySQL. Luego a Admin en MySQL para iniciar la simulacion del servidor y gestor de base de datos.

En Nueva, creamos una base de datos. Ponemos nombre y recomendable poner el prefijo latin1\_spanish.ci.

Luego creamos la tabla. Le damos nombre y cantidad de columnas. Le tenemos que dar nombre a cada una y el tipo de dato que va a contener. Luego guardar.

En nuestro codigo, vamos a tener que instalar la libreria mysql (npm i mysql) para que nuestro proyecto pueda entender ese lenguaje y poder conectarnos con la base de datos. Luego importar mysql con require(‘mysql’).

Primero tenemos que conectar nuestro servidor a la base de datos de la siguiente manera:

const mysql = require('mysql');

const conexion = mysql.createConnection({

    host:'localhost',

    user:'root',

    password:'',

    database:'curso\_nodejs'

})

Ademas podemos verificar la conexion a la base de datos y cometer accion con respecto a eso:

Entonces al conectar, si hay error lanza un mensaje sino lanza un exito.

conexion.connect((error)=>{

    if(error){

        console.log("Se ha producido un error al conectarse a la base de datos");

    }else{

        console.log("Conexion exitosa");

    }

})

Cuando queramos mandar la informacion de un input a la base de datos, lo vamos a hacer gracias al controlador. Allí tenemos que programar en sql dentro del metodo query para poder insertar los datos en la db.

conexion.query('INSERT INTO personas (Nombre, Email) VALUE ("'+datos\_formulario.nombre+'","'+datos\_formulario.email+'")',(resultado)=>{console.log(resultado);})

Aquí vemos que utilizamos el INSERT INTO para insertar valores dentro de la tabla ‘personas’. Los datos que vamos a agregar son nombre e email que corresponden a lo que viene de datos\_formulario. Tambien lo veremos en consola.

(Recordar que en los parametros de VALUE poner el “ ‘valor‘ ”.)

Luego de todo esto, podemos ver en nuestra base de datos la informacion que nos brindó el usuario.

Organizando el MVC (Vease el archivo servidor-mvc.js y afines)

En este punto vamos a desglosar el codigo que hicimos en este archivo servidor-mvc.js para que quede mas organizado siguiendo el principio de mvc.

Primero vamos a ordenar las rutas. Creamos la carpeta ‘routes’. En esta carpeta vamos a guardar las invocaciones o llamadas a vistas de la pagina web. Dentro de esa carpeta vamos a crear por ejemplo contacto\_Route.js. Ahi ponemos todo lo que hicimos de control de rutas, junto con los require necesarios. En ese archivo con exportar la variable que hicimos servidor\_web, es suficiente. La exportamos con module.exports = conexion;

Entonces en servidor-mvc.js vamos a crear el servidor y dentro le agregamos las rutas.

CLASE 4.10 GRABACION 13/6/2023

Antes modularizamos las rutas en la carpeta ‘routes’. Ahora vamos a separar el Modelo (interaccion con la base de datos). Lo vamos a hacer dentro de una carpeta que por lo general la llamaremos ‘model’. En nuestro ejemplo el archivo se llama contacto\_Model.js.

Luego exportamos la variable importante gracias al module.exports. Y luego la importamos en el archivo de la ruta que corresponde, en nuestro caso seria en contacto\_Routes.js.

Nos falta separar los controladores (controla lo que vamos a hacer con el proyecto, controla lo que vamos a hacer con los datos en cada ruta, si mostrarlas y/o interactuar con una base de datos si es necesario). Los controladores estan dentro de los pedidos get, post, etc.

Entonces, dentro de la carpeta controller, pondriamos todos los controladores. En este ejemplo creamos el ‘contacto\_enviaForm\_controller.js’ y ponemos dentro lo que iria dentro del POST correspondiente al formulario. Con el codigo modularizado, lo exportamos con ‘module.exports’ y tenemos que importar la funcion importante al contacto\_Routes.js que es donde se encuentra la llamada POST a la ruta especificada.

Cabe recalcar que en este controlador ademas se esta utilizando el modelo (model/contacto\_Model.js), gracias a este archivo se controla la conexion y verificacion del estado de la base de datos.

Entonces eso quedaría asi:

servidor\_web.route('/envia\_form')

.post(envia\_form\_controller);

Recapitulando la modularizacion MVC:

Entonces, tenemos como archivo principal conexpress, el servidor-mvc.js. En este tenemos las rutas que podemos tomar -->

cada ruta se indica que se va a hacer, hacen pedidos http --> (modularizadas en carpeta routes) -->

cada pedido http tiene un controlador --> (modularizadas en controller) -->

en cada controlador se determina si se llaman las views y si se interactua con model que interactua con la bd para luego el controlador pasarle datos a la view. Tambien puede mostrar la view sin tener que pasar por el model.

(views y model los trae de las carpetas con el mismo nombre).

Tambien. Otra cosa que podemos modularizar es la conexion a la bd. Ya que ese bloque de codigo va a estar presente en cada archivo de los modulos. Lo podemos guardar dentro de un nuevo archivo llamado conexion\_BD.js y guardarlo dentro de una carpeta llamada config. Luego importamos la variable de conexion a los archivos de la carpeta model.

Tambien nos habiamos olvidado de modularizar el control de la base de datos que hay en controller\_envia\_form() que comete la accion en sql. Como hace una consulta a la base de datos gracias al conexion.query, corresponde ponerlo en el archivo contacto\_Model.js

En este contacto\_Model.js, ademas de tener la query que inserta el nuevo usuario a la base de datos, podemos agregar otros query. Todos estos query los guardamos dentro de un objeto que vamos a exportar. Recordar que no podemos hacer varios exports, solo uno.

Cada query la vamos a guardar en una funcion anonima que va a tener por parametro dos cosas. Primero, los datos cargados por el usuario que queremos pasar por la query para interactuar con la base de datos.

Y segundo, una funcion que utilizaremos para manejar esa informacion luego de pasar por la base de datos, por ejemplo renderizar algo.

//pedidos que se van a utilizar en contacto

const contacto\_Operaciones={

    insert: function(datos,functionResultado){//Insertando datos de inputs a la bd

        conexion.query('INSERT INTO personas (Nombre, Email) VALUE ("'+datos.nombre+'","'+datos.email+'")',functionResultado);

        //El parametro 'datos' pasa la informacion dada en controlador y pasa a la query

        //El parametro 'functionResultado' va a ser una funcion que invocamos en este parametro en el controlador. Basicamente controla que hacer con el resultado luego de interactuar con la base de datos. La funcion que pondremos aqui controla el resultado y se renderizará algo de views si es necesario.

    },

    select: function(functionResultado){//Este functionResultado se puede manejar en otro lado para mapear los usuarios por ejemplo

        conexion.query('SELECT \* FROM personas', functionResultado);//Selecciona todos los elementos de la tabla personas y esa informacion se maneja con la functionResultado (no se hizo para el ejemplo)

    }

};

CLASE 4.11 (GRABACION 15/16/2023)

Vamos a trabajar en un proyecto nuevo aplicando MVC. Lo haremos en una nueva carpeta dentro de Fullstack-Bluee -> practica, lo llamaremos proyecto\_nodejs-4.11

Recordar hacer npm init para que este el package-json y modulos y poder instalar librerias con npm.

Hacer npm i express

Creamos un index.js. Importamos express, creamos un servidor.

Lo de importar express y crear el servidor, como es algo que haremos en la mayoria de archivos, podemos modularizarlo en un archivo para solamente importarlo y evitar copiar y pegar tanto. Lo haremos en el archivo servidor.js

Con use asignamos el motor de vista, en este caso ejs. Recordar instalar ejs con npm i ejs.

Recordar el listen(3000,‘localhost’)

Tambien lo empezamos a utilizar con use() y le asignamos el nombre de la ruta y el archivo de la ruta correspondiente. Para eso podemos crear la carpeta ‘routes’ y crearla ahi. En este ejemplo es usuario\_Routes.js.

Dentro de ese archivo ponemos todas las rutas relacionadas a esa seccion con sus respectivos pedidos http. Dentro de cada pedido tendra sus controladores que los llevara a la carpeta ‘controllers’. ASi con eso determinar si renderizar algo de las vistas o si interactuar con algo de ‘models’ para interactuar con la base de datos para luego mostrarlo en las vistas si es necesario.

Recordar que en los models va a ser necesario conectarnos a la base de datos mysql, para eso lo instalamos con npm i mysql. Luego la importamos y creamos la conexion con los datos requeridos.

//Incompleto el texto